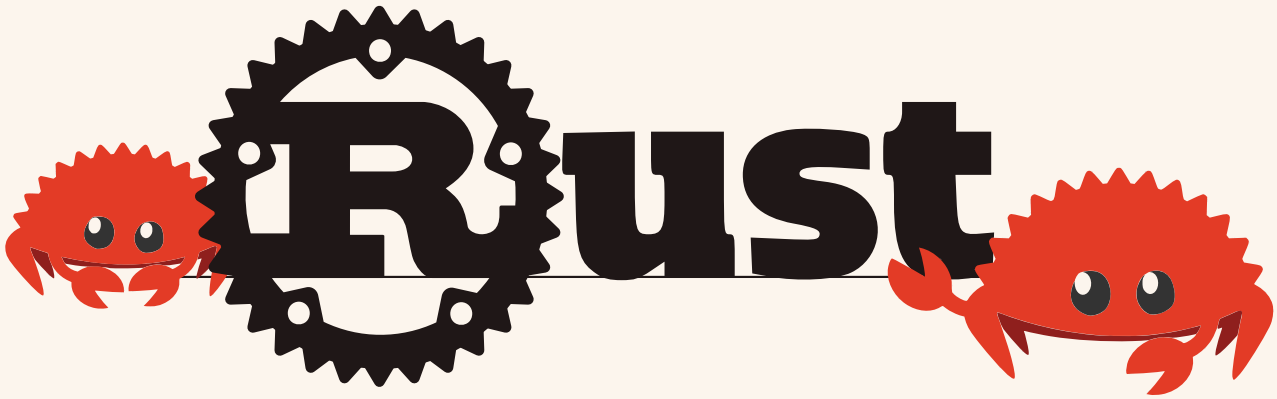


البرمجة بلغة رست

تأليف
مجتمع رست
ترجمة
ناصر داخل

أكاديمية حسوب

البرمجة بلغة رست

مرجع شامل إلى لغة رست Rust بكل تفاصيلها

اسم الكتاب: البرمجة بلغة رست

Book Title: The Rust Programming Language

المؤلف: ستيف كلابنيك وكارول نيكلز ومساهمات من مجتمع لغة رست

Author: Steve Klabnik & Carol Nichols with contributions from the Rust Community

المترجم: ناصر داخل

Translator: Naser Dakhel

المحرر: غفار الرفاعي - جميل بيلوني

Editor: Ghefar Alrefai - Jamil Bailony

تصميم الغلاف: جميل بيلوني

Cover Design: Jamil Bailony

سنة النشر: 2024

Publication Year: 2024

رقم الإصدار: 1.0

Edition: 1.0

https://academy.hsoub.com

academy@hsoub.com

عن الناشـر

أُنتج هذا الكتاب برعاية شركة حسوب وأكاديمية حسوب.

تهدف أكاديمية حسوب إلى تعليم البرمجة باللغة العربية وإثراء المحتوى البرمجي العـربي عـبر توفـير دورات برمجة وكتب ودروس عالية الجودة من متخصصين في مجال البرمجة والمجالات التقنية الأخـرى، بالإضـافة إلى توفير قسم للأسئلة والأجوبة للإجابة على أي سؤال يواجه المتعلم خلال رحلته التعليمية لتكون معه وتؤهلـه حـتى دخول سوق العمل.

حسوب شركة تقنية في مهمة لتطوير العـالم العـربي. تبـني حسـوب منتجـات تركِّز على تحسـين مسـتقبل العمل، والتعليم، والتواصل. تدير حسوب أكبر منصتي عمل حر في العالم العربي، مستقل وخمسات ويعمل في فيها فريق شاب وشغوف من مختلف الدول العربية.

دورة علوم الحاسوب

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

المحتويات باختصار

جدول المحتويات

فهرس الأشكال

فهرس الجداول

دورة تطوير التطبيقات
باستخدام لغة بايثون

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

تمهيد

أهلًا بك إلى لغة البرمجة رست، كتاب تمهيدي حول رست. تساعدك لغة رست في كتابة برمجيات أسرع وأكثر وثوقيّة، وعادةً ما يكون كل من أريحية استخدام مستوى مرتفع والتحكم الذي يقدمه لك المستوى المنخفض طريقان مختلفان يجب عليك الاختيار بينهما إلا أن رست تغير من ذلك الواقع عن طريق الموازنة بين القدرة التقنية الهائلة لها وتجربة المطوّر الجيدة، إذ تعطيك رست خيار التحكم بالتفاصيل منخفضة المستوى (مثل استخدام الذاكرة) دون أن يترتب على ذلك الاختيار أي شيء إضافي تجده عادةً.

هذا الكتاب مترجم عن الكتاب The Rust Programming Language من مساهمة ستيف كلابنيك Steve Klabnik وكارول نيكلز Carol Nichols ومساهمات من مجتمع لغة رست، ويعد مرجعًا شاملًا إلى لغة رست بكل تفاصيلها واستخداماتها.

كلمات الكاتب

تهتم لغة رست Rust بصورةٍ أساسية بتمكين المستخدم empowerment، وبالرغم من عدم وضوح ذلك للوهلة الأولى، تحاول رست دائمًا تمكينك على أنك مستخدم لها بغض النظر عن نوع الشيفرة البرمجية التي تكتبها، بحيث تستطيع البرمجة بكل ثقة في مجالات لم تعتد العمل بها.

خُذ على سبيل المثال العمل على مستوى الأنظمة system-level الذي يتعامل مع التفاصيل ذات المستوى المنخفض في إدارة الذاكرة memory management وتمثيل البيانات data representation والتزامن concurrency، إذ يُنظر إلى هذا المجال بكونه مجال غامض يُمكن فقط لعدد قليل من المبرمجين الذين كرّسوا سنواتٍ طوال لتعلمه وإتقانه التعامل معه، وحتى بالنسبة للعدد القليل من المبرمجين فإنهم يمارسون ويكتبون هذا النوع من الشيفرة البرمجية بحرص خوفًا من أن تكون الشيفرة البرمجية معرضةً للاستغلال الخبيث أو الأخطاء.

تتخلص لغة رست من هذه العقبات بتوفير مجموعة من الأدوات سهلة الاستخدام لمساعدتك خلال عمليـة البرمجـة والتطـوير، وتمكّن رسـت أيضًـا المـبرمجين الـذي يريـدون الغـوص في أعمـاق وتفاصـيل المسـتويات المنخفضة بواسطتها دون الخوف من خطر الأخطـاء أو الثغـرات الأمنيـة ودون تعلم التفاصـيل الدقيقـة الخاصـة بسلسلة من الأدوات المتغيرة، والأفضل من ذلك، فقد صُمّمَت اللغة بهدف توجيهك بصورةٍ طبيعية نحو شيفرة برمجية يمكن الاعتماد عليها وفعالة من ناحية السرعة واستخدام الذاكرة.

تسمح لغة رست للمبرمجين الذين يعملون في مجال الشيفرة البرمجيـة منخفضـة المسـتوى برفـع سـقف طموحـاتهم، إذ يُعـد تنفيـذ العمليـات على التـوازي parallelism في رسـت مثلًا عمليـةً منخفضـة الخطـورة إذ سيكشف المصرّف compiler أي أخطاء تقليدية تحـدث عـادةً لمسـاعدتك، كمـا أنـه يمكنـك من التعامـل مـع تحسين أكثر على شيفرتك البرمجية بثقة أكبر دون القلق بخصوص التسبب بأعطال أو نقاط ضعف.

لغة رست غير محدودةٍ ببرمجة الأنظمة منخفضة المستوى، فهي لغة مُعبّرة expressive ومريحة بقدرٍ كافٍ لجعلها مناسبةً لصنع تطبيقات واجهة سـطر الأوامـر CLI apps وخـوادم الـويب والعديـد من أنـواع الشـيفرات البرمجية الأخرى، التي تجعل منها تجربـةً ممتعـةً لكتابتهـا، وسـتجد بعض الأمثلـة للمجـالين المـذكورين ضـمن الكتاب لاحقًا. يسمح لك العمل مع رست ببناء مهـارات يمكن تطبيقهـا ونقلهـا من مجـال إلى آخـر؛ على سـبيل المثال، يمكنك تعلُّم رست بكتابة تطبيق ويب، ثم التعامـل مـع راسـبيري بـاي Raspberry Pi بالاسـتفادة من المهارات التي تعلمتها.

يشدد هذا الكتاب على إمكانية لغة رست لتمكين مستخدمها، فهو كتابٌ سهل القراءة بلغة بسـيطة وموجّـهٌ لمساعدتك، ليس فقط في معرفتك بلغة رست فحسب، بل للوصول إلى مبرمج بمستوًى عالٍ من الثقة عمومًا. لذا هيا بنا، استعدّ للتعلم وأهلًا وسهلًا بك ضمن مجتمع رست.

– نيكولاس ماتساكيس Nicholas Matsakis وآرون تورن Aaron Turon.

الفئة المستهدفة لرست

تُعـد رسـت لغـة برمجـة مثاليـة للعديـد من الأشـخاص لعـدة أسـباب. لنلقي نظـرةً على بعض مجموعـات المستخدمين وأهمها.

فرق المطورين

تثبت رست كونها أداةً عالية الإنتاجية على الدوام ضـمن فـرق التطـوير الكبـيرة في مسـتويات مختلفـة من معرفة برمجة الأنظمة. الشيفرة البرمجية منخفضة المستوى معرضةٌ إلى بعض الأخطاء صعبة التشخيص، والتي يمكن العثور عليها في لغات البرمجة الأخرى عن طريق التجريب المستمر ومراجعة الشـيفرة البرمجيـة بحـذر من قِبل مطوّرٍ ذي خبرة، إلا أن الأمر مختلـف في لغـة رسـت؛ إذ يلعب المصـرّف دور حـارس البوابـة gatekeeper يرفض تصريف الشيفرة البرمجية التي تحتوي على هذه الأخطاء بما فيها أخطاء الـتزامن concurrency bugs،

وبذلك يمكن لفريق المطورين بالتعاون مع المصرف تركيز وقتهم على التفكير بمنطق البرنـامج بـدلًا من ملاحقـة الأخطاء ومحاولة الكشف عنها.

تقدّم رست أيضًا مجموعةً من أدوات التطوير المعاصرة إلى عالم برمجة الأنظمة:

- أداة كارجو Cargo، التي تُعد بمثابة مدير اعتمادية dependency manager وأداة مُضمّنة، إذ تجعـل من إضافة وتصريف وإدارة الاعتماديات مهمةً سهلةً ومتناسقةً ضمن منظومة رست.
- أداة Rustfmt، التي تضمن تنسيقًا ثابتًا للشيفرة البرمجية لدى جميع المطورين.
- يشغّل خـادم لغـة رسـت بيئـة التطـوير المتكاملـة Integrated Development Environment -أو اختصارًا IDE- لإكمال الشيفرة البرمجية ورسائل الأخطاء المَضمّنة inline error messages.

يصبح عمل المطوّرين باستخدام هـذه الأدوات -وأدوات أخـرى- في منظومـة رسـت عملًا مثمـرًا عنـد كتابـة شيفرة برمجية من مستوى الأنظمة.

الطلاب

رست موجهة للطلاب ومن لديهم اهتمـام بتعلُّم المفـاهيم المتعلقـة بالأنظمـة، إذ يمكن للعديـد من النـاس -باستخدام رست- أن يتعلموا مواضيع مثل تطوير نظم التشغيل، وكما أن مجتمع اللغة سعيد بالإجابة عن أسـئلة الطلبة ويرحب بها. يطمح فريق تطوير رست عن طريق جهوده -بما فيها هذا الكتاب- بجعـل المفـاهيم المتعلقـة بالأنظمة سهلة الوصول إلى الناس وبالأخص من هم جدُد إلى البرمجة.

الشركات

تستخدم مئات الشركات الكبيرة والصغيرة رست في إنتاج البرمجيات وفي مهام مختلفة، التي تتضمن مثلًا أدوات سطر الأوامـر Command line tools وخـدمات الـويب وأدوات عمليـات التطـوير DevOps والأجهـزة المُدمجة embedded devices ومحركات البحث وتطبيقات إنترنت الأشياء Internet of things وتعلم الآلة Machine learning، وكذلك أجزاءً ضخمةً من متصفح فايرفوكس Firefox.

مطوري المصادر المفتوحة

رست موجهة للأشخاص الذين يريـدون بنـاء مجتمـع لغـة البرمجـة رسـت، وأدوات المطـورين والمكتبـات، وسنكون سعداء بمشاركتك في لغة البرمجة رست.

الأشخاص المهتمون بالسرعة والاستقرار

رست موجهة للأشخاص الذين يبحثون عن السرعة والثبـات في لغـة برمجـة، ونقصـد بالسـرعة هنـا سـرعة البرامج التي يمكنك إنشاؤها باستخدام رست وسرعة إنشاء هـذه الـبرامج وكتابتهـا. يتحقـق مصـرف رسـت من

الاستقرار عن طريق مزايا مُضافة وإعادة بناء التعليمات البرمجيـة refactoring، وذلـك على النقيض تمامًـا من الشيفرات البرمجية القديمة في لغات أخرى التي يتفادى المطورون تعديلها. أدى سعي لغة رسـت للوصـول إلى شيفرة برمجية مجرّدة وخصائص عالية المستوى تُصرَّف إلى شيفرة برمجية منخفضة المستوى بسـرعة الشـيفرة البرمجية المكتوبة يدويًا إلى جعل شيفرتها البرمجية الآمنة شيفرةً برمجيةً سريعةً في الوقت ذاته.

تأمل لغة رست بأن تدعم عدّة مجموعات من المستخدمين أيضًا، والمجموعات التي ذُكرت هنـا مـا هي إلا أكبر المجموعات الموجودة، كما تأمل رست عمومًا من التخلص من المقايضات الـتي يضـطر المـبرمج لإجرائهـا منذ قرون مضت، وذلك بتوفير الأمان والإنتاجية والسرعة ضـمن بيئـة العمـل. أعـطِ رسـت محاولـةً واطّلـع على الخيارات التي تناسبك.

الفئة المستهدفة من هذا الكتاب

يفترض هذا الكتاب أنك كتبت شيفرة برمجية مسبقًا بإحدى اللغات الأخرى، إلا أنه لا يفترض أي لغة برمجـة كانت تلك، وحاولنا جعل المحتوى قابل للفهم عمومًا قدر الإمكان بغض النظر عن نوع خلفية القارئ في البرمجة. لن نتكلم كثيرًا عن ماهية البرمجة أو كيف تستطيع التفكير بها، فإذا كنت جديدًا إلى البرمجة، فمن الأفضل قـراءة كتاب تمهيدي حول البرمجة على وجه الخصوص ونرشح لك كتاب تعلم البرمجة للمبتدئين لتبدأ به.

كيفية قراءة هذا الكتاب

يفترض هذا الكتـاب عمومًـا أنـك تقـرأه تسلسـليًا من البدايـة إلى النهايـة، إذ نقـدم مفاهيمًـا concept في الفصول اللاحقة بناءً على مفاهيم تكلمنا عنها في الفصول التي سبقتها، وإذا لم نوضّح بصورةٍ مفصلة المفـاهيم المقدمة في الفصول الأولى، فهذا يعني أننا سنخوض في تفاصيل أكثر عمقًا في فصول لاحقة.

ستجد نوعين من الفصول في هذا الكتاب، هما: الفصول التي تتنـاول المفـاهيم النظريـة، والفصـول الـتي تتناول المشاريع العملية؛ إذ سنتعلم في الفصول النظريـة عن جـانب معين من لغـة رسـت، بينمـا سـنبني في فصول المشاريع برامج بسيطة سويًّا بتطبيق ما تعلمناه إلى حد تلك النقطة. فصول المشاريع هي الفصـل الأول والثاني عشر والعشرين والفصول الأخرى هي فصول نظرية.

يشرح الفصل الأول كيفية تثبيت رست وكيفية كتابـة برنـامج "Hello, World!" وكيفيـة اسـتخدام كـارجو -مدير الحزم وأداة البناء الخاصة برست- بينما يمثّل الفصل الثاني تمهيدًا عمليًا للغة رست، إذ سـنغطي في هـذا الفصل المفاهيم من منظور مستوى عالي، ثم سنزودك بمزيـدٍ من التفاصـيل حـول هـذه المفـاهيم في فصـول لاحقة، وإذا أردت المباشرة بالتطبيق العملي فورًا، سيكون الفصل الثاني هو المكان المناسب لـذلك. قـد تـرغب بتجاوز الفصل الثالث الذي يغطي مزايا لغة رست المشابهة للغات البرمجة الأخرى والتوجه مباشـرةً إلى الفصـل الرابع لتعلم نظام ملكية رست Rust's ownership system، ولكن إن كنتَ متعلمًا يفضل معرفـة التفاصـيل

الدقيقة قبل تعلم ما هو مذكور في الفصول التالية، فقد ترغب بالعودة إلى الفصل الثاني بعد تجاوزه وإتمام الفصل الثالث وتطبيق ما تعلمته على المشروع الموجود في الفصل.

يناقش الفصل الخامس الهياكل structs والتوابع methods، بينما يغطي الفصل السادس المُعدّدات enums وتعابير "match" وبُنى التحكم بدفق البرنامج "if let"، وسنستخدم الهياكل والمعدّدات لإنشاء أنواع مخصصة في رست.

ستتعلم في الفصل السابع نظام وحدات رست Rust's module system وقوانين الخصوصية privacy rules بهدف تنظيم شيفرتك البرمجية وواجهة التطبيق البرمجية Application Programming Interface -أو اختصارًا API- العامة. يناقش الفصل الثامن أكثر هياكل البيانات استخدامًا في التجميعات collections والمضمّنة داخل المكتبة القياسية، مثل الشعاع vector والسلسلة النصية string والخرائط المُعمّاة hash maps. ينظر الفصل التاسع إلى فلسفة رست في التعامل مع الأخطاء وتقنياتها.

يناقش الفصل العاشر الأنواع المُعمّمة generics والسمات traits ودورات الحياة lifetimes التي تمنحك القوة لتعريف شيفرتك البرمجية وتطبيقها على عدة أنواع. الفصل الحادي عشر مخصص للاختبارات، وهي عملية ضرورية حتى مع وجود ضمانات أمان رست وذلك للتأكد من أن منطق البرنامج صائب. سنبني في الفصل الثاني عشر تطبيقنا الخاص لمجموعة من الخصائص الموجودة في أداة سطر الأوامر `grep` التي ستبحث عن نص معين داخل عدة ملفات، ولتنفيذ ذلك، سنلجأ إلى العديد من المفاهيم التي ناقشناها في الفصول السابقة.

ينظر الفصل الثالث عشر إلى المغلّفات closures والمكررات iterators وهي مزايا في لغة رست أتت من لغات برمجة عملية. سنفحص في الفصل الرابع عشر أداة كارجو بتعمق أكبر، وسنناقش أفضل الطرق لمشاركة مكتباتك مع الغير. نناقش في الفصل الخامس عشر المؤشرات الذكية smart pointers التي تقدمها المكتبة القياسية والسمات التي تمكّنها من تحقيق وظيفتها.

نستعرض في الفصل السادس عشر مجموعةً مختلفةً من الوحدات في البرمجة المتزامنة concurrent، وسنتحدث عن كيفية مساعدة رست لك في عملية برمجة برنامج يعمل على خيوط متعددة multiple threads دون أي قلق. ينظر الفصل السابع عشر إلى تشابه مصطلحات لغة رست مع مبادئ البرمجة كائنية التوجه object-oriented programming التي قد ألفتَ سماعها.

يمثل الفصل الثامن عشر مرجعًا إلى الأنماط patterns ومطابقتها، وهي طرق قوية تسمح لك بالتعبير عن الأفكار عن طريق برامج رست. يحتوي الفصل التاسع عشر على عددٍ من المواضيع المتقدمة، مثل رست غير الآمنة unsafe Rust والماكرو macro والمزيد عن دورات الحياة والسمات والأنواع والدوال والمغلّفات.

نُكمل في الفصل العشرين المشروع الذي سننفّذه على خادم ويب منخفض المستوى يعمل بخيوط متعددة.

أخيرًا، تتضمن بعض المُلحقات بعض المعلومات المفيدة حـول اللغـة بتنسـيق مشـابه للمراجـع، إذ يغطي الملحق (أ) كلمات رست المفتاحية، بينما يغطي الملحـق (ب) عوامـل operators رسـت والرمـوز symbols، ويغطي الملحـق (ج) السـمات القابلـة للاشـتقاق derivable traits الموجـودة في المكتبـة القياسـية، ويغطي الملحق (د) بعض أدوات التطوير المفيدة، وأخيرًا يشرح الملحق (هـ) إصدارات رست المختلفة.

لا يوجد هناك أي طريقة خاطئة في قراءة هذا الكتاب، فإذا أردت تجاوز بعض الفصول، فلا مشـكلة، إلا أنـك قد تحتاج إلى العودة إلى هذه الفصول إذا واجهت أي التباس، لذلك افعل ما يناسبك.

يُعد تعلم كيفية قراءة رسالة الخطأ التي يعرضها المصرّف جزءًا مهمًا من عملية تعلم لغة رسـت، إذ سـتدلك هذه الرسائل لتحقيق شيفرة برمجية تعمل بنجاح، وبناءً على ذلك سنوفّر بعض الأمثلة التي لا يمكن تصريفها مع رسالة الخطأ التي سيعرضها المصرف في كل حالة؛ لذلك لن تعمـل بعض الأمثلـة الموجـودة. تأكـد من قراءتـك للنص المحيط بالمثال لرؤية فيما إذا كان المثال الذي تحاول تشغيله يحتوي على خطأ مقصود أم لا. سيسـاعدك فيريس Ferris في التمييز بين الشيفرات البرمجية هذه:

فيريس Ferris أو فراس	المعنى
	لن تُصرّف هذه الشيفرة البرمجية.
	تشكو هذه الشيفرة البرمجية من خطب ما.
	لن تقدّم هذه الشيفرة البرمجية النتيجة المرجوة.

سنقدم في معظم الحالات الشيفرة البرمجية التي تعمل دون مشاكل أولًا، ومن ثم أي إصدار خاطئ منها.

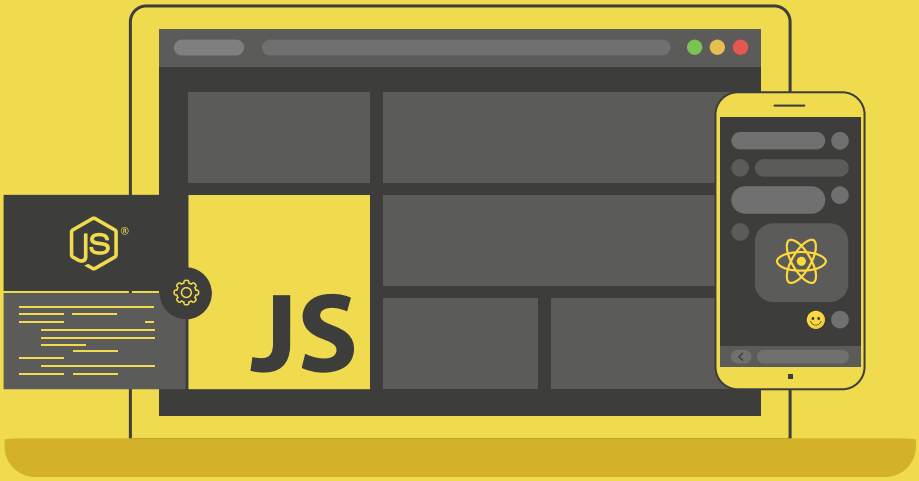
الشيفرة المصدرية

الملفات المصدرية المولّدة لهذا الكتاب موجودة على غيت هب Github.

المساهمة

يرجى إرسال بريد إلكـتروني إلى academy@hsoub.com إذا كـان لـديك اقـتراح أو تصـحيح على النسـخة العربية من الكتاب أو أي ملاحظة حول أي مصطلح من المصطلحات المسـتعملة. إذا ضـمَّنتَ جـزءًا من الجملـة التي يظهر الخطأ فيها على الأقل، فهذا يسهِّل علينا البحث، ويفضل إضافة أرقام الصفحات والأقسام أيضًا.

دورة تطوير التطبيقات
باستخدام لغة JavaScript



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

1. تعلم لغة رست Rust: البدايات

دعنا نبدأ رحلتنا مع لغة رست، إذ هناك الكثير لنتعلمه، وعلى كل رحلة أن تبدأ في مكـان مـا. سـنناقش في هذا الفصل كلًا من التالي:

- تثبيت لغة رست على لينكس Linux وماك أو إس macOS وويندوز Windows.
- كتابة برنامج يطبع العبارة "Hello, world!".
- استخدام أداة كارجو cargo، مدير حزم لغة رست package manager ونظام بنائها build system.

1.1 التثبيت

أولى خطواتنا هنا هي تثبيت لغة رست هي بتنزيل رست عن طريق أداة سطر الأوامـر `rustup`، الـتي تـدير إصدارات رست والأدوات المرتبطة بها، وستحتاج لاتصال بالإنترنت لهذه الخطوة.

> إذا كنت لا تفضل استخدام `rustup` لسبب ما، اطّلع على هذه الصفحة للمزيد من الخيارات لتثبيت لغة رست.

تثبّت الخطوات التالية آخر إصدارات لغة رست المستقرة، ويضمن لك ثبات رست تصـريف جميـع الأمثلـة الموجودة في هذا الكتاب حتى مع الإصدارات القادمة الجديدة، إلا أنه قـد يختلـف الخـرج قليلًا بين الإصـدارات، وذلك بسبب تطوير لغة رست على رسائل الخطأ والتحذيرات. بكلمات أخرى: سيعمل أي إصدار جديـد ومسـتقر من لغة رست تثبّته باتباع الخطوات التالية كما هو متوقع منه ضمن محتوى هذا الكتاب.

1.1.1 الإشارة إلى سطر الأوامر

سنستعرض بعض الأوامر المُستخدمة في الطرفية terminal ضـمن هـذا الفصـل والكتـاب ككُـل، إذ تبـدأ الأسطر التي يجب أن تُدخلها بالرمز "$"، وليس عليك هنا أن تكتب الرمز "$"، الذي يدل على بدايـة أمـر جديـد،

وعادةً ما تعرض الأسطر التي لا تبدأ بهذا الرمز الخرج لأمر سابق. إضافةً لما سبق، تستخدم الأمثلة التي تعتمد على صدفة PowerShell Shell خصوصًا الرمز ">" بدلًا من "$".

1.1.2 تثبيت أداة سطر الأوامر rustup على لينكس أو ماك أو إس

إذا كنت تستخدم نظام لينكس أو ماك أو إس، افتح الطرفية لإدخال الأمر التالي:

```
$ curl --proto '=https' --tlsv1.3 https://sh.rustup.rs -sSf | sh
```

سينزّل الأمر السابق سكريبتًا script ويبدأ بتثبيت أداة `rustup` التي ستثبّت بدورها آخر إصدارات لغة رست المستقرة، وقد يُطلب منك كلمة المرور لحسابك. إذا انتهت عملية التثبيت بنجاح، ستجد السطر التالي:

```
Rust is installed now. Great!
```

ستحتاج أيضًا إلى **رابط linker**، وهو برنامج يستخدمه رست لضمّ الخرج المُصرَّف إلى ملف واحد، وسيكون موجودًا غالبًا لديك. يجب عليك تثبيت مصرف سي C الذي يتضمن عادةً على رابط، إذا حصلت على أخطاء رابط، كما أن مصرف سي مفيد أيضًا لاعتماد بعض حزم لغة رست الشائعة على شيفرة لغة سي.

يمكنك الحصول على مصرف سي على نظام ماك أو إس بكتابة الأمر التالي:

```
$ xcode-select --install
```

يجب أن يثبت مستخدمو نظام لينكس GCC أو Clang اعتمادًا على توثيق التوزيعة distribution؛ فإذا كنت تستخدم مثلًا توزيعة أوبنتو Ubuntu، فيمكنك تثبيت حزمة `build-essential`.

1.1.3 تثبيت أداة سطر الأوامر rustup على ويندوز

إن كنت تستخدم نظام ويندوز، اذهب إلى www.rust-lang.org/tools/install، واتبّع التعليمات لتثبيت لغة رست، إذ ستستلم في مرحلة ما من مراحل التثبيت رسالةً مفادها أنك ستحتاج إلى أدوات بناء C++ الخاصة ببرنامج فيجوال ستوديو Visual Studio إصدار 2013 أو ما بعده، والطريقة الأسهل في الحصول على أدوات البناء هذه هي بتثبيتها مباشرةً من Visual Studio 2022، وتأكد من اختيار "C++ build tools" عند سؤالك عن أي الإصدارات التي تريد تثبيتها وتأكد أيضًا من تضمين حزمة Windows 10 SDK وحزمة اللغة الإنجليزية إلى جانب أي حزمة لغة أخرى من اختيارك.

يستخدم باقي الكتاب أوامر تعمل في كلٍّ من "cmd.exe" وصدفة PowerShell، وإذا كانت هناك أي فروقات معينة سنشرح أيهما يجب أن تستخدم.

1.1.4 التحديث والتثبيت

تُعد عملية التحديث بعد تثبيت لغة رست باستخدام `rustup` عمليةً سهلة، فكل ما يجب عليك فعله هو تنفيذ سكريبت التحديث من الصدفة:

```
$ rustup update
```

لإزالة تثبيت لغة رست وأداة `rustup`، نفذ سكريبت إزالة التثبيت من الصدفة:

```
$ rustup self uninstall
```

1.1.5 استكشاف الأخطاء وإصلاحها

للتأكد من أنك أنهيت عملية تثبيت لغة رست بنجاح، افتح الصدفة وأدخل السطر التالي:

```
$ rustc --version
```

من المفترض أن تجد رقم الإصدار وقيمة الإيداع المُعمّاة commit hash وتاريخ الإيداع لآخر إصدار مستقر جرى إطلاقه بالتنسيق التالي:

```
rustc x.y.z (abcabcabc yyyy-mm-dd)
```

إذا ظهر لك السطر السابق، فهذا يعني أنك ثبّتتَ لغة رست بنجاح؛ وإذا لم تجد هذه المعلومات وكنت تستخدم نظام ويندوز، فتأكد أن رست موجود في متغير النظام system variable المسمى `%PATH%` كما يلي:

إذا كنت تستخدم نظام تشغيل ويندوز، اكتب في واجهة سطر أوامر CMD ما يلي:

```
> echo %PATH%
```

أما في صدفة PowerShell، استخدم السطر التالي:

```
> echo $env:Path
```

وفي نظام تشغيل لينكس وماك أو إس، استخدم:

```
$ echo $PATH
```

إذا كان كل شيء صحيحًا، ورست لا يعمل فهناك عددٌ من الأماكن التي تستطيع الحصول منها على مساعدة، إذ يمكنك مثلًا طرح مشكلتك في قسم الأسئلة والأجوبة في أكاديمية حسوب أو إن أردت يمكنك التواصل مع فريق لغة رست مباشرة عبر صفحة التواصل.

1.1.6 التوثيق المحلي

يتضمن تثبيت لغة رست أيضًا نسخةً محليةً من التوثيق documentation لتتمكّن من قراءتها دون اتصال بالإنترنت، ولفتح النسخة ضمن المتصفح، نفذ الأمر `rustup doc`.

استخدم توثيق الواجهة البرمجية في كل مرة تصادف نوعًا أو دالةً في المكتبة القياسية ولست متأكدًا ممّا تفعل أو كيف تستخدمها.

1.2 كتابة أول برنامج بلغة رست

الآن، وبعد الانتهاء من تثبيت لغة رست، يمكنك كتابة أول برنامج. من المتعارف عليه عند تعلم لغة برمجة جديدة هو كتابة برنامج بسيط يطبع السلسلة النصية "Hello, world!" على الشاشة، لذا دعنا ننجز ذلك.

> يفترض هذا الكتاب معرفتك بأساسيات سطر الأوامر، ولا تتطلب لغة رست طريقةً معينةً لكيفية تعديلك أو استخدامك للأدوات أو مكان وجود شيفرتك البرمجية، لذا يمكنك استخدام بيئة برمجية متكاملة integrated development environment -أو اختصارًا IDE- إذا أردت، ولك الحرية في اختيار ما هو مفضّل لك. هناك العديد من البيئات البرمجية المتكاملة التي تقدم دعمًا للغة رست، ويمكنك تفقّد توثيق البيئة البرمجية المتكاملة التي اخترتها لمزيدٍ من التفاصيل. يحاول فريق تطوير لغة رست مؤخرًا توجيه جهودهم نحو تمكين دعم جيد للبيئات البرمجية المتكاملة عن طريق `rust-analyzer`. يمكنك الاطلاع على الملحق (ج) لمزيدٍ من التفاصيل.

1.2.1 إنشاء مجلد للمشروع

ستبدأ بإنشاء مجلد لتخزين شيفرة لغة رست البرمجية، ولا يهمّ المكان الذي ستخزن الشيفرة فيه، إلا أننا نقترح إنشاء مجلد "projects" للتمارين والمشاريع الموجودة في هذا الكتاب ضمن المجلد الرئيس home.

افتح الطرفية وأدخِل الأوامر التالية لإنشاء مجلد "projects" ومجلد لمشروع "Hello, world!" ضمن المجلد "projects".

لمستخدمي نظام لينكس وماك أو إس وصدفة PowerShell على نظام ويندوز، أدخل التالي:

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

ولطرفية ويندوز CMD، أدخل التالي:

```
> mkdir "%USERPROFILE%\projects"
```

```
> cd /d "%USERPROFILE%\projects"
> mkdir hello_world
> cd hello_world
```

1.2.2 كتابة وتشغيل برنامج بلغة رست

أنشئ ملفًا مصدريًا جديدًا وسمّه باسم "main.rs"، إذ يجب أن تنتهي ملفات شيفرة لغة رست بالامتداد ".rs" دائمًا، وإذا كنت تستخدم أكثر من كلمة واحدة في اسم الملف، فاستخدم الشرطة السفلية underscore للفصل ما بين الكلمات، فعلى سبيل المثال استخدم الاسم "hello_world.rs" بدلًا من "helloworld.rs".

الآن افتح ملف "main.rs" الذي أنشأته لتوك وأدخِل الشيفرة التالية:

```
fn main() {
    println!("Hello, world!");
}
```

[الشيفرة 1: برنامج يطبع "Hello, world!"]

احفظ الملف واذهب مجددًا إلى نافذة الطرفية. أدخل الأوامر التالية لتصريف وتشغيل الملف إذا كنت تستخدم نظام لينكس أو ماك أو إس:

```
$ rustc main.rs
$ ./main
Hello, world!
```

إذا كنت تستخدم نظام ويندوز أدخل الأمر `.\main.exe` بدلًا من `./main`:

```
> rustc main.rs
> .\main.exe
Hello, world!
```

يجب أن تحصل على السلسلة النصية "Hello,world!" مطبوعةً على الطرفية بغض النظر عن نظام تشغيلك، وإن لم يظهر الخرج، فعُد إلى فقرة استكشاف الأخطاء وإصلاحها لطرق الحصول على مساعدة.

إذا حصلت على السلسلة النصية "Hello, world!"، تهانينا، فقد كتبت رسميًا أولى برامج لغة رست، مما يجعلك مبرمج لغة رست، أهلًا بك.

1.2.3 أجزاء برنامج رست

دعنا نطّلع بالتفصيل على الأشياء التي تحدث في برنامج "Hello, world!"، إليك أول الأجزاء:

```
fn main() {

}
```

تعرّف هذه السطور دالةً في لغة رست، ودالة `main` هي دالة مميزة؛ إذ أنها نقطة تشغيل الشيفرة البرمجيـة الأولية في أي برنامج رست تنفيذي. يصرح السطر الأول عن دالة تدعى `main` لا تمتلك أي معاملات ولا تُعيد أي قيمة، وإذا أردنا تحديد أي معاملات، نستطيع فعل ذلك عن طريق وضعها داخل قوسين "()".

لاحظ أيضًا أن متن الدالة محتوًى داخل أقواس معقوصة "{}"، إذ تتطلب لغة رست وجـود هـذه الأقـواس حول متن أي دالة، ويُعد وضع القوس المعقوص الأول على سطر تصريح الدالة ذاته مع ترك مسافة فارغة بينهما تنسيقًا مُحبّذًا.

> إذا أردت الالتزام بتنسيق موحّد ضمن جميع مشاريع رست الخاصة بك، يمكنك استخدام أداة تنسيق تلقائية تدعى `rustfmt` لتنسيق الشيفرة البرمجية بأسلوب معين (يمكنك الإطلاع على مزيدٍ من المعلومات عن `rustfmt` في الملحق (ج))، وقد ضمّن فريق تطوير لغة رست هذه الأداة مع توزيعة لغة رست القياسية، مثل `rustc`، لذا من المفترض أن تكون مثبتة مسبقًا على جهازك.

نجد داخل دالة `main` الشيفرة البرمجية التالية:

```
    println!("Hello, world!");
```

يفعل السطر السابق كامل العمل الـذي يهـدف إليـه برنامجنـا البسـيط، ألا وهـو طباعـة النص إلى الشاشـة وهناك أربع نقاط مهمة بهذا الخصوص يجب ملاحظتها هنا، هي:

أولًا، أسلوب لغة رست في التنسـيق هـو محـاذاة السـطر indent باسـتخدام أربـع مسـافات فارغـة وليس مسافة جدولة tab.

ثانيًا، تستدعي `println!` ماكرو رست، وإذا أردنا استدعاء دالة بدلًا من ذلك، فسنسـتخدم `println` (أي الاستغناء عن `!`)، وسنتناقش بخصوص الماكرو في لغة رست بمزيـد من التفاصـيل في الفصـل التاسـع عشـر، وكل ما عليك معرفته الآن هو أن استخدام `!` يعني أننا نستدعي ماكرو بـدلًا من الدالـة الاعتياديـة، وأن المـاكرو لا يتبع القوانين ذاتها التي تتبعها الدوال.

ثالثًا، السلسلة النصية "Hello, world!" تُمرّر وسيطًا argument للماكرو `println!`، ثمّ تُطبع السلسلة النصية على الشاشة.

رابعًا، نُنهي السطر بالفاصلة المنقوطـة (;) وذلـك يُشـير إلى انتهـاء ذلـك التعبـير expression وأن التعبـير التالي سيبدأ من بعده، وتنتهي معظم أسطر شيفرة لغة رست بالفاصلة المنقوطة.

1.2.4 تصريف البرنامج وتشغيله هما خطوتان منفصلتان

لقد شغّلت لتوّك برنامجًا جديد الإنشاء، دعنا ننظر إلى كل خطوة من هذه العملية بتمعّن.

قبل تشغيل برنامج رست، يجب عليك تصريفه باستخدام مصرف لغة رست بإدخال الأمر `rustc` وتمرير اسم الملف المصدري كما يلي:

```
$ rustc main.rs
```

إذا كان لديك معرفة سابقة بلغة C أو ++C، فستلاحظ أن هذا مماثل لاستخدام الأمر `gcc` أو `clang`. بعد التصريف بنجاح، يُخرج رست ملفًا ثنائيًا تنفيذيًا binary executable.

يمكنك رؤية الملف التنفيذي على نظام لينكس أو ماك أو إس أو صدفة PowerShell على ويندوز عبر تنفيذ الأمر `ls`، وستجد في نظامَي لينكس وماك أو إس ملفين، بينما ستجد ثلاث ملفات إذا كنت تستخدم صدفة PowerShell وهي الملفات ذاتها التي ستجدها عند استخدامك طرفية CMD على ويندوز.

```
$ ls
main  main.rs
```

أدخل في طرفية CMD على ويندوز ما يلي (يعني الخيار `/B` أننا نريد فقط عرض أسماء الملفات):

```
> dir /B
main.exe
main.pdb
main.rs
```

يعرض ما سبق شيفرة الملف المصدر بامتداد ".rs"، إضافةً إلى الملف التنفيذي (المسمى main.exe على ويندوز و main على بقية المنصات)، وعند استخدام ويندوز هناك ملف يحتوي على معلومات لتصحيح الأخطاء بامتداد ".pdb"، ويمكنك تشغيل الملف التنفيذي "main" أو "main.exe" على النحو التالي:

```
$ ./main # على ويندوز .\main.exe أو
```

إذا كان "main.rs" هو برنامج "Hello, world!" فسيطبع السطر السابق "Hello, world!" على طرفيتك.

قد لا تكون معتادًا على كون جزء التصريف خطوة مستقلة إذا كنت تألف لغة ديناميكية، مثل روبي Ruby، أو بايثون Python، أو جافاسكربت JavaScript. تعدّ لغة رست لغة مُصرّفة سابقة للوقت ahead-of-time compiled language، مما يعني أن البرنامج يُصرّف وينتج عن ذلك ملف تنفيذي يُمنح لأحد آخر، بحيث يمكن له تشغيله دون وجود لغة رست عنده، وعلى النقيض تمامًا إذا أعطيت أحدًا ما ملفًا بامتداد ".rb" أو

"py." أو "js." فلن يستطيع تشغيله إن لم يتواجد تطبيق روبي أو بايثون أو جافاسكربت مثبتًا عنده، والفارق هنا هو أنه عليك استخدام أمرٍ واحد فقط لتصريف وتشغيل البرنامج. تصميم لغات البرمجة مبني على المقايضات.

يُعد تصريف البرنامج باستخدام `rustc` فقط كافيًا للبرامج البسيطة، إلا أنك ستحتاج لإدارة جميع الخيارات مع زيادة حجم برنامجك وجعل مشاركة شيفرتك البرمجية مع الغير عملية أسهل، ولذلك سنقدّم لك في الفقرة التالية أداة كارجو Cargo التي ستساعدكَ في كتابة برامج لغة رست لها تطبيقات فعلية في الحياة الواقعية.

1.3 مرحبا كارجو

كارجو هو نظام بناء لغة رست ومدير حزم، إذ يعتمد معظم مستخدمي لغة رست على هذه الأداة لإدارة مشاريع رست لأنها تتكفل بإنجاز الكثير من المهام نيابةً عنك، مثل بناء شيفرتك البرمجية وتنزيل المكتبات التي تعتمد شيفرتك عليها وبناء هذه المكتبات (ندعو المكتبات التي تحتاجها شيفرتك البرمجية لتعمل بالاعتماديات dependencies).

لا تحتاج برامج رست البسيطة -مثل البرنامج الذي كتبناه سابقًا- إلى أي اعتماديات، لذا إذا أردنا بناء مشروع "Hello, world!" باستخدام كارجو، فسيستخدم فقط الجزء الذي يتكفل ببناء الشيفرة البرمجية ضمن كارجو لا غير. ستُضيف بعض الاعتماديات عند كتابتك لبرامج أكثر تعقيدًا، وإذا كنت تستخدم كارجو حينها، فستكون إضافة تلك الاعتماديات سهلةً جدًا.

يفترض هذا الكتاب بأنك تستخدم كارجو بما أن معظم مشاريع لغة رست تستخدمه؛ إذ يُثبَّت كارجو تلقائيًا مع رست إذا استخدمت البرنامج الرسمي لتثبيته والذي ناقشناه في فقرة التثبيت، وإذا ثبتت رست باستخدام طرق أخرى، تأكد من وجود كارجو بإدخال الأمر التالي إلى الطرفية:

```
$ cargo --version
```

إذا ظهر لك رقم الإصدار، فهذا يعني أنه موجود. إذا ظهرت رسالة خطأ مثل "command not found"، ألقِ نظرةً على توثيق طريقة التثبيت التي اتبعتها لمعرفة طريقة تثبيت كارجو بصورةٍ منفصلة.

1.3.1 إنشاء مشروع باستخدام كارجو

دعنا نُنشئ مشروعًا جديدًا باستخدام كارجو ونُقارن بين هذه الطريقة وطريقتنا السابقة في إنشاء مشروع "Hello, world!". انتقل إلى مجلد "projects" (أو أي اسم مغاير اخترته لتخزّن فيه شيفرتك البرمجية)، ثم نفذ الأوامر التالية (بغض النظر عن نظام تشغيلك):

```
$ cargo new hello_cargo
$ cd hello_cargo
```

يُنشئ الأمر الأول مجلدًا جديدًا باسم "hello_cargo" -إذ أننا اخترنا تسمية "hello_cargo" لمشروعنا- ومن ثم يُنشئ كارجو ملفاته في المجلد الذي اخترناه بذات الاسم.

اذهب إلى المجلد "hello_cargo" واعرض الملفات الموجودة فيه، ستجد أن كارجو قد أنشأ ملفين ومجلدًا واحدًا داخله، هم: ملف باسم "cargo.toml"، ومجلد "src"، وملف "main.rs"؛ كما أنه هيأ مستودع غيت Git جديد مع ملف ".gitignore". لن تُوَلّد ملفات غيت إذا استخدمت الأمر `cargo new` ضمن مستودع جيت موجود مسبقًا، ويمكنك تجاوز ذلك السلوك عن طريق استخدام الأمر `cargo new --vcs=git`.

> غيت git هو نظام شائع للتحكم بالإصدارات، ويمكنك تغيير نظام التحكم بالإصدارات عند تنفيذ الأمر `cargo new` بإضافة الراية `--vcs`. يمكنك تنفيذ الأمر `cargo new --help` إذا أردت رؤية الخيارات المتاحة.

افتح الملف "Cargo.toml" باستخدام محرر النصوص المفضل لديك، يجب أن تكون الشيفرة البرمجية بداخله مشابهة للشيفرة 2.

```
[package]
name = "hello_cargo"
version = "0.1.0"
edition = "2021"

[dependencies]
```

[الشيفرة 2: محتوى ملف Cargo.toml الناتج عن تنفيذ الأمر `cargo new`]

يمكنك الاطلاع على مزيدٍ من المفاتيح وتعريفاتها على الرابط.

هذا الملف مكتوب بتنسيق لغة TOML (اختصارًا للغة توم المُختصرة الواضحة Tom's Obvious, Minimal Langaug)، وهي لغة تنسيق كارجو.

يمثل السطر الأول `[package]` قسم ترويسة الذي يشير إلى أن ما يليه هي معلومات لإعداد الحزمة. نُضيف المزيد من الأقسام إذا أردنا إضافة المزيد من المعلومات إلى هذا الملف.

تُعدّ الأسطر الثلاث التالية المعلومات التي يحتاجها كارجو لتصريف برنامجك ألا وهي: اسم المشروع وإصدار لغة رست المُستخدم، وسنتحدث عن مفتاح `edition` في الملحق (ج).

السطر الأخير `[dependencies]` هو بداية جزء جديد يحتوي على أي اعتماديات يعتمد عليها مشروعك لعمله. يُشار إلى حزم الشيفرات البرمجية في لغة رست باسم الصناديق crates، ولا نحتاج هنا أي صناديق أخرى لهذا المشروع، ولكننا سنحتاج إلى صناديق إضافية في مشاريع لاحقة وعندها سنستخدم هذا القسم.

أما الآن فافتح الملف "main.rs" الموجود داخل المجلد "src" وألقِ نظرةً على ما داخله:

```
fn main() {
    println!("Hello, world!");
}
```

قد ولّد كارجو برنامج "Hello, world!" لك على نحوٍ مماثل للبرنامج الذي كتبناه سابقًا في الشيفرة 1، والاختلاف الوحيد حتى اللحظة بين مشروعنا السابق ومشروع كارجو هو أن كارجو أضاف الشيفرة البرمجية داخل مجلد "src" وأنه لدينا ملف الإعداد "Cargo.toml" في المجلد الرئيسي.

يتوقع كارجو بأن تتواجد ملفاتك المصدرية داخل المجلد "src"، وأن يكون المجلد الرئيسي للمشروع فقط للملفات التوضيحية README files والمعلومات عن رخصة البرنامج license information وملفات الضبط configuration files وأي ملفات أخرى غير مرتبطة بشيفرتك المصدرية مباشرةً. يساعدك استخدام كارجو في تنظيم ملفاتك إذ أن هناك مكان لكل شيء وكلّ شيء يُخزّن في مكانه.

يمكنك تحويل مشروعك إلى مشروع يستخدم كارجو إذا أنشأت مشروعًا جديدًا دون استخدام كارجو على نحوٍ مماثل لما فعلناه في مشروع "Hello, world!"، فكل ما عليك فعله هو نقل الشيفرة المصدرية الخاصة بالمشروع إلى مجلد "src" وإنشاء ملف "Cargo.toml" موافق لتفاصيل المشروع.

1.3.2 بناء وتشغيل مشروع كارجو

دعنا ننظر الآن إلى ما هو مختلف عندما نشغّل برنامج "Hello, world!" باستخدام كارجو. أنشئ مشروعك بإدخال الأمر التالي داخل المجلد "hello_cargo":

```
$ cargo build
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

يُنشئ الأمر السابق ملف تنفيذي ضمن المجلد "target/debug/hello_cargo" (أو "target\debug\hello_cargo.exe" على ويندوز) بدلًا عن مجلدك الحالي، ويمكنك تشغيل الملف التنفيذي باستخدام الأمر التالي:

```
$ ./target/debug/hello_cargo
Hello, world!
```

إذا مر كل ما سبق بنجاح فستُطبع "Hello, world!" على الطرفية. يتسبب تنفيذ الأمر `cargo build` للمرة الأولى بإنشاء كارجو لملف جديد في المجلد الرئيسي باسم "Cargo.lock" وهذا الملف يتابع إصدارات الاعتماديات المستخدمة في مشروعك، وبما أن هذا المشروع لا يحتوي على أي اعتماديات، فلن يكون هذا الملف ذا أهمية كبيرة، إلا أن هذا يعني أنه لا يوجد أي حاجة لتغيير محتويات هذا الملف يدويًا بل يتكفل كارجو بمحتوياته بدلًا عنك.

بنينا مشروعًا باستخدام `cargo build` وشغّلناه باستخدام الأمر `./target/debug/hello_cargo` إلا أنه يمكننا استخدام الأمر `cargo run` أيضًا لتصريف الشيفرة البرمجية وتشغيلها مما ينتج عن تشغيل الملف التنفيذ باستخدام أمر واحد فقط:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

يستخدم معظم المطورين `cargo run` لأنه أكثر ملاءمةً من تذكُّر تشغيل `cargo build` ثم استخدام المسار الكامل وصولًا للملف الثنائي (التنفيذي).

لاحظ أننا لم نرى هذه المرة أي خرج يشير إلى أن كارجو يصرّف "hello_cargo"، وذلك لأن كارجو لاحظ أن الملف لم يتغير وبالتالي فقد شغّل مباشرةً الملف الثنائي، وإلّا في حالة تعديل الشيفرة المصدرية سيعيد كارجو بناء المشروع قبل تشغيله وستجد خرجًا مشابهًا لما يلي:

```
$ cargo run
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.33 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

يحتوي كارجو أيضًا على أمر آخر وهو `cargo check`، ويتحقق هذا الأمر من شيفرتك البرمجية للتأكد من أنها ستُصرّف بنجاح ولكنه لا يولّد أي ملف تنفيذي:

```
$ cargo check
   Checking hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
```

لكن في أيّ الحالات لن تحتاج إلى ملف تنفيذي؟ يكون الأمر `cargo check` أسرع غالبًا في تنفيذه من الأمر `cargo build` لأنه يتخطى مرحلة توليد الملف التنفيذي، وبالتالي إذا أردت التحقق باستمرار من صحة شيفرتك البرمجية خلال كتابتها فاستخدام `cargo check` سيُسرّع العملية بصورةٍ ملحوظة. ينفّذ معظم مستخدمي لغة رست الأمر `cargo check` دوريًا عادةً للتأكد من صحة شيفرتهم البرمجية ومن أنها ستُصرّف دون مشاكل، ثمّ ينفذون الأمر `cargo build` عندما يحين الوقت لتوليد ملف تنفيذي واستخدامه.

دعنا نلخّص ما تعلمناه لحد اللحظة بخصوص كارجو:

- نُنشئ المشاريع باستخدام الأمر `cargo new`.

- نبني المشروع باستخدام الأمر `cargo build`.
- نستطيع بناء المشروع وتشغيله بأمر واحد وهو `cargo run`.
- يمكننا بناء المشروع دون توليد ملف تنفيذي ثنائي بهدف التحقق من الأخطاء في الشيفرة البرمجية باستخدام الأمر `cargo check`.
- يخزن كارجو ناتج عملية البناء في المجلد "target/debug" عوضًا عن تخزينها في مجلد الشيفرة البرمجية ذاتها.

الميزة الإضافية في كارجو هي أن الأوامر هي نفسها ضمن جميع أنظمة التشغيل التي تعمل عليها، وبالتالي ومن هذه النقطة فصاعدًا لن نزوّدك بنظام التشغيل بالتحديد (لينكس أو ماك أو إس أو ويندوز) لكل توجيه.

1.3.3 بناء المشروع لإطلاقه

يمكنك استخدام الأمر `cargo build --release` عندما يصل مشروعك إلى مرحلة الإطلاق لتصريفه بصورةٍ مُحسّنة، وسيولّد هذا الأمر ملفًا تنفيذيًا في المجلد "target/release" بدلًا من المجلد "target/debug"، ويزيد التحسين من سرعة تنفيذ شيفرتك البرمجية إلا أن عملية تصريفه ستستغرق وقتًا أطول، وهذا السبب في تواجد خيارين لبناء المشروع: أحدهما هو بهدف التطوير عندما تحتاج لبناء المشروع بسرعة وبصورةٍ دورية والآخر لبناء النسخة النهائية من البرنامج التي ستعطيها إلى المستخدم وفي هذه الحالة لن تُصرّف البرنامج دوريًا وسيكون البرنامج الناتج أسر ع ما يمكن. إذا أردت قياس الوقت الذي تستغرقه شيفرتك البرمجية لتُنفّذ، استخدم الأمر `cargo build --release` وقِس الوقت باستخدام الملف التنفيذي الموجود في المجلد "target/release".

1.3.4 أداة كارجو مثل أداة عرض

لن يقدّم لك كارجو في المشاريع البسيطة الكثير من الإيجابيات مقارنةً باستخدام الأمر `rustc`، إلا أن الفارق سيتضح أكثر حالما تعمل على برنامج معقدة تتألف من عدّة صناديق crates، وعندما تنمو البرامج لعدة ملفات أو تحتاج إلى اعتماديات، فمن الأسهل في هذه الحالة استخدام كارجو لتنسيق عملية بناء المشروع.

على الرغم من بساطة مشروع "hello_cargo" السابق إلا أنه يستخدم الآن الأدوات الواقعية والعملية التي ستستخدمها طوال مسيرتك مع لغة رست، وحتى تستطيع العمل على أي مشروع موجود مسبقًا، يمكنك استخدام الأوامر التالية للتحقق من الشيفرة البرمجية باستخدام غيت، ثمّ انتقل إلى مجلد المشروع وابنه:

```
$ git clone example.org/someproject
$ cd someproject
$ cargo build
```

لمزيدٍ من المعلومات حول كارجو، انظر إلى التوثيق.

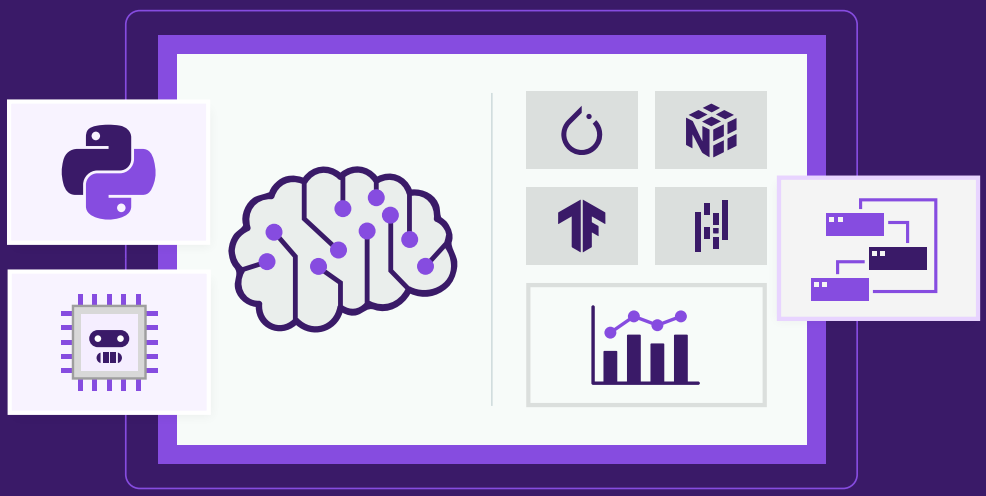
1.4 خاتمة

لقد قطعتَ شوطًا كبيرًا ضمن رحلتك مع لغة رست في هذا الفصل، إذ تعلمنا ما يلي:

- تثبيت آخر إصدارات لغة رست المستقرة باستخدام `rustup`.
- تحديث إصدار لغة رست الموجود إلى آخر جديد.
- فتح التوثيق المحلي (دون اتصال بالإنترنت).
- كتابة وتشغيل برنامج "Hello, world!" باستخدام `rustc` مباشرةً.
- إنشاء وتشغيل برنامج جديد باستخدام أداة كارجو.

حان الوقت المناسب لبناء برنامج أكثر واقعية للاعتياد على قراءة وكتابة شيفرة رست. لذا، سنبني في الفصل التالي برنامج لعبة تخمين، لكن إذا أردت تعلم أساسيات البرمجة في لغة رست، انتقل إلى الفصل الذي يليه مباشرةً.

دورة الذكاء الاصطناعي



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

2. برمجة لعبة تخمين أعداد

دعنا نتعرّف إلى رست بالعمل على مشروع عملي سويًّا، إذ سيقدم هذا الفصل بعض المفاهيم الشائعة في رست وكيفية استخدامها في برامج حقيقية، وسنتعلم كل من `let` و `match` والتوابع methods والدوال المترابطة associated functions واستخدام صناديق crates خارجية والمزيد، وسنناقش هذه التفاصيل بتعمق أكبر في فصول لاحقة، إلا أننا سنتدرب على الأساسيات في هذا الفصل.

سنعمل على برنامج بسيط وشائع للمبتدئين ألا وهو لعبة تخمين. إليك كيف سيعمل البرنامج: سيولّد البرنامج رقمًا صحيحًا عشوائيًا بين 1 و100، ثمّ سينتظر من اللاعب إدخال التخمين، ثم سيجيب البرنامج فيما إذا كان التخمين أكبر أو أصغر من الإجابة، وفي حال كان التخمين صحيحًا، سيطبع البرنامج رسالة تهنئة وينتهي البرنامج.

2.1 إعداد المشروع الجديد

اذهب إلى مجلد "directory" الذي أنشأناه في سابقًا لإعداد المشروع الجديد باستخدام أداة كارجو Cargo كما يلي:

```
$ cargo new guessing_game
$ cd guessing_game
```

يأخذ الأمر الأول `cargo new` اسم المشروع، وهو في حالتنا "guessing_game" مقل وسيطٍ أول، بينما ينتقل الأمر الثاني إلى مجلد المشروع.

ألقِ نظرةً إلى محتويات الملف "Cargo.toml" الناتج:

```
[package]
name = "guessing_game"
version = "0.1.0"
edition = "2021"


# يمكنك رؤية المزيد من المفاتيح من الرابط
https://doc.rust-lang.org/cargo/reference/manifest.html


[dependencies]
```

كما رأينا في سابقًا، يولّد الأمر `cargo new` برنامج "Hello, world!" لك. ألقِ نظرةً على محتويات ملف "src/main.rs":

```
fn main() {
    println!("Hello, world!");
}
```

دعنا الآن نصرّف هذا البرنامج ونشغله باتباع نفس الخطوة وباستخدام أمر `cargo run`:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 1.50s
     Running `target/debug/guessing_game`
Hello, world!
```

تبرز أهمية الأمر `run` عندما تريد أن تفحص التغييرات في مشروعك تباعًا وأن تفحص البرنامج بسرعة بعد كل إضافة قبل المضيّ قدمًا للإضافة التالية وهذا ما سنفعله بالضبط في لعبتنا هذه.

أعِد الآن فتح الملف "src/main.rs"، إذ سنكتب الشيفرة البرمجية لمشروعنا فيه.

2.2 معالجة التخمين

الجزء الأول من برنامج لعبة التخمين هو سؤال المستخدم ليدخل التخمين، ومن ثم معالجة هذا الدخل والتحقق من أنه بتنسيق مناسب. دعنا بدايةً نسمح المستخدم بإدخال تخمين. اكتب الشيفرة التالية في الملف "src/main.rs":

```
use std::io;
```

```
fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {guess}");
}
```

[الشيفرة1: شيفرة برمجية تأخذ التخمين من المستخدم في الدخل وتطبعه]

تحتوي الشيفرة البرمجية السابقة على كثيرٍ من المعلومات الجديدة، لذلك دعنا نراجعها سطرًا بسطر. علينا أن نستخدم المكتبة `io` وأن نضيفها إلى نطاق scope المشروع للحصول على دخل المستخدم ومن ثم طباعة نتيجة الدخل في الخرج؛ إذ تأتي مكتبة `io` مع المكتبة القياسية، التي تُعرف باسم `std`:

```
use std::io;
```

تحتوي رست افتراضيًا على مجموعةٍ معرفةٍ من العناصر ضمن المكتبة القياسية التي تُضاف إلى نطاق أي برنامج، وتُسمى هذه العناصر باسم المقدمة prelude ويمكنك رؤية جميع محتوياتها في توثيق المكتبة القياسية.

إذا أردت استخدام نوع محدد غير متواجد في المقدمة، فعليك إضافته إلى النطاق عن طريق استخدام تعليمة `use`. تتيح لك مكتبة `std::io` استخدام عدد من المزايا المفيدة منها القدرة على تلقّي دخل المستخدم.

دالة `main` هي النقطة التي يبدأ منها البرنامج كما رأينا في الفصل الأول:

```
fn main() {
```

تصرّح الصيغة `fn` عن دالة جديدة وتُشير الأقواس `()` إلى أن الدالة لا تأخذ أي معاملات، ويُشير القوس المعقوص `{` إلى بداية متن الدالة.

تشير `println!` إلى ماكرو كما تطرقنا إلى ذلك في الفصل الأول ويطبع هذا الماكرو السلسلة النصية إلى الشاشة:

```
    println!("Guess the number!");

    println!("Please input your guess.");
```

تطبع الشيفرة السابقة جملةً تدلّ المستخدم على ماهية اللعبة ومن ثم جملة تطلب منه إدخالًا.

2.3 تخزين القيم والمتغيرات

نُنشئ الآن متغيرًا لتخزين دخل المستخدم كما يلي:

```
let mut guess = String::new();
```

أصبح الآن برنامجنا أكثر إثارةً للاهتمام؛ فهناك الكثير من الأشياء التي تحدث عند تنفيذ هذا السطر القصير، إذ نستخدم تعليمة `let` لإنشاء متغير، إليك مثالًا آخر عن إنشاء متغير:

```
let apples = 5;
```

يُنشئ هذا السطر متغيرًا جديدًا باسم apples ويُسنده إليه القيمة 5. المتغيرات في لغة رست ثابتة immutable افتراضيًا، وهذا يعني أن المتغير سيحافظ على قيمته الأولية التي أُسندت إليه ولن تُغيّر وسنتحدث في هذا الموضوع بتوسع أكثر لاحقًا. لجعل المتغير قابلًا للتغيير mutable نستخدم الكلمة المفتاحية `mut` قبل اسم المتغير:

```
let apples = 5; // ثابت
let mut bananas = 5; // متغيّر
```

لاحظ أن `//` تتسبب ببدء تعليق سطري ينتهي بنهاية السطر وتتجاهل رست كل ما ورد ضمن التعليق، وسنناقش التعليقات لاحقًا بتوسع أكبر.

بالعودة إلى برنامج لعبة التخمين، فأنت تعلم الآن أن `let mut guess` سيُضيف متغيرًا يقبل التغيير باسم guess، وتُخبر إشارة المساواة `=` رست أنّنا نريد إسناد قيمة ما إلى المتغير، يقع على يمين إشارة المساواة القيمة التي نريد إسنادها إلى `guess` وهي قيمةٌ ناتجةٌ عن استدعاء الدالة `String::new` وهي دالة تُعيد نُسخةً instance من النوع "String"؛ وهو نوع من أنواع السلاسل النصية الموجود في المكتبة القياسية وهو نص بترميز UTF-8 وقابل للزيادة.

تُشير `::` في السطر `String::new` إلى أن new مرتبطةٌ بدالة من نوع String؛ و**الدالة المرتبطة associated function** هي دالة تُطبّق على نوع ما -وفي هذه الحالة هو String- وتُنشئ الدالة `new` هذه سلسلةً نصيةً جديدةً وفارغة، وستجد دالة `new` هذه في العديد من الأنواع لأنه اسم شائع لدالة تُنشئ قيمةً جديدةً من نوعٍ ما.

إذا نظرنا إلى السطر `let mut guess = String::new();` كاملًا، فهو سطرٌ لإنشاء متغير قابل للتغيير مُسندٌ إلى نُسخة جديدة وفارغة من النوع String.

2.4 تلقي دخل المستخدم

تذكر أننا ضمّننا إمكانية تلقي الدخل وعرض الخرج عن طريق `use std::io;` من المكتبة القياسية في السطر الأول من البرنامج. دعنا الآن نستدعي دالة `stdin` من وحدة `io` التي ستسمح لنا بالتعامل مع دخل المستخدم:

```
io::stdin()
        .read_line(&mut guess)
```

يمكننا استخدام استدعاء الدالة السابقة حتى لو لم نستورد مكتبة `io` بكتابتنا `use std::io` في بداية البرنامج ولكن الاستدعاء حينها سيكون بالشكل `std::io::stdin`. تُعيد الدالة `stdin` نسخة من النوع `std::io::Stdin` وهو نوع يُمثّل مُعالجًا للدخل القياسي من الطرفية.

يستدعي السطر `.read_line(&mut guess)` تابع `read_line` ضمن معالج الدخل القياسي للحصول على دخل المستخدم، كما أننا نمرّر `&mut guess` مثل وسيط إلى `read_line` للدلالة على السلسلة النصية التي سيُخزّن بها دخل المستخدم. تتمثّل وظيفة `read_line` بأخذ ما يكتبه المستخدم إلى الدخل القياسي وإلحاقه append بالسلسلة النصية (دون الكتابة فوق overwriting محتوياته)، ولذا فنحن نمرّر هنا السلسلة النصية وسيطًا، ويجب أن يكون الوسيط قابلًا للتغيير حتى يكون التابع قادرًا على تغيير محتويات السلسلة النصية.

يُشير الرمز `&` إلى أن هذا الوسيط يمثل مرجعًا، وهي طريقةٌ تسمح لأجزاء مختلفة من شيفرتك البرمجية بالوصول إلى الجزء ذاته من البيانات دون الحاجة إلى نسخ البيانات إلى الذاكرة عدة مرات. تُعد ميزة المراجع ميزةً معقّدةً وأكبر ميزات رست هو مستوى الأمان العالي وسهولة استخدام المراجع. لا تحتاج لمعرفة المزيد من هذه التفاصيل حتى تُنهي كتابة هذا البرنامج، إذ يكفي للآن أن تعرف أن المراجع غير قابلة للتغيير افتراضيًا -كما هو الحال في المتغيرات- وبالتالي يجب عليك كتابة `&mut guess` بدلًا من `&guess` إذا أردت جعلها قابلة للتغيير (سنشرح لاحقًا المراجع باستفاضة).

2.4.1 التعامل مع الأخطاء الممكنة باستخدام نوع النتيجة Result

ما زلنا نعمل على السطر البرمجي ذاته، ونناقش الآن السطر الثالث من النص، إلا أنه يجب الملاحظة أنه يمثل جزءًا من السطر البرمجي المنطقي ذاته. يمثل الجزء الثالث التابع:

```
.expect("Failed to read line");
```

كان بإمكانك كتابة السطر البرمجي على النحو التالي:

```
io::stdin().read_line(&mut guess).expect("Failed to read line");
```

إلا أن قراءة سطر طويل عملية صعبةٌ ومن الأفضل تقسيمه لأجزاء، لذلك من المحبّذ استخدام سطور جديدة ومسافات فارغة أخرى لتجزئة السطور الطويلة عندما تستدعي تابعًا على النحو التالي: `.method_name()`؛ دعنا الآن نناقش عمل السطر هذا.

تضع الدالة `read_line` كل ما يكتبه المستخدم إلى السلسلة النصية التي نمررها لها كما ذكرنا سابقًا، إلا أنها تُعيد أيضًا قيمة `Result` وهي مُعدّد enumeration وغالبًا ما يُختصر بكتابة enum؛ وهو نوع يُمكن أن يأخذ عدّة حالات ونسمّي كل حالة ممكنة له بمتغاير variant.

سنغطّي المعددات بتفصيل أكبر لاحقًا، إلا أن الهدف من أنواع `Result` هو لترميز معلومات التعامل مع الأخطاء.

متغايرات `Result` هي `Ok` و `Err`؛ إذ يشير مغاير `Ok` إلى نجاح العملية ويحتوي بداخله على قيمة النجاح المولّدة؛ بينما يشير المتغاير `Err` إلى فشل العملية ويحتوي بداخله على معلومات حول سبب أو كيفية فشلها.

لقيم النوع `Result` توابع معرفة لهم مثل أي قيم من نوع آخر، وتحتوي نسخةُ من النوع `Result` على التابع `expect` الذي يمكنك استدعاءه؛ فإذا كانت نسخة `Result` هذه لها قيمة `Err` فهذا يعني أن التابع `expect` سيتسبب بتوقف البرنامج وعرض الرسالة التي مرّرتها وسيطًا إلى التابع `expect`؛ وإذا أعاد التابع `read_line` قيمة `Err`، فهذا يعني أن الخطأ الناجم مرتبط بنظام التشغيل؛ وإذا كانت نسخة `Result` تحتوي على القيمة `Ok`، فسيأخذ التابع `expect` القيمة المُعادة التي تخزنها `Ok` ويُعيد القيمة إليك فقط كي تستخدمها، وتمثل القيمة في هذه الحالة عدد البايتات التي أدخلها المستخدم.

إذا لم تستدعي التابع `expect`، سيُصرّف البرنامج بصورةٍ طبيعية، ولكنك ستحصل على التحذير التالي:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `Result` that must be used
  --> src/main.rs:10:5
   |
|     io::stdin().read_line(&mut guess);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: `#[warn(unused_must_use)]` on by default
   = note: this `Result` may be an `Err` variant, which should be
handled
```

```
warning: `guessing_game` (bin "guessing_game") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.59s
```

يحذرك رست هنا أنك لم تستخدم قيمة `Result` المُعادة من التابع `read_line`، مما يعني أن البرنامج لن يستطيع التعامل مع الأخطاء ممكنة الحدوث.

الطريقة الصحيحة في تجنُّب التحذيرات هي بتطبيق طريقة معينة للتعامل مع أخطاء، إلا أننا نريد من البرنامج أن يتوقف في حالتنا هذه، لذا فيمكننا استخدام `expect`. ستتعلم ما يخص التعافي من الأخطاء (متابعة عمل البرنامج بعد ارتكاب الأخطاء) لاحقًا.

2.4.2 طباعة القيم باستخدام مواضع println! المؤقتة

هناك سطر واحد متبقي لمناقشته -بتجاهل الأقواس المعقوصة- ألا وهو:

```
    println!("You guessed: {guess}");
```

يطبع هذا السطر السلسلة النصية التي تحتوي دخل المستخدم، وتمثل مجموعة الأقواس المعقوصة `{}` مواضع مؤقتة placeholders. فكّر بالأمر وكأن `{}` كماشة سلطعون تُمسك القيمة في مكانها، ويمكنك طباعة أكثر من قيمة واحدة باستخدام الأقواس المعقوصة، إذ يدل أول قوسين على أول قيمة موجودة في لائحة بعد السلسلة النصية المُنسقّة، ويدل ثاني قوسين على القيمة الثاني في اللائحة وهلم جرًّا. إذا أردنا طباعة عدة قيم باستخدام استدعاء واحد للماكرو `println!` فسيبدو على النحو التالي:

```
let x = 5;
let y = 10;

println!("x = {} and y = {}", x, y);
```

سنحصل بعد تنفيذ الشيفرة السابقة على الخرج `x = 5 and y = 10`.

2.4.3 التأكد من عمل الجزء الأول

دعنا نتأكد من عمل الجزء الأول من لعبة التخمين، لذلك شغّل الشيفرة البرمجية باستخدام الأمر `cargo run`:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 6.44s
     Running `target/debug/guessing_game`
```

```
Guess the number!
Please input your guess.
6
You guessed: 6
```

وبهـذا تكـون قـد أنجـزت الجـزء الأول من اللعبـة، والبرنـامج الآن قـادر على اسـتقبال الـدخل من لوحة المفاتيح وطباعته.

2.5 توليد الرقم السري

الآن، علينا أن نولد الرقم السري الذي سيخمنه المستخدم، إذ يجب أن يكون هذا الرقم مختلفًـا في كـل مـرة حتى تكون اللعبة أكثر متعةً للعب في كل مـرة تحـاول التخمين. سنسـتخدم رقمًـا عشـوائيًا بين 1 و100 حـتى لا تكون اللعبة صعبةً جدًا. لا تتضمّن المكتبة القياسـية الخاصـة برسـت على مولّـد عشـوائي للأرقـام، إلا أن فريـق تطوير رست يقدم صندوق `rand` بهذه الوظيفة.

2.5.1 استخدام صندوق ما للحصول على إمكانيات أكبر

تذكر أن الصندوق هو مجموعة من ملفات رست المصـدرية، إذ يمثّـل المشـروع الـذي نبنيـه الآن صـندوقًا ثنائيًا binary crate أي أنه ملف تنفيذي، بينما يمثل صـندوق `rand` صـندوق مكتبـة crate library، أي أنـه يحتوي شيفرة مصدرية مكتوبة لتُستخدم في برامج أخرى ولا يمكن تشغيلها بصورةٍ مستقلة.

تبرز أداة كارجو عند تنسيقها للصناديق الخارجية. قبل كتابة الشيفرة البرمجيـة الـتي تسـتخدم `rand`، علينـا تعديل الملف "Cargo.toml" لتضمين صندوق `rand` مثل اعتمادية. افتح الملـف وأضِـف السـطر التـالي إلى نهاية الملف أسفل ترويسة القسم `[dependencies]` التي أنشأه لك كـارجو مسـبقًا، وتأكـد من تحديـد `rand` بدقة باستخدام رقم الإصدار وإلا فإن الشيفرة البرمجية الموجودة قد لا تعمل.

اسم الملف: Cargo.toml

```
rand = "0.8.3"
```

كُل ما يتبع ترويسة القسم في ملف "Cargo.toml" هو جزءٌ من قسم ما يستمر حتى بدايـة القسـم الآخـر، وفي القسم `[dependencies]` أنت تُعلم كارجو بالصناديق الخارجية التي يعتمد مشـروعك عليهـا وأي إصـدار منها يتطلّب، ونحدّد في حالتنا هذه الصـندوق `rand` ذو الإصـدار "0.8.3"، ويفهم كـارجو الإدارة الدلاليـة لنسـخ البرمجيات Semantic Versioning -أو اختصارًا SemVer- وهي صيغة قياسية لكتابة أرقـام الإصـدارات، وفي الحقيقة فإن الرقم "0.8.3" هو اختصارٌ للرقم "^0.8.3"، وهو يعني أن أي إصدار مسموح هو "0.8.3" على الأقل و"0.9.0" على الأكثر.

يضع كارجو في الحسبان أن هذه الإصدارات تحتوي على واجهات برمجية عامة public APIs متوافقة مع الإصدار "0.8.3" ويضمن ذلك التحديد أنك ستحصل على آخر الإصدارات المتوافقة مع الشيفرة البرمجية في هذا الفصل، إذ من غير المضمون أن تكون الإصدارات المساوية إلى "0.9.0" أو أعلى تحتوي على ذات الواجهة البرمجية التي نتبعها في الأمثلة هنا.

الآن ومن دون تغيير في الشيفرة البرمجية، دعنا نبني المشروع كما هو موضح في الشيفرة 2.

```
$ cargo build
    Updating crates.io index
  Downloaded rand v0.8.3
  Downloaded libc v0.2.86
  Downloaded getrandom v0.2.2
  Downloaded cfg-if v1.0.0
  Downloaded ppv-lite86 v0.2.10
  Downloaded rand_chacha v0.3.0
  Downloaded rand_core v0.6.2
   Compiling rand_core v0.6.2
   Compiling libc v0.2.86
   Compiling getrandom v0.2.2
   Compiling cfg-if v1.0.0
   Compiling ppv-lite86 v0.2.10
   Compiling rand_chacha v0.3.0
   Compiling rand v0.8.3
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53s
```

[الشيفرة 2: الخرج الناتج من تنفيذ الأمر `cargo build` بعد إضافة صندوق rand مثل اعتمادية]

قد تجد اختلافًا في أرقام الإصدارات (إلا أنها ستكون متوافقة مع الشيفرة البرمجية والشكر إلى SemVer) وسطورًا مختلفة (بحسب نظام التشغيل الذي تستخدمه) وقد تكون السطور مكتوبةً بترتيب مختلف.

يبحث كارجو عن آخر الإصدارات التي تحتاجها اعتمادية خارجية عند تضمينها وذلك من المسجل registry وهي نسخة من البيانات من Crates.io، وهو موقع ينشر فيه الناس مشاريع رست مفتوحة المصدر حتى يتسنى للآخرين استخدامها.

يتفقد كارجو قسم `[dependencies]` بعد تحديث المسجل ويحمّل أي صندوق موجود لم يُحمّل بعد. في حالتنا هذه وعلى الرغم من أننا أضفنا `rand` فقط مثل اعتمادية، فقد أضاف كارجو أيضًا صناديق أخرى يعتمد

`rand` عليها حتى يعمل، وبعد تحميل الصناديق يُصرّفها رست ويصرّف المشروع باستخدام الاعتماديات المتاحة.

إذا نفذت الأمر `cargo build` مجددًا دون أي تغيير فلن تحصل على أي خرج إضافي عن السطر `Finished`، إذ يعرف كارجو أنه حمّل وصرّف الاعتماديات وأنك لم تغيّر أي شيء بخصوصهم في ملف "Cargo.toml"، كما يعرف كارجو أنك لم تغيّر أي شيء على شيفرتك البرمجية ولهذا فهو لا يُعيد تصريفها أيضًا، وفي هذه الحالة لا يوجد أي شيء ليفعله ويغادر مباشرةً.

إذا فتحت الملف "src/main.rs" وعدّلت تعديلًا بسيطًا ومن ثمّ حفظته وحاولت إعادة بناء المشروع، فستجد السطرين التاليين فقط في الخرج:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

توضح السطور السابقة أن كارجو حدّث وبنى التغييرات الطفيفة إلى الملف "src/main.rs"، ويعلم كارجو أنه يستطيع إعادة استخدام الاعتماديات التي حمّلها سابقًا وصرّفها بما أنك لم تعدل عليها.

2.5.2 التأكد من أن المشاريع يمكن إعادة إنتاجها باستخدام ملف Cargo.lock

لدى كارجو آلية تتحقق من أنك تستطيع إعادة بناء الأداة كل مرة تبني أنت أو شخص آخر شيفرتك البرمجية، يستخدم كارجو فقط إصدارات الاعتماديات التي حددتها إلا إذا حددت عكس ذلك. على سبيل المثال، لنقل أن إصدار 0.8.4 من صندوق `rand` سيُطلق الأسبوع القادم، ويتضمن هذا الإصدار تصحيحًا مهمًا لمشكلة ما إلا أنه يحتوي أيضًا على تراجع regression، وسيتسبب هذا بتعطل شيفرتك البرمجية؛ تُنشئ رست في هذه الحالة ملفًا يدعى "Cargo.lock" عند أول تنفيذ للأمر `cargo build` ويقع هذا الملف في مجلد "guessing_game".

يوجِد كارجو جميع إصدارات الاعتماديات التي تلائم مشروعك عندما تبنيه للمرة الأولى، ومن ثم يكتب الإصدارات إلى ملف "Cargo.lock"، وبالتالي سيجد كارجو عندما تبني مشروعك في المستقبل أن الملف "Cargo.lock" موجود وسيستخدم عندها الإصدارات المحددة في ذلك الملف بدلًا من إيجاد الإصدارات المناسبة مجددًا، ويسمح لك هذا بالحصول على نسخة من المشروع قابلة لإعادة الإنتاج تلقائيًا، وبكلمات أخرى، سيظلّ مشروعك معتمدًا على الإصدار "0.8.3" حتى تقرّر التحديث إلى إصدار آخر بصورةٍ صريحة، ويعود الشكر إلى ملف "Cargo.lock" في ذلك. بما أن ملف "Cargo.lock" مهم للحصول على نسخ قابلة لإعادة الإنتاج، فمن الشائع أن يُضاف إلى نظام التحكم بالإصدارات version control مع باقي الشيفرة المصدرية في مشروعك.

2.5.3 تحديث صندوق للحصول على إصدار جديد

يقدم لك كارجو إمكانية تحديث صندوق ما باستخدام الأمر update، الذي سيتجاهل بدوره الملف "Cargo.lock" وسيبحث عن آخر الإصدارات التي تلائم متطلباتك في ملف "Cargo.toml"، إذ يكتب كارجو هذه الإصدارات إلى "Cargo.lock"، وإلا فسيبحث كارجو افتراضيًا عن إصدارات أحدث من "0.8.3" وأقدم من "0.9.0". إذا كان للصندوق rand إصداران جديدان هما "0.8.4" و"0.9.0" فستجد ما يلي عند تشغيل cargo update:

```
$ cargo update
    Updating crates.io index
    Updating rand v0.8.3 -> v0.8.4
```

يتجاهل كارجو الإصدار "0.9.0"، وستلاحظ أيضًا بحلول هذه النقطة أن ملف "Cargo.lock" يُشير إلى أن الإصدار الحالي من صندوق rand هو "0.8.4"؛ ولاستخدام الإصدار "0.9.0" من rand أو أي إصدار آخر من السلسلة "K"0.9.x عليك تحديث ملف "Cargo.toml" ليبدو على النحو التالي:

```
[dependencies]
rand = "0.9.0"
```

سيحدّث كارجو في المرة القادمة التي تنفذ فيها الأمر cargo build مسجّل الصناديق المتاحة ويُعيد تقييم متطلبات rand حسب الإصدار الجديد الذي حدّدته.

هناك الكثير من الأشياء التي يمكننا الحديث عنها بخصوص كارجو ونظامه، وهذا ما سنفعله لاحقًا، إلا أن ما ذكرناه الآن كافي مبدئيًا. يجعل كارجو عملية إعادة استخدام المكتبات عملية أكثر سهولة، ويمكّن مستخدمي لغة رست من كتابة مشاريع صغيرة تعتمد على عدد من الحزم.

2.6 توليد الرقم العشوائي

دعنا نبدأ باستخدام rand لتوليد الرقم العشوائي، إذ تكمن خطوتنا التالية في التعديل على محتويات الملف "src/main.rs" كما هو موضح في الشيفرة 3.

```
use std::io;
use rand::Rng;

fn main() {
    println!("Guess the number!");
```

```
    let secret_number = rand::thread_rng().gen_range(1..=100);

    println!("The secret number is: {secret_number}");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {guess}");
}
```

[الشيفرة 3: إضافة شيفرة برمجية لتوليد الرقم العشوائي]

نُضيف أولًا السطر `use rand::Rng`، إذ تُعرّف السمة `Rng` التوابع التي تستخدمها مولّدات الأرقام العشوائية، ويجب أن تكون هذه السمة ضمن النطاق حتى نستطيع استخدام هذه التوابع. سنناقش السمات بتفصيل أكبر لاحقًا.

ثم نُضيف سطرين في وسط البرنامج، إذ نستدعي في السطر الأول الدالة `rand::thread_rng` التي تُعطينا مولّد رقم عشوائي معيّن سنستخدمه وهو مولد محلي لخيط التنفيذ الحالي ويُبذر seeded بواسطة نظام التشغيل، ونستدعي بعدها التابع `gen_range` على مولد الأرقام العشوائية، التابع السابق معرّف بالسمة `Rng` التي أضفناها إلى النطاق باستخدام التعليمة `use rand::Rng`. يأخذ التابع `gen_rang` تعبير مجال range expression مثل وسيط، ويولّد رقمًا ينتمي إلى ذلك المجال، إذ نستخدم تعبير المجال من النوع ذو التنسيق `start..=end`، ويتضمن المجال الحد الأعلى والأدنى داخله، لذا بكتابتنا للمجال "1..=100" فنحن نحدّد الأعداد بين 1 و100.

> لن تعرف أي السمات وأي التوابع والدوال التي يجب أن تستدعيها من الصندوق من تلقاء نفسك، لذا يتضمن كل صندوق توثيق مرفقًا بتوجيهات لكيفية استخدام الصندوق. ميزة أخرى لطيفة من كارجو هي أن تنفيذ الأمر `cargo doc --open` سيتسبب ببناء التوثيق المزوّد بواسطة جميع الاعتماديات المحلية وفتحها في متصفحك، وإن كنت مهتمًا على سبيل المثال بالاستخدامات الأخرى الموجودة في الصندوق `rand` فكل ما عليك فعله هو تنفيذ الأمر `cargo doc --open` والنقر على `rand` في الشريط الجانبي على الجانب الأيسر.

يطبع السطر الجديد الثاني الرقم السري، وهذا مفيد بينما تُطوّر البرنامج حتى تكون قادرًا على تجربته، إلا أننا سنحذفه من الإصدار الأخير في نهاية المطاف، فاللعبة عديمة الفائدة إذا كانت تطبع الإجابة فور تشغيلها.

جرّب تنفيذ البرنامج عدة مرات:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53s
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4

$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

يجب أن تحصل على رقم مختلف في كل مرة ويجب أن تكون الأرقام بين 1 و100، وإذا حدث ذلك فأحسنت.

2.7 مقارنة التخمين مع الرقم السري

يمكننا الآن مقارنة التخمين الذي أدخله المستخدم مع الرقم السري العشوائي، وتوضّح الشيفرة 4 هذه الخطوة، لاحظ أن الشيفرة البرمجية لن تُصرّف بنجاح بعد كما سنوضح لاحقًا.

اسم الملف: src/main.rs

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    // --snip--
    println!("Guess the number!");
```

```
    let secret_number = rand::thread_rng().gen_range(1..=100);

    println!("The secret number is: {secret_number}");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {guess}");

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
}
```

[الشيفرة 4: التعامل مع الحالات المُمكنة ضمن عملية المقارنة]

نُضيف أولًا تعليمة `use` أخرى تقدّم لنا نوعًا جديدًا يدعى `std::cmp::Ordering` إلى النطاق من المكتبة القياسية، والنوع `Ordering` هو مُعدّد enum آخر يحتوي على المتغايرات `Less` و `Greater` و `Equal`، وهي النتائج الثلاث الممكنة عندما تُقارن ما بين قيمتين.

نُضيف بعدها خمسة أسطر في النهاية، وتستخدم هذه الأسطر بدورها النوع `Ordering`، ويُقارن التابع `cmp` بين قيمتين ويُمكن استدعاؤه على أي شيء يمكن مقارنته، ويتطلب الأمر استخدام المرجع للشيء الذي تريد مقارنته، وفي حالتنا هذه فهو يقارن `guess` مع `secret_number`، ثم يُعيد متغايرًا من متغايرات المعدّد `Ordering` الذي أضفناه سابقًا إلى النطاق باستخدام تعليمة `use`، ونستخدم هنا تعبير `match` لتحديد ما الذي سنفعله لاحقًا بناءً على متغاير `Ordering` المُعاد من استدعاء `cmp` باستخدام القيمتين `guess` و `secret_number`.

يتألف تعبير `match` من أذرع arms، ويتألف كل ذراع من نمط يُستخدم في عملية المقارنة والشيفرة البرمجية التي يجب أن تعمل في حال كانت القيمة المُعطاة إلى `match` توافق نمط الذراع. تأخذ رست القيمة

المُعطاة إلى `match` وتنظر إلى كـلّ نمـط ذراع. تُعـدّ الأذرع وبنيـة `match` من أبـرز مزايـا رسـت، إذ تسـمح لـك بالتعبير عن عدّة حالات قد تحدث ضمن شيفرتك البرمجية وأن تتأكد من معالجتها جميعًا، وسنغطّي هذه المزايا بتعمُّق أكبر لاحقًا.

دعنا نوضـح مثـالًا عن تعبـير `match` نسـتخدمه هنـا؛ لنقُـل أن المسـتخدم قـد خمّن القيمـة 50 وأن الـرقم العشوائي السري المولّد كان 38، تُقارن شيفرتنا البرمجية القيمـة 50 إلى 38 ويُعيـد التـابع `cmp` في هـذه الحالـة القيمة `Ordering::Greater` لأن 50 أكبر من 38، ويتلقّى التعبـير `match` القيمـة `Ordering::Greater` ويبدأ بتفقد كل نمط ذراع، إذ يُنظر إلى نمط الذراع الأولى وهو `Ordering::Less` وهي قيمةٌ لا توافـق القيمـة `Ordering::Greater` وبالتالي يجري تجاهل الشيفرة البرمجية ضمن الذراع وينتقل إلى نمط الـذراع الأخـرى وهو `Ordering::Greater` الذي يُطابق `Ordering::Greater!`، وبالتالي تُنفّذ الشيفرة البرمجية الموجود ضمن الذراع ويُطبع النص "Too big!" إلى الشاشة. ينتهي التعبير `match` بعد أول مطابقة ناجحة، لذا لن تجري المطابقة مع نمط الذراع الثالثة في هذه الحالة.

إلا أن الشيفرة 4 لن تُصرّف، دعنا نجرّب ذلك:

```
$ cargo build
   Compiling libc v0.2.86
   Compiling getrandom v0.2.2
   Compiling cfg-if v1.0.0
   Compiling ppv-lite86 v0.2.10
   Compiling rand_core v0.6.2
   Compiling rand_chacha v0.3.0
   Compiling rand v0.8.3
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types
  --> src/main.rs:22:21
   |
|     match guess.cmp(&secret_number) {
   |                     ^^^^^^^^^^^^^^^ expected struct `String`,
found integer
   |
   = note: expected reference `&String`
              found reference `&{integer}`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `guessing_game` due to previous error
```

تتلخّص المشكلة الأساسية بوجود **أنواع غير متوافقة mismatched types**. لدى رست نظام نوع ساكن static type قوي، إلا أنها تحتوي أيضًا على واجهة نوع type interface، وبالتالي استنتجت رست عند كتابتنا للتعليمة `let mut guess = String::new()` بأن `guess` يجب أن تكون `String` ولم تُجبرنا على كتابة النوع، بينما `secret_number` على الجانب الآخر هو من نوع عددي وهناك عدد من أنواع رست الرقمية التي يمكن أن تحتوي على القيم بين 1 و100، مثل `i32` وهو عدد بطول 32 بت، و `u32` وهو عديم الإشارة unsigned بطول 32 بت، و`i64` وهو عدد بطول 64 بت، إضافةً إلى أنواع أخرى، ويستخدم رست النوع `i32` افتراضيًا إن لم يُحدد النوع وهو نوع `secret_number` في هذه الحالة، والسبب في حدوث المشكلة هو عدم قدرة رست على المقارنة بين نوع عددي وسلسلة نصية.

إذًا، علينا أن نحوّل النوع `String` الذي يقرأه البرنامج في الدخل إلى نوع عددي، وذلك كي يتسنّى لنا مقارنته مقارنةً عدديّةً مع الرقم السري، ونُنجز ذلك عن طريق إضافة السطر الجديد التالي إلى متن الدالة `main`:

اسم الملف: src/main.rs

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    println!("The secret number is: {secret_number}");

    println!("Please input your guess.");

    // --snip--

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    let guess: u32 = guess.trim().parse().expect("Please type a
number!");
```

```
        println!("You guessed: {guess}");

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => println!("You win!"),
        }
    }
```

السطر الجديد هو:

```
let guess: u32 = guess.trim().parse().expect("Please type a number!");
```

نُنشئ متغيرًا باسم "guess"، ولكن مهلًا ألا يوجد متغير باسم "guess" في برنامجنا مسبقًا؟ نعم، ولكن رست تسمح لنا بتظليل shadow القيمة السابقة للمتغير `guess` بقيمة أخرى جديدة، ويسمح لنا التظليل بإعادة استخدام اسم المتحول `guess` بدلًا من إجبارنا على إنشاء متغير جديد، بحيث يصبح لدينا مثلًا `guess_str` و `guess`، وسنغطّي هذا الأمر بتفصيلٍ أكبر لاحقًا، ويكفي الآن أن تعرف بوجود هذه الميزة وأن استخدامها شائع عندما نريد تحويل قيمة من نوع إلى آخر.

نُسند المتغير الجديد إلى التعبير `guess.trim().parse()`، إذ تشير `guess` ضمن التعبير إلى المتغير `guess` الأصلي الذي يحتوي على الدخل (سلسلة نصية string). يحذف التابع `trim` عند استخدامه ضمن نسخة من النوع `String` أي مسافات فارغة whitespaces في بداية ونهاية السلسلة النصية، وهو أمر لازم الحدوث قبل أن نحوّل السلسلة النصية إلى النوع `u32` الذي يمكن أن يحتوي فقط على قيمة عددية، وذلك لأن المستخدم يضغط على زرّ الإدخال enter لإنهاء عمل التابع `read_line` بعد إدخال تخمينه مما يُضيف محرف سطر جديد إلى السلسلة النصية؛ فعلى سبيل المثال، إذا أدخل المستخدم 5 وضغط على زر الإدخال، فستأخذ السلسلة النصية `guess` القيمة "5\n"، إذ يمثل المحرف "\n" محرف سطر جديد (يتسبب الضغط على زر الإدخال في أنظمة ويندوز برجوع السطر وإضافة سطر جديد "\r\n") ويُزيل التابع `trim` المحرف "\n" أو "\r\n" ونحصل بالتالي على "5" فقط.

يحول التابع parse السلاسل النصية إلى نوع آخر، ونستخدمه هنا لتحويل السلسلة النصية إلى عدد، وعلينا إخبار رست بتحديد النوع الذي نريد التحويل إليه باستخدام `let guess: u32`، إذ تُشير النقطتان ":" الموجودتان بعد `guess` إلى أننا سنحدد نوع المتغير بعدها. تحتوي رست على عدد من الأنواع العددية المُضمّنة built-in منها `u32` الذي استخدمناه هنا وهو نوع عدد صحيح عديم الإشارة بطول 32-بت وهو خيار افتراضي جيّد للقيم الموجبة الصغيرة، وستتعلّم لاحقًا عن الأنواع العددية الأخرى. إضافةً إلى ما سبق، تعني `u32` في

مثالنا والمقارنة مـع `secret_number` أن رسـت سيسـتنتج أن `secret_number` يجب أن تكـون من النـوع `u32` أيضًا، لذا أصبحت المقارنة الآن بين قيمتين من النوع ذاته.

يعمـل التـابع `parse` فقـط على المحـارف الـتي يمكن أن تُحـوَّل منطقيًـا إلى أعـداد، لـذلك من الشـائع أن يتسبّب بأخطاء؛ فعلى سبيل المثال لن يستطيع التابع التحويل السلسلة النصية إلى نوع عددي إذا كانت تحتوي على القيمـة "A %" ولهـذا السـبب فـإن التـابع `parse` يُعيـد أيضًـا النـوع `Result` بصـورةٍ مماثلـة للتـابع `read_line`، الـذي ناقشـناه سـابقًا في فقـرة "التعامـل مـع الأخطـاء الممكنـة باسـتخدام النـوع `Result`"، وسنتعامل مع النوع `Result` هذا بطريقة مماثلة باستخدام تابع `expect` مجددًا. إذا أعاد التـابع `parse` متغـاير `Result` المتمثل بالقيمة `Err` فهذا يعني أنه لم يستطع التحويل إلى نوع عددي من السلسلة النصية، وفي هذه الحالة، سيوقف استدعاء `expect` اللعبة وستُطبع الرسالة التي نمررها له، بينما يُعيد متغاير `Result` ذو القيمة `Ok` إذا استطاع تحويل القيمة بنجاح من نوع سلسلة نصية إلى نوع عـددي، وتُعيـد عنـدها `expect` العـدد الـذي نريده من قيمة `Ok`.

دعنا نشغّل البرنامج الآن.

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
  76
You guessed: 76
Too big!
```

رائـع، فعلى الـرغم من أننـا أضـفنا مسـافات فارغـة قبـل التخمين، إلا أن البرنـامج توصّـل إلى أن تخمين المستخدم هو 67. شغّل البرنامج عدّة مرات أخرى لتتأكد من السـلوك المختلـف لحـالات مختلفـة من الإدخـال: خمّن العدد بصورةٍ صحيحة، خمّن عددًا أكبر من الإجابة، خمّن عددًا أصغر من الإجابة.

تعمل اللعبة لدينا الآن جيدًا، إلا أن المستخدم يمكنه التخمين مرةً واحدةً فقط، دعنا نغيّـر من ذلـك بإضـافة حلقة تكرارية loop.

2.8 السماح بعدة تخمينات باستخدام الحلقات التكرارية

تُنشئ الكلمة المفتاحية `loop` حلقةً تكراريةً لا نهائية، وسنستخدم الحلقة هنا بهـدف منح المسـتخدم فرصًـا أكبر في تخمين العدد:

اسم الملف: src/main.rs

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    // --snip--

    println!("The secret number is: {secret_number}");

    loop {
        println!("Please input your guess.");

        // --snip--


        let mut guess = String::new();

        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = guess.trim().parse().expect("Please type a
number!");

        println!("You guessed: {guess}");

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => println!("You win!"),
        }
```

```
    }
}
```

نقلنا محتوى البرنامج من تلقي الدخل guess إلى ما بعده لداخل الحلقة. تأكد من محاذاة السطور الموجودة داخل الحلقة التكرارية بمقدار أربع مسافات فارغة، وشغّل البرنامج مجددًا. سيسألك البرنامج الآن عن تخمين جديد إلى ما لا نهاية وهذه مشكلةٌ جديدة، إذ لن يستطيع المستخدم الخروج من البرنامج في هذه الحالة.

يمكن للمستخدم إيقاف البرنامج قسريًا عن طريق استخدام اختصار لوحة المفاتيح "ctrl-c"، إلا أن هناك طريقة أخرى للهروب من هذا البرنامج الذي لا يشبع، إذ يمكن للمستخدم أن يُدخل قيمة غير عددية كما ذكرنا في القسم "مقارنة التخمين إلى الرقم السري" الذي يناقش استخدام `parse` ويتسبب ذلك بتوقف البرنامج، ويمكننا الاستفادة من ذلك الأمر بالسماح لمستخدمنا بمغادرة البرنامج كما هو موضح هنا:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 1.50s
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread 'main' panicked at 'Please type a number!: ParseIntError
{ kind: InvalidDigit }', src/main.rs:28:47
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

نستطيع الآن مغادرة اللعبة بكتابة `quit`، إلا أنك ستلاحظ أن إدخال أي قيمة غـير عدديـة سيتسـبب بـذلك أيضًا، ولكن مشاكلنا لم تنتهي بعد، فما زلنا نريد أن نغادر اللعبة بعد أن نحصل على التخمين الصحيح.

2.9 مغادرة اللعبة بعد إدخال التخمين الصحيح

دعنا نبرمج اللعبة بحيث نُغادر منها عند فوز المستخدم بإضافة تعليمة `break`:

اسم الملف: src/main.rs

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    println!("The secret number is: {secret_number}");

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = guess.trim().parse().expect("Please type a
number!");

        println!("You guessed: {guess}");

        // --snip--

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
```

```
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

عند إضافة السطر `break` بعد "You win!"، يخرج البرنامج من الحلقة عندما يكون تخمين المستخدم مساويًا إلى الرقم السري، ويعني الخروج من الحلقة أيضًا الخروج من البرنامج لأن الحلقة هي آخر جزء من الدالة `main`.

2.10 التعامل مع الدخل غير الصالح

دعنا نجعل البرنامج يتجاهل دخل المستخدم عندما يكون ذو قيمة غير عددية بدلًا من إيقافه لتحسين اللعبة أكثر، وذلك ليتسنّى للمستخدم إعادة إدخال التخمين بصورةٍ صحيحة. يمكننا تحقيق ما سبق عن طريق تغيير السطر الذي يحتوي على تحويل `guess` من `String` إلى `u32` كما توضح الشيفرة 5.

اسم الملف: src/main.rs

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    println!("The secret number is: {secret_number}");

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();
```

```
            // --snip--

            io::stdin()
                .read_line(&mut guess)
                .expect("Failed to read line");

            let guess: u32 = match guess.trim().parse() {
                Ok(num) => num,
                Err(_) => continue,
            };

            println!("You guessed: {guess}");

            // --snip--


            match guess.cmp(&secret_number) {
                Ordering::Less => println!("Too small!"),
                Ordering::Greater => println!("Too big!"),
                Ordering::Equal => {
                    println!("You win!");
                    break;
                }
            }
        }
    }
```

[الشيفرة 5: تجاهل تخمين غير عددي وسؤال المستخدم عن تخمين آخر بدلًا من إيقاف البرنامج]

بدّلنا استدعاء التابع `expect` بتعبير `match` لتفادي إيقاف البرنامج والتعامل مع الخطأ. تذكر أن `parse` تُعيد قيمةً من نوع `Result` ويمثّل `Result` معدّدًا يحتوي على المغايرين `Ok` و `Err`. نستخدم هنا تعبير `match` بصورةٍ مماثلة لما فعلناه عند استخدام نتيجة `Ordering` في تابع `cmp`.

إذا نجح التابع `parse` بتحويل السلسلة النصية إلى عدد، فسيعيد القيمة `Ok` التي تحتوي على العدد الناتج، وستطابق قيمة `Ok` نمط الذراع الأول وبذلك سيعيد تعبير `match` قيمة `num` التي أنتجها التابع `parse` ووضعها داخل قيمة `Ok`، وسينتهي المطاف بهذا الرقم حيث نريده في متغير `guess` الجديد الذي أنشأناه.

إذا لم يكن التابع `parse` قادرًا على تحويل السلسلة النصية إلى عـدد، فسـيعيد قيمـةً من النـوع `Err` الـتي تحتوي بدورها على معلومات حول الخطأ، لا تُطابق قيمـة `Err` نمـط `Ok(num)` في ذراع `match` الأولى إلا أنهـا تطابق النمط `Err(_)` في الذراع الثانية، وترمز الشرطة السفلية `_` إلى الحصول على جميع القيم الممكنـة، وفي مثالنا هذا فنحن نقول أننا نريد أن نطابق جميع قيم `Err` الممكنة بغض النظر عن المعلومات الموجـودة داخلهـا، وبالتالي سينفذ البرنامج الذراع الثانية الـتي تتضـمن على `continue` الـتي تخـبر البرنـامج بالـذهاب إلى الـدورة الثانية من الحلقة `loop` وأن تسأل المستخدم عن تخمينٍ آخر، لذا أصبح برنامجنا يتجاهل جميـع أخطـاء `parse` الممكنة بنجاح.

يجب أن تعمل جميع أجزاء البرنامج كما هو متوقّع، لنجرّبه:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 4.45s
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

عظيم، استطعنا إنهاء كامل لعبة التخمين عن طريق تعديل بسيط، إلا أنه يجب أن تتذكر أن برنامجنا مـا زال يطبع المرقم السري، وذلك ساعدنا جدًا خلال تجربتنا للبرنامج وفحصه إلا أنه يُفسـد لعبتنـا، لـذا لنحـذف السـطر `println!` الذي يطبع الرقم السري على الشاشة. توضح الشيفرة 6 محتوى البرنامج النهائي.

اسم الملف: src/main.rs

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {guess}");

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

[الشيفرة 6: الشيفرة البرمجية للعبة التخمين كاملةً]

2.11 خاتمة

أنهينا بالوصول إلى هذه النقطة لعبة التخمين كاملةً، تهانينا.

كان هذا المشروع بمثابة تطبيق عملي وطريقة للتعرف على مفاهيم رست الجديدة، مثل `let` و `match` والدوال واستخدام الصناديق الخارجية وغيرها. ستتعلم المزيد عن هذه المفاهيم بالتفصيل فيما يتبع، إذ سنتكلم عن المفاهيم الموجودة في معظم لغات البرمجة، مثل المتغيرات وأنواع البيانات والدوال وسنستعرض كيفية استخدامها في لغة رست، ثم سنتوجه لمناقشة مفهوم الملكية ownership وهي ميزة تجعل من لغة رست مميّزة دونًا عن لغات البرمجة الأخرى، ومن ثمّ سنناقش صيغة syntax التابع والهياكل structs، ومن ثمّ سنشرح كيفية عمل المعدّدات enums.

3. مبادئ البرمجة الأساسية

سيغطّي هذا الفصل مفاهيمًا شائعة في جميع لغات البرمجة تقريبًا وكيفية عملها في لغة رست، إذ تتشارك جميع لغات البرمجة أشياء فيما بينها في عُمقها، ولن يكون أي من المفاهيم التي سنناقشها في هذا الفصل محصورةً على رست، إلا أننا سنُناقشها من منظور رست وسنشرح اصطلاحات اللغة في استخدام هذه المفاهيم.

ستتعلم على وجه الخصوص كل من المتغيرات variables والأنواع الأساسية basic types والدوال functions والتعليقات comments والتحكم بتدفق البرنامج flow control، وستكون هذه الأساسيات موجودةً في كل برنامج رست وسيعطيك تعلُّمها في مرحلة مبكرة أساسًا قويًا لتبدأ بالبناء عليه.

الكلمات المفتاحية

لدى لغة رست مجموعةٌ من الكلمات المفتاحية keywords المحجوزة لاستخدام اللغة فقط كما هو الأمر في لغات البرمجة الأخرى، وتذكر أنك لا تستطيع استخدام هذه الكلمات لأسماء للمتغيرات أو للدوال. تمتلك معظم الكلمات المفتاحية معنىً مميّزًا وستستخدمها لتحقيق مهام مختلفة ضمن برنامج رست الخاص بك. لا تمتلك بعض الكلمات المفتاحية أي مهمّة مخصصة لها حاليًا إلا أنه من الممكن أن تُضاف إلى رست في المستقبل. يمكنك رؤية جميع الكلمات المفتاحية في الملحق (أ).

3.1 المتغيرات والتعديل عليها

المتغيرات في رست غير قابلة للتعديل افتراضيًا كما ذكرنا في السابق، وهذه ميزة من ميزات لغة رست التي تهدف إلى جعل شيفرتك البرمجية المكتوبة آمنة وسهلة التزامن concurrency قدر الإمكان، إلا أنها تمنحك خيار التعديل mutable على المتغيرات إذا أردت ذلك. دعنا نرى لماذا تفضّل لغة رست جعل المتغيرات غير قابلة للتعديل في المقام الأول وما هي الحالات التي قد تريد التعديل عليها.

3.1.1 قابلية التعديل على المتغيرات

عندما تُسند أي قيمة إلى اسم متغيرٍ ما للمرة الأولى فهي غير قابلة للتغيير. لتوضيح ذلك دعنا نولّد مشروعًا جديدًا باسم "variables" في مجلد "projects" باستخدام الأمر `cargo new variables`.

بعد ذلك، افتح الملف "src/main.rs" في المجلد "variables" الجديد، واستبدل الشيفرة البرمجية الموجودة داخله بالشيفرة التالية. لن تُصرّف الشيفرة البرمجية هذه بعد، وسنفحص أولًا خطأ عدم قابلية التعديل immutability error.

اسم الملف: src/main.rs

```
fn main() {
    let x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

احفظ وشغّل البرنامج باستخدام الأمر `cargo run`، إذ يجب أن تتلقى رسالة خطأ كما هو موضّح في الخرج التالي:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
|     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
|     println!("The value of x is: {x}");
|     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
error: could not compile `variables` due to previous error
```

يوضح المثال أن المُصرّف يُساعدك لإيجاد الأخطاء ضمن البرنامج، ويمكن لأخطاء المصرّف أن تكون مُحبِطة ولكنها تعني أن برنامجك لا يفعل ما تريد فعله بأمان، وهذا لا يعني أنك مبرمج سيّء، إذ يحصل مستخدمو لغة رست المتمرسون على أخطاء تصريفيّة أيضًا.

تُشير رسالة الخطأ "`cannot assign twice immutable variable x`" إلى أنك لا تستطيع إعادة إسناد قيمة أخرى إلى المتغير غير القابل للتعديل `x`.

من المهم الحصول على أخطاء عند تصريف الشيفرة البرمجية عندما نحاول تعديل قيمة من المُفترض أن تكون غير قابلة للتعديل، فقد يؤدي هذا السلوك إلى أخطاء داخل البرنامج، ومن الممكن أن تتوقف بعض أجزاء البرنامج عن العمل بصورةٍ صحيحة إذا كان جزء من أجزاء برنامجك يعمل وفق الافتراض أن القيمة لن تتغير أبدًا وغيّر جزءٌ آخر هذه القيمة، وهذا النوع من الأخطاء صعب التعقّب عادةً بالأخص عندما يكون الجزء الثاني من البرنامج يُغيّر القيمة فقط في بعض الحالات. يضمن لك مصرّف لغة رست أن القيمة لن تتغيّر إذا حدّدتَ أنها لن تتغير، لذا لن يكون عليك تعقّب هذا النوع من الأخطاء بنفسك، وستكون شيفرتك البرمجية سهلة الفهم.

قد تكون قابلية التعديل مفيدة جدًا في بعض الحالات، وقد تجعل من شيفرتك البرمجية أسهل للكتابة، ويمكنك جعل المتغيرات قابلة للتعديل عكس حالتها الافتراضية باستخدام الكلمة `mut` أمام اسم المتغير، كما يُبرز استخدام الكلمة المفتاحية هذه أيضًا لقارئ الشيفرة البرمجية أن هذه القيمة ستتغير ضمن جزء ما من أجزاء البرنامج إلى قيمة جديدة.

على سبيل المثال، دعنا نغير محتويات الملف "src/main.rs" إلى التالي:

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

عند تشغيل البرنامج، نحصل على التالي:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/variables`
The value of x is: 5
The value of x is: 6
```

سُمح لنا الآن بتغيير قيمة `x` من 5 إلى 6 عند استخدام `mut`. يعتمد استخدام -أو عدم استخدام- قابلية التعديل على الشيء الأكثر وضوحًا في كل حالة بنظرك.

3.1.2 الثوابت

الثوابت constants هي قيم مُسندة إلى اسم ما ولا يُمكن تغييرها كما هو الحال في المتغيرات غير القابلة للتعديل، إلا أن هناك بعض الفروقات بين الثوابت والمتغيرات.

أولًا، ليس من المسموح استخدام `mut` مع الثوابت، إذ أنها غير قابلة للتعديل افتراضيًا وهي الحالة الدائمة لها، نصرّح ثابتًا باستخدام الكلمة المفتاحية `const` بدلًا من `let` ويجب تحديد نوع القيمة عندها. سنغطّي أنواع البيانات لاحقًا، لذا لا تقلق بخصوص التفاصيل الآن، فكل ما يجب معرفته هو أنه يجب علينا تحديد النوع دائمًا مع الثوابت.

ثانيًا، يُمكن التصريح عن الثوابت ضمن أي نطاق بما فيه النطاق العام global scope الذي يجعل منها قيمًا يمكن الاستفادة منها لأجزاء مختلفة من البرنامج.

ثالثًا، يمكن أن تُسند الثوابت فقط إلى تعابير ثابتة وليس إلى قيمة تُحسب عند تشغيل البرنامج.

إليك مثالًا عن تصريح ثابت ما:

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

اسم الثابت هو "THREE_HOURS_IN_SECONDS" وتُسند قيمته إلى ناتج ضرب 60 (الثواني في الدقيقة) مع 60 (الدقائق في الساعة) مع 3 (عدد الساعات التي نريد حسابها في هذا البرنامج)، تُسمّى الثوابت في لغة رست اصطلاحًا باستخدام أحرف كبيرة وشرطة سفلية underscore بين كل كلمة وأخرى. يستطيع المصرّف في هذه الحالة تقييم ناتج العمليات عند تصريف البرنامج، مما يسمح لنا بكتابة القيمة بطريقة أكثر بساطة وأسهل فهمًا للقارئ من كتابة القيمة 10,800 فورًا. راجع الفصل مرجع لغة رست قسم تقييم الثوابت لمزيدٍ من المعلومات عن العمليات الممكن استخدامها في تصريح الثوابت.

الثوابت صالحة للاستخدام طوال فترة تشغيل البرنامج وضمن النطاق التي صُرّحت فيه، مما يجعل من الثوابت أمرًا مفيدًا في تطبيقك الذي تتطلب أجزاء مختلفة منه معرفة قيمة معينة في الوقت ذاته، مثل عدد النقاط الأعظمي المسموح الحصول عليها من قِبل كل لاعب في لعبة أو سرعة الضوء.

تُفيدك عملية تسمية القيم المُستخدمة ضمن برنامجك مثل ثوابت في معرفة معنى القيمة لكل من يقرأ شيفرتك البرمجية، كما تُساعد في وجود القيمة في مكان واحد ضمن برنامجك بحيث يسهل عليك تعديلها مرةً واحدةً إذا أردت في المستقبل.

3.1.3 التظليل Shadowing

يُمكنك التصريح عن متغير جديد يحمل اسمًا مماثلًا لمتغير آخر سابق كما رأينا سابقًا عندما ناقشنا كيفية برمجة لعبة تخمين، ويقول مبرمجو اللغة عادةً أن المتغير الأوّل تظلّل shadowed بالثاني، مما يعني أن المتغير الثاني هو ما سيجده المُصرّف عندما تستخدم اسم المتغير من النقطة تلك فصاعدًا. بالمثل، فالمتغير الثاني ظَلّل overshadow الأول آخذًا استخدامات اسم المتغير لنفسه حتى يُظلّل المتغير الثاني بنفسه أو ينتهي النطاق الذي ينتمي إليه. يمكننا تظليل متغير ما باستخدام اسم المتغير ذاته وإعادة استخدامه مع الكلمة المفتاحية `let` كما هو موضح:

اسم الملف: src/main.rs

```
fn main() {
    let x = 5;

    let x = x + 1;

    {
        let x = x * 2;
        println!("The value of x in the inner scope is: {x}");
    }

    println!("The value of x is: {x}");
}
```

يُسند البرنامج أولًا المتغير `x` إلى القيمة 5، ثم يُنشئ متغيرًا جديدًا `x` بتكرار `let x =` آخذًا القيمة الأصلية ومُضيفًا إليها 1، وبالتالي تصبح قيمة `x` مساويةً إلى 6، ثم تُظلّل تعليمة `let` الثالثة ضمن النطاق الداخلي المحتوى داخل الأقواس المعقوصة curly brackets المتغير `x` وتنشئ متغير `x` مجددًا وتضرب قيمته السابقة بالرقم 2، فتصبح قيمة المتغير 12، وعند نهاية النطاق ينتهي التظليل الداخلي ويعود المتغير `x` إلى قيمته السابقة 6. نحصل على الخرج التالي عند تشغيل البرنامج:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/variables`
The value of x in the inner scope is: 12
The value of x is: 6
```

التظليل مختلف عن استخدام الكلمة المفتاحية `mut` لأنك ستحصل على خطأ تصريفي compile-time error إذا حاولت إعادة إسناد قيمة إلى هذا المتغير عن طريق الخطأ دون استخدام الكلمة المفتاحية `let`. يمكننا إجراء عدة تغييرات على قيمة ما باستخدام `let` وجعل المتغير غير قابل للتعديل بعد هذه استكمال |شهذه التغييرات.

الفارق الآخر بين `mut` والتظليل هو أنه يمكننا تغيير نوع القيمة باستخدام الاسم ذاته عندما نُعيد إنشاء متغير جديد عمليًا باستخدام الكلمة المفتاحية `let`. على سبيل المثال، لنقل أن برنامجنا يسأل المستخدم بأن يُدخل عدد المسافات الفارغة التي يُريدها بين نص معين بإدخال محرف المسافة الفارغة ومن ثمّ يجب أن نُخزّن هذا الدخل على أنه رقم:

```
fn main() {
    let spaces = "   ";
    let spaces = spaces.len();
}
```

متغير `spaces` الأول من نوع سلسلة نصية string بينما المتغير `spaces` الآخر من نوع عددي، إذًا، يوفّر علينا التظليل هنا عناء إنشاء متغير باسم جديد، مثل `spaces_str` و `spaces_num`، ويمكننا بدلًا من ذلك إعادة استخدام الاسم "spaces"، إلا أننا سنحصل على خطأ تصريفي إذا حاولنا استخدام `mut` في هذه الحالة:

```
fn main() {
    let mut spaces = "   ";
    spaces = spaces.len();
}
```

يُشير الخطأ إلى أنه من غير المسموح التعديل على نوع المتغير:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0308]: mismatched types
 --> src/main.rs:3:14
  |
|     let mut spaces = "   ";
  |                      ----- expected due to this value
|     spaces = spaces.len();
  |              ^^^^^^^^^^^^ expected `&str`, found `usize`

For more information about this error, try `rustc --explain E0308`.
```

```
error: could not compile `variables` due to previous error
```

الآن وبعد رؤيتنا لكيفية عمل المتغيرات، سننظر إلى أنواع البيانات الأخرى التي يمكننا استخدامها.

3.2 أنواع البيانات Data Types في لغة رست

تنتمي كل قيمة في لغة رست إلى نوع بيانات معيّن، ويُساعد ذلك لغة رست بمعرفة نوع البيانات التي تدلّ عليها هذه القيمة وكيفية التعامل معها، وسننظر إلى مجموعتين من أنواع البيانات، هي: القيم المُفردة scalar والقيم المركّبة compound.

تذكر أن لغة رست لغة برمجة متقيدة بأنواع البيانات statically typed، أي أنه يجب أن تعرف أنواع البيانات جميعها عند وقت التصريف، ويستطيع المصرّف عادةً استنتاج نوع المتغيرات بناءً على القيمة وكيفية استخدامها ضمن الشيفرة البرمجية، إلا أننا يجب أن نحدّد الأنواع في بعض الحالات، مثل التحويل من `String` إلى نوع عددي باستخدام `parse` كما رأينا في سابقًا في فقرة مقارنة التخمين إلى الرقم السري :

```
let guess: u32 = "42".parse().expect("Not a number!");
```

نحصل على الخطأ التالي إن لم نُضف النوع `u32` : كما هو موضح أعلاه، ويدل الخطأ على أن المصرّف يحتاج المزيد من المعلومات حول النوع الذي نريد استخدامه:

```
$ cargo build
   Compiling no_type_annotations v0.1.0
(file:///projects/no_type_annotations)
error[E0282]: type annotations needed
 --> src/main.rs:2:9
  |
|     let guess = "42".parse().expect("Not a number!");
  |         ^^^^^ consider giving `guess` a type

For more information about this error, try `rustc --explain E0282`.
error: could not compile `no_type_annotations` due to previous error
```

ستجد ترميزًا مختلفًا لكل من أنواع البيانات الأخرى.

3.2.1 الأنواع المفردة

يمثّل النوع المفرد scalar type قيمة فردية، ولدى لغة رست أربع أنواع مُفردة أولية هي: الأعداد الصحيحة integers والأعداد ذات الفاصلة العشرية floating-point numbers والقيم البوليانية booleans والمحارف

characters، وقد تتعرف على بعضها من لغة برمجة أخرى تعاملت معها سابقًا. دعنا نتحدّث عن كيفية استعمال هذه الأنواع في لغة رست.

١. أنواع الأعداد الصحيحة

العدد الصحيح integer هو عدد لا يحتوي على جزء كسري، وسبق لنا استخدام نوع من أنواع الأعداد الصحيحة سابقًا وهو `u32`، ويحدد التصريح عن هذا النوع أن القيمة المُسندة إلى المتغير ستكون عدد صحيح عديم الإشارة unsigned integer (تبدأ الأعداد الصحيحة ذات الإشارة بالحرف `i` بدلًا من `u`)، ويأخذ مساحة 32 بت. يوضّح الجدول 1 أنواع الأعداد الصحيحة المُضمّنة في لغة رست، ويمكننا استخدام أي من هذه المتغايرات variants للتصريح عن نوع قيمة العدد الصحيح.

الطول	ذو إشارة	عديم الإشارة
8-بت	`i8`	`u8`
16-بت	`i16`	`u16`
32-بت	`i32`	`u32`
64-بت	`i64`	`u64`
128-بت	`i128`	`u128`
يعتمد على معمارية الحاسب	`isize`	`usize`

الجدول 1: أنواع الأعداد الصحيحة في رست

يُمكن أن يكون كل متغاير ذو إشارة أو عديم إشارة وذو طول محدّد، إذ تُشير كلمة ذو إشارة signed وعديم الإشارة unsigned إلى إمكانية كون العدد سالبًا أم لا، وبتعبير آخر، هل يحتاج العدد إلى إشارةٍ معه (ذو إشارة signed) أم أنه سيكون موجبًا فقط وسيُمثّل بالتالي دون أي إشارة (عديم الإشارة unsigned). الأمر مماثل لكتابة الأعداد على ورقة، فعندما نحتاج لاستخدام الإشارة يُوضّح العدد وبجانبه إشارة (سواءً موجبة أو سالبة)، وفي حال كان الافتراض أن جميع الأعداد موجبة، فلا نضع أي إشارة بجانب الأعداد، وتُخزّن الأعداد ذات الإشارة باستخدام تمثيل المتمّم الثنائي two's complement.

يُمكن أن يخزِّن كل متغاير ذي إشارة القيم المنتمية إلى المجال من 1-2n- إلى 1 - 2n-1، إذ تمثّل "n" عدد البتات التي يستخدمها المتغاير، وبالتالي يمكن للنوع `i8` تخزين القيم التي تنتمي إلى المجال من 27- إلى 1-27 الذي يساوي من 128- إلى 127، بينما يمكن للمتغايرات عديمة الإشارة تخزين القيم ضمن المجال من 0 إلى 2n-1، وبالتالي يمكن للنوع `u8` أن يخزن الأعداد من 0 إلى 28-1 وهو ما يساوي المجال من 0 إلى 255.

إضافةً لما سبق، يعتمد النوعان `isize` و`usize` على معمارية الحاسب الذي يعمل عليه برنامجك، وهو بطول 64 بت إذا كان من معمارية 64 بت وبطول 32 بت إذا كان من معمارية 32 بت.

يُمكنك كتابة الأعداد الصحيحة المُجرّدة integer literals بـأي من التنسـيقات الموضـحة في الجـدول 2، لاحظ أن لغة رست توفر صياغة لكتابة الأعداد بطريقة تدل على نوعها لتُمثيل عدة أنواع عدديـة إذ تسـمح بوجـود لاحقة للنوع suffix type مثل "57u8" لتحديد نوعـه، ويُمكن أن تسـتخدم صـياغة الأعـداد تلـك أيضًـا الرمـز _ بمثابة فاصل بصري لجعل الأعداد أسهل للقراءة مثل "1_000" الذي يحمل القيمة "1000" ذاتها.

العدد المجرد	مثال
عشري	`98_222`
ست عشري	`0xff`
ثُماني	`0o77`
ثُنائي	`0b1111_0000`
بايت (فقط بحجم `u8`)	`b'A'`

الجدول 2: الأعداد الصحيحة المجردة في لغة رست

إذًا، كيـف يمكنـك معرفـة أي أنـواع الأعـداد الصـحيحة الـتي يجب عليـك اسـتخدامها؟ أنـواع لغـة رسـت الافتراضية هي الخيار الأمثل إذا لم تكُن متأكدًا بخصوص هذا الأمـر، نـوع العـدد الصـحيح الافتراضـي هـو `i32`، والحالة التي قد تستخدم فيها أحد النوعين `isize` أو`usize` هي عندما تستخدم قيمـة المتغـير دليلًا index مـا ضمن مجموعة collection.

طفحان الأعداد الصحيحة

بفرض أن هناك متغير من النوع `u8` الذي يمكنه تخـزين القيم من 0 إلى 255. إذا حـاولت إسـناد قيمـة إلى ذلك المتغير خارج النطاق المذكور -مثـل القيمـة 256- فسيتسـبب ذلـك بحـدوث مـا يسـمى طفحـان الأعـداد الصحيحة integer overflow الذي قد يتسبب بحدوث نتيجة من اثنتان.

تتفقد لغة رست عند تصـريف البرنـامج في نمـط تنقيح الأخطـاء debug mode حـالات طفحـان الأعـداد الصحيحة التي ستتسبب بهلع panic برنامجك عند تشغيله، ويستخدم مبرمجو لغة رست مصطلح **هلع panic** عندما يتوقف البرنامج بسبب خطأ ما، وسنناقش هذا الأمر بتعمق أكبر لاحقًا. لا تتحقـق لغـة رسـت من حـالات طفحان الأعداد الصحيحة التي تتسبب بهلـع البرنـامج عنـد تصـريفه باسـتخدام نمـط الإطلاق release mode باستخدام الراية flag `--release`، وتجري رست بـدلًا من ذلـك عمليـةً تُعـرف بانتقـال المتمم الثنـائي two's complement wrapping إذا حدث أي طفحان. باختصار، تنتقل القيمة التي تحتوي على قيمة أكبر من القيمة العظمى الممكن للنوع تخزينها إلى أصغر قيمة يمكن للمتغير تخزينها، فعلى سبيل المثال تصبح القيمة 256 في النوع `u8` مساويةً إلى الصفر والقيمة 257 إلى 1 وهكذا، لن يهلع البرنامج في هذه الحالـة، بـل سـيحمل المتغـير قيمةً مختلفة، ويُعدّ الاعتماد على عملية الانتقال wrapping في طفحان الأعداد الصحيحة خطأً.

يُمكنك استخدام أحد الطرق التالية للتعامل على نحوٍ صريح مع حالات الطفحان وهي طرق مُضمّنة في المكتبة القياسية للأنواع العددية الأولية:

- تمكين الانتقال في جميع أنماط بناء البرنامج باستخدام توابع `wrapping_*` مثل `wrapping_add`.
- إعادة القيمة `None` إذا لم يكن هناك أي طفحان باستخدام التوابع `checked_*`.
- إعادة القيمة والقيمة البوليانية التي تشير إلى حدوث طفحان باستخدام توابع `overflowing_*`.
- إشباع saturate القيم العُظمى والدُنيا للقيمة باستخدام توابع `saturating_*`.

ب. أنواع أعداد الفاصلة العشرية

لدى لغة رست نوعَين من أنواع أعداد الفاصلة العشرية floating-point numbers وهي الأعداد التي تحتوي على فواصل عشرية، وهما `f32` و `f64`، وبحجم 32 بت و64 بت، والنوع الافتراضي هو `f64`، لأنها تكون بنفس سرعة المُعالجات الحديثة `f32` ولكنها أكثر دقة، وجميع أنواع أعداد الفاصلة العشرية ذات إشارة.

إليك مثالًا يوضح أعداد الفاصلة العشرية عمليًا:

اسم الملف: src/main.rs

```
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

تُمثّل أعداد الفاصلة العشرية بحسب معيار IEEE-754. للنوع `f32` دقة وحيدة single-precision، بينما للنوع `f64` دقة مضاعفة double precision.

ج. العمليات على الأنواع العددية

تدعم لغة رست العمليات الرياضية الأساسية التي تتوقع إجرائها على الأنواع العددية، وهي الجمع والطرح والضرب والقسمة وباقي القسمة. يُقرّب ناتج قسمة الأعداد الصحيحة إلى أقرب عدد صحيح، وتوضح الشيفرة البرمجية التالية كيفية إجراء كل من العمليات باستخدام تعليمة `let`:

اسم الملف src/main.rs

```
fn main() {
    // الجمع
    let sum = 5 + 10;
```

```
    // الطرح
    let difference = 95.5 - 4.3;

    // الضرب
    let product = 4 * 30;

    // القسمة
    let quotient = 56.7 / 32.2;
    let floored = 2 / 3; // Results in 0

    // باقي القسمة
    let remainder = 43 % 5;
}
```

يستخدم كل تعبير من التعابير السابقة عاملًا رياضيًا ويُقيّم الناتج إلى قيمة واحدة، ثم تُسند هـذه القيمـة إلى المتغير. يحتوي الملحق (ب) على لائحة بجميع العوامل التي تحتويها رست.

د. النوع البولياني

للنوع البولياني type boolean في لغة رست -كما هو الحال في معظم لغـات البرمجـة الأخـرى- قيمتـان: `true` و `false`، ويبلغ حجم النوع هذا بتًا واحدًا، ويُحدّد النوع البولياني في لغة رسـت باسـتخدام الكلمـة `bool` كما يوضح المثال التالي:

اسم الملف: src/main.rs

```
fn main() {
    let t = true;

    let f: bool = false; // تحديد النوع بوضوح
}
```

الاستخدام الأساسي للقيم البوليانية هـو في التعـابير الشـرطية conditionals مثـل تعـابير `if`، وسـنغطّي تعابير `if` وكيفية عملها في لغة رست لاحقًا.

ه. نوع المحرف

نوع char في لغة رست هو أكثر أنواع القيم الأبجدية بدائية، إليك بعض الأمثلة عن تصريح قيم char:

اسم الملف: src/main.rs

```
fn main() {
    let c = 'z';
    let z: char = 'ℤ'; // تحديد النوع بوضوح
    let heart_eyed_cat = '😻';
}
```

لاحظ أننا حددنا النوع char المجرد باستخدام علامتَي تنصيص فردية، بعكس نوع السلسلة النصية string المجرّد الذي يستخدم علامتَي تنصيص مزدوجة، ويبلغ حجم النوع char في لغة رست أربعة بايتات وتمثل القيمة قيمة يونيكود Unicode عددية التي يُمكن أن تمثل قيمًا أكثر ممّا تستطيع الآسكي ASCII تمثيله. تتضمن لغة رست كذلك الأحرف المُعلّمة accented letters وكل من المحارف الصينية واليابانية والكورية، إضافةً إلى الرموز التعبيرية emoji والمسافات الفارغة ذات العرض الصفري zero-width space، إذ تُعد جميع القيم السابقة المذكورة قيمًا صالحة ويُمكن تخزينها في متغير من نوع char.

تتراوح قيم يونيكود العددية من "U+0000" إلى "U+D7FF" ومن "U+E000" إلى "U+10FFFF"، إلا أن مصطلح المحرف character غير موجود في نظام اليونيكود، وبالتالي يمكن ألا يتطابق فهمك كإنسان لماهية المحرف مع تعريف النوع char في لغة رست، وسنناقش هذا الموضوع بالتفصيل لاحقًا.

3.2.2 الأنواع المركبة

يُمكن للأنواع المركبة compound types أن تجمع عدّة قيم في نوع واحد، وللغة رست نوعان من الأنواع المركبة وهي المجموعات tuples والمصفوفات arrays.

ا. نوع المجموعة

المجموعة هي طريقة عامة لجمع عدّة قيم من أنواع مختلفة إلى نوع مُركب واحد، وللمجموعات حجم مُحدّد إذ لا يُمكن أن يكبر أو يصغر الحجم بعد التصريح عنه.

نستطيع إنشاء مجموعة عن طريق كتابة لائحة من العناصر يُفصل ما بينها بالفاصلة داخل قوسين، وكل موضع داخل هذه اللائحة يمثل قيمةً بنوع مُعيّن، ويمكن أن تختلف هذه الأنواع فيما بينها. أضفنا أنواع عناصر اللائحة في مثالنا التالي ولكن هذه الخطوة اختيارية:

اسم الملف: src/main.rs

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

يُسند المتغير tup إلى كامل المجموعة، لأن المجموعة تمثّل عنصرًا مركبًا واحدًا، وللحصول على القيم الفردية داخل المجموعة يمكننا استخدام مطابقة الأنماط pattern matching لتفكيك destructure قيمة المجموعة كما هو موضح:

اسم الملف: src/main.rs

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {y}");
}
```

يُنشئ هذا البرنامج مجموعة ويُسندها إلى المتغير tup، ومن ثم يستخدم نمطًا مع let لأخذ المتغير tup وتحويله إلى ثلاث قيم منفصلة وهي x و y و z، ويدعى هذا بالتفكيك destructuring لأنه يُفكك المجموعة الواحدة إلى ثلاث أجزاء، ويطبع البرنامج أخيرًا قيمة y المساوية إلى "6.4".

يمكننا أيضًا الوصول إلى عناصر المجموعة مباشرةً باستخدام النقطة (.) متبوعةً بدليل القيمة التي نريد الوصول إليها، كما هو موضح في المثال التالي:

اسم الملف: src/main.rs

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

يُنشئ هذا البرنامج مجموعةً باسم x، ثم يستخدم قيمة كل من عناصرها باستخدام دليل كل منها، ويبدأ الدليل الأول بالرقم 0 كما هو الحال في معظم لغات البرمجة.

للمجموعة اسم مميّز إذا كانت فارغة ألا وهو **الوحدة unit**، وتُكتب قيمتها وقيمة أنواعها بالشكل ()، اللتان تُمثّلان قيمة فارغة أو قيمة إعادة فارغة return type empty، تُعيد التعابير ضمنيًا قيمة الوحدة إذا لم يكن التعبير يُعيد أي قيمة أخرى.

ب. نوع المصفوفة

المصفوفة هي نوع من الأنواع الأخرى التي تحتوي على مجموعة من قيم متعددة، ويجب أن تكون جميع هذه القيم من النوع ذاته على عكس المجموعة، وللمصفوفات حجم ثابت بعكس بعض لغات البرمجة الأخرى.

نكتب القيم في المصفوفة مثل لائحة من القيم مفصول ما بينها بفاصلة داخل أقواس معقوفة square brackets:

اسم الملف: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

يُمكن للمصفوفات أن تكون مفيدةً عندما تريد من بياناتك أن تكون موجودةً على المكدّس stack بدلًا من الكومة heap (سنناقش المكدس والكومة لاحقًا) أو عندما تريد أن تتأكد أن هناك مجموعة ثابتة العدد من العناصر. المصفوفة ليست نوعًا مرنًا مثل نوع الشعاع vector، فالشعاع هو نوع مماثل يحتوي على مجموعة وهو مُضمّن في المكتبة القياسية ويمكن أن يتغير حجمه بالزيادة أو النقصان، وإن لم تكُن متأكدًا أيُّهما تستخدم، فذلك يعني أنك غالبًا بحاجة استخدام الشعاع، وسنناقش هذا الأمر بالتفصيل لاحقًا.

تبرز أهمية المصفوفات عندما تعرف عدد العناصر التي تحتاجها، على سبيل المثال إذا كنت تستخدم أسماء الأشهر في برنامج فمن الأفضل في هذه الحالة استخدام المصفوفة بدلًا من الشعاع لأنك تعلم أنك بحاجة 12 عنصر فقط:

```
let months = ["January", "February", "March", "April", "May", "June",
"July", August", "September", "October", "November", "December"];
```

يُكتب نوع المصفوفة باستخدام الأقواس المعقوفة مع نوع العناصر ومن ثم فاصلة منقوطة وعدد العناصر ضمن المصفوفة كما هو موضح:

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

يمثل النوع `i32` في مثالنا هذا نوع عناصر المصفوفة، بينما يمثل العدد "5" الذي يقع بعد الفاصلة المنقوطة عدد عناصر المصفوفة الخمس.

يمكنك تهيئة المصفوفة بحيث تحمل القيمة ذاتها لكافة العناصر عن طريق تحديد القيمة الابتدائية initial value متبوعةً بفاصلة منقوطة ومن ثم طول المصفوفة ضمن أقواس معقوفة، كما هو موضح:

```
let a = [3; 5];
```

ستحتوي المصفوفة `a` على 5 عناصر وستكون قيم العناصر جميعها مساوية إلى 3 مبدئيًا، وهذا الأمر مماثل لكتابة السطر البرمجي `let a = [3, 3, 3 ,3 ,3];` إلا أن هذه الطريقة مختصرة.

الوصول إلى عناصر المصفوفة

تُمثل المصفوفة جزءًا واحدًا معلوم الحجم من الذاكرة، والذي يُمكن تخزينه في المكدس، ويمكنك الوصول إلى عناصر المصفوفة باستخدام الدليل كما هو موضح:

اسم الملف: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

في مثالنا السابق، سيُسند إلى المتغير `first` القيمة الابتدائية 1 لأنها القيمة الموجودة في الدليل `[0]` ضمن المصفوفة، بينما سيُسند إلى المتغير `second` القيمة 2 لأنها القيمة الموجودة في الدليل `[1]` ضمن المصفوفة.

محاولة الوصول الخاطئ إلى عناصر المصفوفة

دعنا نرى ما الذي سيحدث إذا حاولت الوصول إلى عنصر من عناصر المصفوفة إذا كان ذلك العنصر يقع خارج المصفوفة بعد نهايتها، ولنقل أننا سننفّذ الشيفرة البرمجية التالية المشابهة للعبة التخمين الفصل السابق بالحصول على دليل المصفوفة من المستخدم:

اسم الملف: src/main.rs

```
use std::io;
fn main() {
    let a = [1, 2, 3, 4, 5];
```

```
    println!("Please enter an array index.");

    let mut index = String::new();

    io::stdin()
        .read_line(&mut index)
        .expect("Failed to read line");

    let index: usize = index
        .trim()
        .parse()
        .expect("Index entered was not a number");

    let element = a[index];

    println!("The value of the element at index {index} is:
{element}");
}
```

ستُصرّف الشيفرة البرمجية بنجاح، وإذا شغلت البرنامج باستخدام `cargo run` وأدخلت القيم 0 أو 1 أو 2 أو 3 أو 4، فسيطبع البرنامج القيمة الموافقة لهذا الدليل ضمن المصفوفة، إلا أنك ستحصل على الخرج التالي إذا حاولت إدخال قيمة أكبر من حجم المصفوفة (مثل 10):

```
thread 'main' panicked at 'index out of bounds: the len is 5 but the
index is 10', src/main.rs:19:19
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

تسبب البرنامج بخطأ عند التشغيل runtime error عند إدخال قيمة خاطئة إلى عملية الوصول للعناصر بالدليل، وانتهى البرنامج برسالة خطأ ولم يُنفّذ تعليمة `println!` الأخيرة. تتفقد لغة رست الدليل الذي حدّدته عند محاولتك الوصول إليه فيما إذا كان أصغر من حجم المصفوفة، وإذا كان الدليل أكبر أو يساوي حجم المصفوفة فسيهلع panic البرنامج، وتحدث عملية التفقد هذه عند وقت التشغيل خصوصًا في هذه الحالة وذلك لأن المصرف ربما لن يعرف القيمة التي سيدخلها المستخدم عند تشغيل الشيفرة بعد ذلك.

كان هذا مثالًا لمبادئ أمان ذاكرة رست بصورةٍ عملية، وتفتقر معظم لغات البرمجة منخفضة المستوى هذا النوع من التحقق، إذ يُمكن الوصول إلى ذاكرة خاطئة عندما تُعطي دليلًا خاطئًا في هذه اللغات. تحميك لغة رست من هذا النوع من الأخطاء بالخروج من البرنامج فورًا عوضًا عن السماح بالوصول إلى ذاكرة خاطئة

والاستمرار بالبرنامج، وسنناقش لاحقًا كيفية تعامل لغة رست مع الأخطاء وكيف يُمكنك كتابة شيفرة برمجية سهلة القراءة وآمنة بحيث لا يهلع البرنامج عند تنفيذها أو تسمح بالوصول إلى ذاكرة خاطئة.

3.3 الدوال Functions

تنتشر الدوال في معظم شيفرات رست البرمجية، وقد رأيت سابقًا واحدةً من أهم الدوال في اللغة ألا وهي دالة `main` وهي نقطة البداية للكثير من البرامج، كما أنك رأيت أيضًا الكلمة المفتاحية `fn` التي تسمح لك بالتصريح عن دالةٍ جديدة.

تستخدم شيفرة رست البرمجية نمط الثعبان snake case نمطًا اصطلاحيًا لأسماء الدوال والمتغيرات، إذ تكون الأحرف في هذا النمط جميعها أحرف صغيرة ويفصل ما بين الكلمة والأخرى شرطة سفلية. إليك برنامجًا يحتوي على مثال لتعريف دالة:

اسم الملف: src/main.rs

```
fn main() {
    println!("Hello, world!");

    another_function();
}

fn another_function() {
    println!("Another function.");
}
```

نعرّف الدالة في رست بإدخال الكلمة `fn` متبوعةً باسم الدالة يليها قوسين هلاليّين parentheses `()`، بينما تُخبر الأقواس المعقوفة curly brackets `{}` المصرف بموضع بداية وانتهاء متن الدالة.

يُمكننا استدعاء أي دالة عرفناها سابقًا بإدخال اسمها متبوعًا بقوسين هلاليّين، وبما أن الدالة `another_function` مُعرفةٌ في البرنامج، يمكننا استدعائها من داخل الدالة `main`. لاحظ أننا عرفنا `another_function` بعد دالة `main` في الشيفرة البرمجية إلا أنه يمكننا تعريفها قبلها أيضًا، إذ لا تُبالي رست بموضع تعريف الدوال طالما يوجد التعريف داخل النطاق scope الذي استدعيت الدالة منه.

دعنا نبدأ مشروعًا ثنائيًا project binary جديدًا باسم "functions" للنظر إلى الدوال بتعمّق أكبر، وضع مثال "another_function" السابق في ملف "src/main.rs" ونفّذه. يجب أن يظهر لك الخرج التالي:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.28s
     Running `target/debug/functions`
Hello, world!
Another function.
```

تُنفّذ هذه السطور البرمجية بالترتيب التي ظهرت فيه في الدالة `main`، أي تُطبع الرسالة "Hello, world!" أولًا، ثم تُستدعى الدالة `another_function` وتُطبع رسالتها.

3.3.1 المعاملات

يُمكننا تعريف الدوال بحيث تحتوي على معاملات parameters، وهي متغيرات خاصة تنتمي إلى بصمة الدالة function's signature، ويُمكنك استخدام قيم فعلية لهذه الدالة عند احتوائها على معاملات، وتُدعى هذه القيم بالوسطاء arguments إلا أنه غالبًا ما يُستخدم المصطلحان معامل ووسيط بصورةٍ تبادلية interchangeably لأي من المتغيرات في تعريف الدالة أو القيم الفعلية المُمرّرة للدالة عند استدعائها.

نُضيف معاملًا في هذا الإصدار من الدالة `another_function`:

اسم الملف: src/main.rs

```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {x}");
}
```

يجب أن تحصل على الخرج التالي عند تجربتك لتشغيل البرنامج:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 1.21s
     Running `target/debug/functions`
The value of x is: 5
```

يحتوي تصريح الدالة `another_function` على معامل واحد باسم `x` وهو من النوع `i32`، بالتالي يضع الماكرو `println!` القيمة 5 عند تمريرها إلى الدالة مثل قيمة للمعامل `x` في تنسيق السلسلة النصية.

يجب التصريح عن نوع كل معامل في بصمة الدالة، وهذا أمر متعمد في تصميم لغة رست؛ إذ يعني تحديد أنواع المعاملات في تعريف الدالة أن المُصرّف لن يحتاج منك استخدامها في مكان آخر ضمن الشيفرة البرمجية لمعرفة النوع الذي قصدته، وبالتالي يستطيع المصرف إعطاء رسائل خطأ ذات معنًى ومضمون مُساعد أكثر إذا كان يعلم نوع المعاملات التي تأخذها الدالة.

يجب فصل المعاملات بالفاصلة عند تعريف أكثر من معامل واحد كما هو موضح:

اسم الملف: src/main.rs

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {value}{unit_label}");
}
```

يُنشئ هذا المثال دالةً باسم `print_labeled_measurment` بمعاملَين، إذ يسمى المعامل الأول `value` وهو من النوع `i32`، بينما يسمى النوع الثاني `unit_label` وهو من النوع `char`، وتطبع الدالة نصًّا يحتوي على كل من `value` و `unit_label`.

دعنا نجرّب تشغيل الشيفرة البرمجية السابقة، وذلك باستبدال البرنامج الموجود حاليًا في ملف "src/main.rs" لمشروع "function" بالشيفرة البرمجية السابقة، وتشغيل البرنامج باستخدام `cargo run`:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/functions`
The measurement is: 5h
```

نحصل على الخرج السابق طالما استُدعيت الدالة بالقيمة "5" للمعامل `value` والقيمة 'h' للمعامل `unit_label`.

3.3.2 التعابير والتعليمات

يتألف متن الدالة من مجموعة من التعليمات التي تنتهي -اختياريًا- بتعبير expression، والدوال التي غطيناها حتى الآن لم تتضمن تعبيرًا في نهاية التعليمة، إلا أننا قد رأينا تعبيرًا بمثابة جزء من تعليمة. من المهم

أن نميّز بين المصطلحين، وذلك لأن رست لغة مبنية على التعابير وذلك الأمر لا ينطبـق على بقيـة اللغـات، لـذا دعنا ننظر إلى ماهية التعليمات والتعابير وما هو الفرق فيما بينهما وكيف يؤثر كل منهما على متن الدالة.

التعليمات هي توجيهات تُجري عمليات ما ولا تُعيد قيمةً، بينما تُقيّم التعابير إلى قيمة ناتجة. دعنا ننظر إلى بعض الأمثلة.

استخدمنا في الحقيقة سابقًا كلًا من التعابير والتعليمات، وذلك بإنشاء متغير وإسناد قيمـة إليـه باسـتخدام الكلمة المفتاحية `let`، نجد في الشيفرة1 التعليمة `let y = 6;`.

اسم الملف: src/main.rs

```
fn main() {
    let y = 6;
}
```

[الشيفرة 1: تعريف دالة main يحتوي على تعليمة واحدة]

تُعدّ تعاريف الدوال تعليمات أيضًا، فالمثال السابق هو تعليمة واحدة بذات نفسه.

لا تُعيد التعليمات أي قيمة، لذلك لا يُمكنك إسناد تعليمة `let` إلى متغير آخر كما نحاول في المثال التـالي، إذ ستحصل على خطأ:

اسم الملف: src/main.rs

```
fn main() {
    let x = (let y = 6);
}
```

ستحصل على الخطأ التالي عند محاولتك لتشغيل البرنامج السابق:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found statement (`let`)
 --> src/main.rs:2:14
  |
|     let x = (let y = 6);
  |              ^^^^^^^^^
  |
  = note: variable declaration using `let` is a statement
```

```
error[E0658]: `let` expressions in this position are unstable
 --> src/main.rs:2:14
  |
|     let x = (let y = 6);
  |              ^^^^^^^^^
  |
  = note: see issue #53667
<https://github.com/rust-lang/rust/issues/53667> for more information

warning: unnecessary parentheses around assigned value
 --> src/main.rs:2:13
  |
|     let x = (let y = 6);
  |             ^         ^
  |
  = note: `#[warn(unused_parens)]` on by default
help: remove these parentheses
  |
-     let x = (let y = 6);
+     let x = let y = 6;
  |

For more information about this error, try `rustc --explain E0658`.
warning: `functions` (bin "functions") generated 1 warning
error: could not compile `functions` due to 2 previous errors; 1
warning emitted
```

لا تُعيد التعليمة `let y = 6` أي قيمة، لذا لا يوجد هناك أي قيمة لإسنادها إلى `x`، وهذا الأمر مختلفٌ عن باقي لغات البرمجة مثل سي C وروبي Ruby إذ تُعيد عملية الإسناد في هذه اللغات قيمة الإسناد، وبالتالي يمكنك كتابة التعليمة `x = y = 6` بحيث تُسند القيمة 6 إلى كل من `x` و `y` إلا أن هذا الأمر لا ينطبق في رست.

تُقيِّم وتركّب التعابير معظم الشيفرة البرمجية التي ستكتبها في رست، خُذ على سبيل المثال تعبير العملية الحسابية `5 + 6`، التي تُقيّم إلى القيمة 11. يمكن أن تكون التعابير جزءًا من التعليمات، ففي الشيفرة1 تُمثّل 6 في التعليمة `let y = 6;` تعبيرًا يُقيّم إلى القيمة 6. يُعد كل من استدعاء الدالة واستدعاء الماكرو وإنشاء نطاق جديد باستخدام الأقواس المعقوفة تعبيرًا، على سبيل المثال:

اسم الملف: src/main.rs

```
fn main() {
    let y = {
        let x = 3;
        x + 1
    };

    println!("The value of y is: {y}");
}
```

التعبير التالي هو جزءٌ يُقيم إلى القيمة 4:

```
{
    let x = 3;
    x + 1
}
```

تُسند القيمة فيما بعد إلى `y` كجزء من تعليمة `let`، لاحظ أن السطر `x + 1` لا يحتوي على فاصلة منقوطة في نهايته مثل معظم الأسطر التي كتبناها لحد اللحظة، وذلك لأن التعابير لا تحتوي على فاصلة منقوطة في النهاية، وإذا أضفت الفاصلة المنقوطة فسيتحول التعبير إلى تعليمة ولن يكون هناك أي قيمة مُعادة حينها. تذكّر ما سبق بينما نتكلم عن القيم المُعادة من الدوال والتعابير لاحقًا.

3.3.3 الدوال التي تعيد قيمة

يُمكن للدوال أن تُعيد قيمًا إلى الشيفرة البرمجية التي استدعتها، ولا نُسمّي القيم المُعادة هذه إلا أنه يجب التصريح عن نوعها باستخدام السهم `->`. القيمة المُعادة من الدالة في رست هي مرادف لقيمة التعبير الأخير في متن الدالة، ويُمكنك إعادة قيمة مبكرًا من الدالة باستخدام الكلمة المفتاحية `return` وتحديد القيمة بعدها، إلا أن معظم الدوال تُعيد قيمة التعبير الأخير ضمنيًا. إليك مثالًا عن دالة تُعيد قيمة:

اسم الملف: src/main.rs

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();
```

```
    println!("The value of x is: {x}");
}
```

لا يوجد في الدالة `five` أي استدعاءات، أو ماكرو، أو حتى تعليمة `let`، بل فقط الرقم 5، وتلك دالة صالحة في لغة رست. لاحظ أن نوع القيمة المُعادة من الدالة مُحدّد أيضًا بكتابة `-> i32`. يجب أن تحصل على الخرج التالي إذا جرّبت تشغيل الشيفرة البرمجية:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/functions`
The value of x is: 5
```

تُمثّل القيمة 5 في الدالة `five` القيمة المُعادة من الدالة، وهذا السبب في تحديدنا لنوع القيمة المعادة بالنوع `i32`، لكن دعنا ننظر إلى الدالة بتعمُّق أكبر، إذ يوجد جزآن مُهمّان، هما:

أولًا، يوضح السطر `let x = five();` أننا نستخدم القيمة المُعادة من الدالة لإسنادها مثل قيمة أولية للمتغير وبما أن الدالة تُعيد القيمة 5، فهذا الأمر موافق لكتابة السطر البرمجي التالي تمامًا:

```
let x = 5;
```

ثانيًا، لا تحتوي الدالة `five` أي معاملات وتُعرِّف نوع القيمة المعادة، إلا أن متن الدالة يحتوي على القيمة 5 بصورةٍ منفردة دون فاصلة منقوطة وذلك لأنه تعبير نُريد قيمته على أنها قيمة الدالة المُعادة.

دعنا ننظر إلى مثال آخر:

اسم الملف: src/main.rs

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

سيطبع تنفيذ الشيفرة البرمجية السابقة `The value of x is: 6`، إلا أننا سنحصل على خطأ إذا استبدلنا الفاصلة المنقوطة في نهاية السطر `x + 1` مما يُغيّر السطر من تعبير إلى تعليمة.

اسم الملف: src/main.rs

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1;
}
```

تصريف الشيفرة البرمجية السابقة سيتسبب بخطأ كما هو موضح:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error[E0308]: mismatched types
 --> src/main.rs:7:24
  |
| fn plus_one(x: i32) -> i32 {
  |    --------            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return`
expression
|     x + 1;
  |          - help: remove this semicolon

For more information about this error, try `rustc --explain E0308`.
error: could not compile `functions` due to previous error
```

تُشير الرسالة الأساسية إلى أن سبب الخطأ هو بسبب "أنواع غير متوافقة mismatched types". يدل تعريف الدالة `plus_one` على أنها تُعيد قيمةً من النوع `i32` إلا أن التعبير لا يُقيَّم إلى قيمة، وهو الشيء المُعبّر بالقوسين `()` نوع الوحدة unit type، وبالتالي لا يوجد هناك أي قيمة لإعادتها مما يتناقض مع تعريف الدالة ويتسبب بخطأ. توفّر رست في رسالة الخطأ رسالةً لمساعدتك في حل هذه المشكلة إذ تقترح إزالة الفاصلة المنقوطة مما سيحلّ المشكلة بدوره.

3.4 التعليقات Comments

يسعى جميع المبرمجين لجعل شيفرتهم البرمجيـة سـهلة الفهم، إلا أن الشـرح الإضـافي في بعض الأحيـان لازم، وهنا تأتي أهمية التعليقات في الشيفرة المصدرية التي يتجاهلها المُصرّف إلا أنها مفيدة بحقّ للناس الذين يقرؤون شيفرتك المصدرية.

إليك تعليقًا بسيطًا:

```
// hello, world
```

يبدأ التعليـق في لغـة رسـت بشـرطتين مـائلتين ويسـتمر التعليـق إلى نهايـة السـطر، وإذا أردت اسـتخدام التعليق ليشمل عدّة أسطر فعليك استخدام // في كل سطر كما يلي:

```
// So we're doing something complicated here, long enough that we need
// multiple lines of comments to do it! Whew! Hopefully, this comment
will
// explain what's going on.
```

يُمكن إضافة التعليقات في نهاية الأسطر البرمجية:

اسم الملف: src/main.rs

```
fn main() {
    let lucky_number = 7; // I'm feeling lucky today
}
```

إلا أنك غالبًا ما سترى التعليقـات بالتنسـيق التـالي على سـطر منفصـل عن بقيـة الشـيفرة البرمجيـة الـتي تشرحها:

اسم الملف: src/main.rs

```
fn main() {
    // I'm feeling lucky today
    let lucky_number = 7;
}
```

يوجد طريقة أخرى لكتابة التعليقـات ألا وهي التعليقـات التوثيقيـة documentation comments الـتي سنناقشها لاحقًا.

3.5 التحكم بسير التنفيذ Control Flow

تُعد القدرة على تشغيل جزء من الشيفرة البرمجية إذا تحقق شرط ما، أو تشغيل جزء ما باستمرار بينما الشرط محقق من الكتل الأساسية في بناء أي لغة برمجة، كما تُعد تعابير `if` والحلقات التكرارية أكثر اللبنات التي تسمح لك بالتحكم بسير تنفيذ البرنامج flow control في البرامج المكتوبة بلغة رست.

3.5.1 تعابير if الشرطية

يسمح لك تعبير `if` بتفرعة branch شيفرتك البرمجية بحسب الشروط، ويُمكنك كتابة الشرط بحيث "إذا تحقق هذا الشرط فنفذ هذا الجزء من الشيفرة البرمجية، وإلا فلا تنفّذه".

أنشئ مشروعًا جديدًا باسم "branches" في المجلد "projects"، إذ سنستخدم هذا المشروع للتعرف على تعابير `if`. عدّل الملف "src/main.rs" ليحتوي على الشيفرة التالية:

اسم الملف: src/main.rs

```
fn main() {
    let number = 3;

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

تبدأ جميع تعابير `if` بالكلمة المفتاحية `if` متبوعةً بالشرط، ويتحقق الشرط في مثالنا السابق فيما إذا كانت قيمة المتغير `number` أصغر من 5، ونضع شيفرة برمجية مباشرةً بعد الشرط داخل أقواس معقوصة تُنفّذ إذا كان الشرط المذكور محققًا، تُدعى الشيفرة البرمجية المُرتبطة بالشرط في تعابير `if` بالأذرع arms في بعض الأحيان، وهي تشبه أذرع تعابير `match` التي تكلمنا عنها سابقًا.

يُمكننا أيضًا تضمين تعبير `else` اختياريًا وهو ما فعلناه في هذا المثال، وذلك لإعطاء البرنامج كتلة بديلة للتنفيذ إذا كان الشرط في تعليمة `if` السابقة غير مُحقّق. يتخطى البرنامج كتلة تعليمة `if` ببساطة وينفذ بقية البرنامج في حال عدم وجود تعبير `else`.

نحصل على الخرج التالي في حال تجربتنا لتنفيذ الشيفرة البرمجية:

```
$ cargo run
```

```
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
condition was true
```

دعنا نُغيّر قيمة `number` إلى قيمة أخرى تجعل قيمة الشرط `false` ونرى ما الذي سيحدث:

```
let number = 7;
```

نفّذ البرنامج مجددًا، وانظر إلى الخرج:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
condition was false
```

من الجدير بالذكر أيضًا أن الشـرط في هـذه الشـيفرة البرمجيـة يجب أن يكـون من النـوع `bool` وإذا لم يكن كذلك فسنحصل على خطأ، جرّب تنفيذ الشيفرة البرمجية التالية على سبيل المثال:

اسم الملف: src/main.rs

```
fn main() {
    let number = 3;

    if number {
        println!("number was three");
    }
}
```

يُقيَّم شرط `if` إلى القيمة 3 هذه المرة، ويعرض لنا رست الخطأ التالي:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: mismatched types
 --> src/main.rs:4:8
  |
|     if number {
  |        ^^^^^^ expected `bool`, found integer
```

```
For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` due to previous error
```

يُشير الخطأ إلى أن رست توقّع تلقي قيمة من نوع `bool` إلا أنه حصل على قيمة عدد صحيح integer. لا تُحوّل رست الأنواع غير البوليانية إلى أنواع بوليانية بعكس لغات البرمجة الأخرى، مثل روبي وجافا سكريبت، إذ عليك أن تكون دقيقًا بكتابة شرط تعليمة `if` ليكون تعبيرًا يُقيَّم إلى قيمة بوليانية، على سبيل المثال إن أردنا لكتلة تعليمة `if` أن تعمل فقط في حالة كان الرقم لا يساوي الصفر فيمكننا تغيير التعبير كما يلي:

اسم الملف: src/main.rs

```rust
fn main() {
    let number = 3;

    if number != 0 {
        println!("number was something other than zero");
    }
}
```

سيطبع تشغيل الشيفرة البرمجية السابقة "number was something other than zero".

ا. التعامل مع عدة شروط باستخدام else if

يُمكنك استخدام عدة شروط باستخدام `if` و `else` في تعابير `else if`، على سبيل المثال:

اسم الملف: src/main.rs

```rust
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
```

```
}
```

لهـذا البرنـامج أربعـة مسـارات مختلفـة ممكنـة التنفيـذ، ومن المُفـترض أن تحصـل على الخـرج التـالي بعد تشغيله:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
number is divisible by 3
```

يتحقَّق البرنامج عند تنفيذه من كل تعبير `if` فيما إذا كان مُحققًا ويُنفّذ أول متن شرط يُحقّق، لاحظ أنه على الـرغم من قابليـة قسـمة 6 على 2 إلا أننـا لم نـرى الخـرج "number is divisible by 2"، أو الخـرج "number is not divisible by 4, 3, or 2" من كتلة التعليمة `else`، وذلك لأن رست تُنفّذ الكتلة الأولى التي تحقق الشرط فقط وحالما تجد هذه الكتلة، فإنها لا تتفقّد تحقق الشروط الأخرى التي تلي تلك الكتلة.

قد يسبب استخدام الكثير من تعابير `else if` الفوضى في شيفرتك البرمجية، لذا إذا كان لديك أكـثر من تعبير واحد، تأكد من إعـادة النظـر إلى شـيفرتك البرمجيـة ومحاولـة تحسـينها، وسـنناقش لاحقًـا هيكـل تفـرعي branching construct في رست يُدعى `match` وقد صُمّم لهذه الحالات خصيصًا.

ب. استخدام if في تعليمة let

يُمكننا استخدام `if` في الجانب الأيمن من تعليمة `let` بالنظر إلى أنها تعبير وإسناد النتيجة إلى متغـير كمـا توضح الشيفرة 2.

اسم الملف: src/main.rs

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {number}");
}
```

[الشيفرة 2: إسناد نتيجة تعبير if إلى متغير]

يُسند المتغير `number` إلى قيمة بناءً على نتيجة تعبير `if`، نفّذ الشيفرة البرمجية السابقة ولاحظ النتيجة:

```
$ cargo run
```

```
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/branches`
The value of number is: 5
```

تذكر أن كُتل الشيفرات البرمجية تُقيّم إلى قيمة آخر تعبير موجود داخلها والأرقام بحد ذاتها هي تعابير أيضًا، في هذه الحالة تعتمد قيمة تعبير `if` بالكامل على أي كتلة برمجية تُنفَّذ، وهذا يعني أنه يجب أن تكون القيم المُحتمل أن تكون نتيجة تعبير `if` من النوع ذاته.

نجد في الشيفرة 2 نتيجة كل من ذراع `if` و `else`، إذ تُمثّل القيمة النوع الصحيح `i32`. نحصل على خطأ إذا كانت الأنواع غير متوافقة كما هو الحال في المثال التالي:

اسم الملف: src/main.rs

```
fn main() {
    let condition = true;

    let number = if condition { 5 } else { "six" };

    println!("The value of number is: {number}");
}
```

عندما نحاول تصريف الشيفرة البرمجية السابقة سنحصل على خطأ، إذ يوجد لذراعي `if` و `else` قيمتين من أنواع غير متوافقة، ويدلّنا رست على مكان المشكلة في البرنامج بالضبط عن طريق الرسالة:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: `if` and `else` have incompatible types
 --> src/main.rs:4:44
  |
|     let number = if condition { 5 } else { "six" };
  |                                -          ^^^^^ expected integer,
found `&str`
  |                                |
  |                                expected because of this

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` due to previous error
```

يُقيَّم التعبير الموجود في كتلة `if` إلى عدد صحيح، بينما يُقيَّم التعبير الموجـود في كتلـة `else` إلى سلسـلة نصية، وذلك لن يعمل لأنه يجب على المتغيرات أن تكون من النوع ذاته وذلك حتى تعـرف رسـت نـوع المتغـير `number` وقت التصريف بصورةٍ نهائية ومؤكدة، إذ تسمح معرفة نوع `number` للمصرف بـالتحقق من أن النـوع المُستخدم صالح الاستخدام في كل مكان نستخدم فيه المتغير `number`، ولن تكـون رسـت قـادرةً على التحقـق من هذا الأمر إذا كان نوع المتغـير `number` يُحـدّد عنـد وقت التشـغيل runtime فقـط، إذ سيُصـبح المصـرف مُشوَّشًا ولن يُقدم الضمانات ذاتها في الشيفرة البرمجية إذا كان عليه تتبع عدة أنواع افتراضية لأي متغير.

3.5.2 التكرار باستخدام الحلقات

نحتاج غالبًا لتنفيذ جـزء محـدد من الشـيفرة البرمجيـة أكـثر من مـرة واحـدة، ولتحقيـق ذلـك تزوّدنـا رسـت بالحلقات التي تُنفّـذ الشـيفرة البرمجيـة داخـل متن الحلقـة من البدايـة إلى النهايـة ومن ثم إلى البدايـة مجـددًا، وللتعرّف إلى الحلقات دعنا نُنشئ مشروعًا جديدًا باسم "loops".

لرست ثلاثة أنواع من الحلقات، هي: `loop` و `while` و `for`، دعنا نجرّب كل منها.

ا. تكرار الشيفرة البرمجية باستخدام loop

تُعلِم الكلمة المفتاحية `loop` رست بوجوب تنفيذ جـزء من الشـيفرة البرمجيـة على نحـوٍ متكـرر إلى الأبـد أو لحين تحديد التوقف بصورةٍ صريحة.

على سبيل المثال، عدّل محتويات الملـف "src/main.rs" في مجلـد مشـروعنا الجديـد "loops" ليحتـوي على الشيفرة البرمجية التالية:

اسم الملف: src/main.rs

```
fn main() {
    loop {
        println!("again!");
    }
}
```

عندما نُشغّل البرنـامج السـابق سـنجد النص "again!" مطبوعًـا مـرةً بعـد الأخـرى باسـتمرار إلى أن نوقـف البرنامج يـدويًا، ونسـتطيع إيقافـه باسـتخدام اختصـار لوحـة المفـاتيح "ctrl-c"، إذ تـدعم معظم الطرفيـات هـذا الاختصار لإيقاف البرنامج في حال تكرار حلقة للأبد. جرّب الأمر:

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.29s
```

```
     Running `target/debug/loops`
again!
again!
again!
again!
^Cagain!
```

يُمثل الرمز ^C الموضع الذي ضغطت فيه على الاختصار "ctrl-c"، وقد تجد الكلمة again! مطبوعةً بعد ^C أو قد لا تجدها بحسب مكان التنفيذ ضمن الشيفرة البرمجية عند ضغطك على إشارة المقاطعة interrupt signal.

تُزوّدنا رست أيضًا لحسن الحظ بطريقة أخرى للخروج قسريًا من حلقة تكرارية باستخدام شيفرة برمجية، إذ يمكننا استخدام الكلمة المفتاحية break داخل الحلقة التكرارية لإخبار البرنامج بأننا نريد إيقاف تنفيذ الحلقة. تذكر أننا فعلنا ذلك عند كتابتنا شيفرة برنامج لعبة التخمين سابقًا وذلك للخروج من البرنامج عندما يفوز اللاعب بتخمين الرقم الصحيح.

كما أننا استخدمنا أيضًا الكلمة المفتاحية continue في لعبة التخمين، وهي كلمة تُخبر البرنامج بتخطي أي شيفرة برمجية متبقية داخل الحلقة في التكرار الحالي والذهاب إلى التكرار اللاحق.

ب. إعادة قيم من الحلقات

واحدة من استخدامات loop هي إعادة تنفيذ عملية قد تفشل، مثل التحقق إذا أنهى خيط thread ما العمل، وقد تحتاج أيضًا إلى تمرير نتيجة هذه العملية خارج الحلقة إلى باقي الشيفرة البرمجية؛ ولتحقيق ذلك يمكنك إضافة القيمة التي تُريد إعادتها بعد تعبير break، إذ سيتوقف عندها تنفيذ الحلقة وستُعاد القيمة خارج الحلقة حتى يتسنى لك استخدامها كما هو موضح:

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };
```

```
    println!("The result is {result}");
}
```

نُصرّح عن متغير باسم `counter` ونُهيّئه بالقيمة "0" قبل الحلقة التكرارية، ثم نصرح عن متغير باسم `result` لتخزين القيمة المُعادة من الحلقة. نُضيف 1 إلى المتغير `counter` عند كل تكرار للحلقة ومن ثم نتحقق فيما إذا كان المتغير `counter` مساويًا إلى القيمة 10، وعندما يتحقق هذا الشرط نستخدم الكلمة المفتاحية `break` مع القيمة `counter * 2`، ونستخدم بعد الحلقة فاصلة منقوطة لإنهاء التعليمة التي تُسند القيمة إلى `result`، وأخيرًا نطبع القيمة `result` التي تكون في هذه الحالة مساويةً إلى 20.

ج. تسمية الحلقات للتفريق بين عدة حلقات

تُطبّق `break` و `continue` في حال وجود حلقة داخل حلقة على الحلقة الداخلية الموجود بها الكلمة المفتاحية، ويمكنك تحديد تسمية الحلقة loop label اختياريًا عند إنشاء حلقة حتى يُمكنك استخدام `break` أو `continue` مع تحديد تسمية الحلقة بدلًا من تنفيذ عملها على الحلقة الداخلية. يجب أن تبدأ تسمية الحلقة بعلامة تنصيص واحدة، ويوضح المثال التالي استخدام حلقتين متداخلتين:

```
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
```

```
}
```

للحلقة الخارجية التسمية `'counting_up` وستعدّ من 0 إلى 2، بينما تعدّ الحلقة الداخلية عديمة التسمية من 10 إلى 9. لا تُحدد `break` الأولى أي تسمية لذلك ستغادر الحلقة الداخلية فقط، بينما ستغادر تعليمة `break 'counting_up` الحلقة الخارجية، وتطبع الشيفرة البرمجية السابقة ما يلي:

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.58s
     Running `target/debug/loops`
count = 0
remaining = 10
remaining = 9
count = 1
remaining = 10
remaining = 9
count = 2
remaining = 10
End count = 2
```

د. الحلقات الشرطية باستخدام while

سيحتاج البرنامج غالبًا إلى تقييم شرط داخل حلقة، بحيث يستمر تنفيذ الحلقة إذا كان الشرط محققًا وإلا فسيتوقف تنفيذها عن طريق استدعاء `break` وإيقاف الحلقة ومن الممكن تطبيق شيء مماثل باستخدام مزيج من `loop` و `if` و `else` و `break`، ويمكنك تجربة الأمر الآن داخل برنامج إذا أردت ذلك. يُعد هذا النمط شائعًا جدًا وهذا هو السبب وراء وجود بنية مُضمَّنة في رست لهذا الاستخدام تُدعى حلقة `while`. نستخدم في الشيفرة 3 التالية الحلقة `while` لتكرار الحلقة ثلاث مرات بالعدّ تنازليًا في كل مرة وعند الخروج من الحلقة نطبع رسالة ونُنهي البرنامج.

اسم الملف: src/main.rs

```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{number}!");
```

```
        number -= 1;
    }

    println!("LIFTOFF!!!");
}
```

[الشيفرة 3: استخدام حلقة while لتنفيذ شيفرة برمجية عند تحقق شرط ما]

يُغنينا استخدام هذه البنية عناء استخدام الكثير من التداخلات بواسطة `loop` و `if` و `else` و `break` كما أنه أكثر وضوحًا، إذ طالما يكون الشرط محققًا ستُنفّذ الحلقة وإلا فسيغادر الحلقة.

ه. استخدام for مع تجميعة Collection

يمكنك اختيار البنية `while` للانتقال بين عناصر التجميعة مثل المصفوفات. توضح الشيفرة 4 ذلك الاستخدام بطباعة كل عنصر في المصفوفة `a`.

اسم الملف: src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("the value is: {}", a[index]);

        index += 1;
    }
}
```

[الشيفرة 4: الانتقال بين عناصر التجميعة باستخدام حلقة while]

إليك الشيفرة البرمجية التي تنتقل بين عناصر المصفوفة، إذ تبدأ من الدليل "0" وتنتقل إلى ما يليه لحد الوصول إلى الدليل الأخير في المصفوفة (أي عندما يكون `index < 5` غير محقق). سيطبع تنفيذ الشيفرة السابقة عناصر المصفوفة كما يلي:

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32s
     Running `target/debug/loops`
```

```
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

تظهر جميع قيم عناصر المصفوفة الخمسة ضمن الطرفية كما هو متوقع. على الرغم من أن `index` سيصل إلى القيمة 5 في مرحلة ما إلا أن تنفيذ الحلقة يتوقف قبل محاولة طباعة العنصر السادس من المصفوفة.

سلوك البرنامج معرض للخطأ، فقد يهلع البرنامج إذا كانت قيمة الدليل أو الشرط الذي يُفحص خاطئة، على سبيل المثال إذا استبدلنا تعريف المصفوفة `a` ليكون لها أربعة عناصر ولكننا نسينا تحديث الشرط إلى `while index < 4`، ستهلع الشيفرة البرمجية، كما أن هذا السلوك بطيء لأن المصرف يُضيف شيفرة برمجية عند وقت التشغيل لإنجاز التحقق من الشرط فيما إذا كان الدليل خارج حدود المصفوفة عند كل تكرار ضمن الحلقة. بدلًا من ذلك، يمكننا استخدام حلقة `for` وتنفيذ شيفرة برمجية لكل عنصر في التجميعة، وتبدو الحلقة بالشكل الموضح في الشيفرة 5.

اسم الملف: src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("the value is: {element}");
    }
}
```

[الشيفرة 5: الانتقال بين عناصر التجميعة باستخدام حلقة for]

ستجد الخرج ذاته للشيفرة 4 عند تنفيذ الشيفرة السابقة، والأهم هنا أننا زدنا من أمان شيفرتنا البرمجية وأزلنا أي فرص للأخطاء الناجمة عن الذهاب إلى ما بعد حدود المصفوفة، أو عدم الذهاب إلى نهايتها وبالتالي عدم طباعة جميع العناصر.

لست مضطرًا لتغيير أي شيفرة برمجية باستخدام حلقة `for` إذا عدلت رقم العناصر في المصفوفة، الأمر الذي ستضطر لفعله في حال استخدامك للشيفرة 4.

تُستخدم حلقات `for` كثيرًا نظرًا للأمان والإيجاز التي تقدمه مقارنةً ببُنى الحلقات الأخرى الموجودة في رست، حتى أن معظم مبرمجين لغة رست يفضلون استخدام الحلقة `for` عند تنفيذ شيفرة برمجية يُفترض تنفيذها عدد معين من المرات كما هو الحال في مثال العد التنازلي الذي أنجزناه باستخدام حلقة `while` في

الشيفرة 3، ويُنجز ذلك الأمر باستخدام Range المُضمَّن في المكتبة القياسية، والذي يولّد بدوره جميع الأرقام في السلسلة بدءًا من رقم معين وانتهاءً برقم آخر.

إليك ما سيبدو عليه برنامج العد التنازلي باستخدام حلقة for وتابع آخر لم نتكلم عنه بعد وهو rev، المُستخدم في عكس المجال:

اسم الملف: src/main.rs

```
fn main() {
    for number in (1..4).rev() {
        println!("{number}!");
    }
    println!("LIFTOFF!!!");
}
```

تبدو هذه الشيفرة البرمجية أفضل، أليس كذلك؟

3.6 خاتمة

تهانينا، فقد تعلمت في هذا الفصل عن المتغيرات وأنواع البيانات المُفردة والمركبة والدوال والتعليقات وتعابير if والحلقات. إذا أردت التمرّن على المفاهيم الواردة في هذا الفصل، حاول بناء البرامج التالية:

- برنامج يحول درجات الحرارة من وإلى فهرنهايت وكالفن.
- توليد الرقم ذو الموضع n من سلسلة فيبوناتشي.
- طباعة كلمات أنشودة موطني.

عندما تستعدّ للمضي قدمًا إلى الفصل التالي، سنتكلم عن مفهوم في رست غير شائع في بقية لغات البرمجة ألا وهو الملكية ownership.

دورة إدارة تطوير المنتجات

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

4. الملكية Ownership

تُعد الملكية ownership واحدةً من مزايا رست الفريدة ولهذه الميزة تأثيرٌ كبير على كيفية عمل اللغة ككُـل، إذ أنها تُمكن رست من ضمان أمان الذاكرة دون الحاجة إلى كانس مهملات garbage collector، لذا من المهم فهم كيفية عمل الملكية. نستعرض في هذا الفصل مفهوم الملكية، إضافةً إلى العديد من المزايا المرتبطـة مثـل الاستعارة borrowing والشرائح slices وكيف تُخزن رست البيانات في الذاكرة.

4.1 ما هي الملكية ownership؟

المُلكية ownership هي مجموعة من القـوانين الـتي تحـدد كيـف يُـدير برنـامج رسـت اسـتخدام الـذاكرة، ويتوجب على جميع البرامج أن تُدير الطريقة التي تستخدم فيها ذاكرة الحاسوب عند تشغيلها. تلجـأ بعض لغـات البرمجة إلى كانس المهملات garbage collector الذي يتفقد باستمرار الذاكرة غير المُستخدمة بعد الآن أثنـاء عمل البرنامج، بينما تعطي لغات البرمجة الأخرى مسؤولية تحديد الذاكرة وتحريرها للمبرمج مباشرةً، إلا أن رست تسلك طريقًا آخر ثالثًا؛ ألا وهو أن الذاكرة تُدار عبر نظام ملكية يحتوي على مجموعـة من القـوانين الـتي يتفقـدها المصرّف، إذ لا يُصرّف البرنامج إذا حدث خرق لأحد هذه القوانين، ولن تبطئ أي من مزايا الملكية برنامجـك عنـد تشغيله.

سيتطلب مفهوم الملكية بعض الوقت للاعتياد عليه بالنظر إلى أنه مفهوم جديد للعديـد من المـبرمجين، إلا أنك ستجد قوانين نظام الملكية أكثر سهولة بالممارسة وستجدها من البديهيات التي تسـمح لـك بكتابـة شـيفرة برمجية آمنة وفعّالة، لذا لا تستسلم.

ستحصل على أساس قوي في فهم المزايا التي تجعل من رست لغة فريدة فـور فهمـك للملكيـة، وسـتتعلم في هذا الفصل مفهوم الملكية بكتابة بعض الأمثلة التي تركز على هيكل بيانات data structure شائع جدًا هو السلاسل النصية strings.

المكدس Stack والكومة Heap

لا تطلب معظم لغات البرمجة منك بالتفكير بالمكدس stack والكومة heap بصورةٍ متكررة عادةً، إلا أن هذا الأمر مهم في لغات برمجة النظم مثل رست، إذ يؤثر وجود القيمة في المكدس أو الكومة على سلوك اللغة واتخاذها لبعض القرارات المعينة، وسنصف أجزاءً من نظام الملكية بما يتعلق بالكومة والمكدس لاحقًا، وسنستعرض هنا مفهومي -الكومة والمكدس- ونشرحهما للتحضير لذلك.

يمثل كلًا من المكدس والكومة أجزاءً متاحةً من الذاكرة لشيفرتك البرمجية حتى تستخدمها عند وقت التشغيل runtime، إلا أنهما مُهيكلان بطريقة مختلفة؛ إذ يُخزن المكدس القيم بالترتيب الذي وردت فيه ويُزيل القيم بالترتيب المعاكس، ويُشار إلى ذلك بمصطلح **الداخل آخرًا، يخرج أولًا last in, first out**. يمكنك التفكير بالمكدس وكأنه كومة من الأطباق، فعندما تضيف المزيد من الأطباق، فإنك تضيفها على قمة الكومة وعندما تريد إزالة طبق فعليك إزالة واحد من القمة، ولا يُمكنك إزالة الأطباق من المنتصف أو من القاع. تُسمى عملية إضافة البيانات بالدفع إلى المكدس pushing onto the stack بينما تُدعى عملية إزالة البيانات بالإزالة من المكدس popping off the stack، ويجب على جميع البيانات المخزنة في المكدس أن تكون من حجم معروف وثابت، بينما تُخزّن البيانات ذات الحجم غير المعروف عند وقت التصريف أو ذات الحجم الممكن أن يتغير في الكومة بدلًا من ذلك.

الكومة هي الأقل تنظيمًا إذ يمكنك طلب مقدار معين من المساحة عند تخزين البيانات إليها، وعندها يجد محدد المساحة memory allocator حيزًا فارغًا في الكومة يتسع للحجم المطلوب، يعلّمه على أنه حيّز قيد الاستعمال، ثم يُعيد مؤشرًا pointer يشير إلى عنوان المساحة المحجوزة، وتدعى هذه العملية بحجز مساحة الكومة allocating on the heap وتُدعى في بعض الأحيان بحجز المساحة فقط (لا تُعد عملية إضافة البيانات إلى المكدس عملية حجز مساحة). بما أن المؤشر الذي يشير إلى الكومة من حجم معروف وثابت، يمكنك تخزينه في المكدس، وعندما تريد البيانات الفعلية من الكومة يجب عليك تتبع المؤشر. فكر بالأمر وكأنه أشبه بالجلوس في مطعم، إذ تصرّح عن عدد الأشخاص في مجموعتك عند دخولك إلى المطعم، ثم يبحث كادر المطعم عن طاولة تتسع للجميع ويقودك إليها، وإذا تأخر شخص ما عن المجموعة يمكنه سؤال كادر المطعم مجددًا ليقوده إلى الطاولة.

الدفع إلى المكدس أسرع من حجز الذاكرة في الكومة، لأن محدد المساحة لا يبحث عن مكان للبيانات الجديدة وموقع البيانات المخزنة، فهو دائمًا على قمة المكدس، بالمثل، يتطلب حجز المساحة في الكومة مزيدًا من العمل، إذ يجب على محدد المساحة البحث عن حيز كبير بما فيه الكفاية ليتسع البيانات ومن ثم حجزها للتحضير لعملية حجز المساحة التالية.

الوصول إلى البيانات من الكومة أبطأ من الوصول إلى البيانات من المكدس وذلك لأنه عليك أن تتبع المؤشر لتصل إلى حيز الذاكرة، وتؤدي المعالجات المعاصرة عملها بصورةٍ أسرع إذا انتقلت من مكان إلى آخر ضمن الذاكرة بتواتر أقل. لنبقي على تقليد التشابيه، فكر بالأمر وكأن النادل في المطعم يأخذ الطلبات من العديد من الطاولات، وفي هذه الحالة فمن الأفضل أن يحصل النادل على جميع الطلبات في الطاولة الواحدة قبل أن ينتقل إلى الطاولة التي تليها، فأخذ الطلب من الطاولة (أ) ومن ثم أخذ الطلب من الطاولة (ب) ومن ثم العودة

إلى الطاولة (أ) والطاولة (ب) مجددًا عملية أبطأ بكثير، وبالمثل فإن المعالج يستطيع إنجاز عمله بصورةٍ أفضل إذا تعامل مع البيانات القريبة من البيانات الأخرى (كما هو الحال في المكدس) بـدلًا من العمـل على بيانـات بعيـدة عن بعضها (مثل الكومة).

عندما تستدعي شيفرتك البرمجية دالةً ما، يُدفع بالقيم المُمررة إليها إلى المكدس (بما فيهـا المؤشـرات إلى البيانات الموجودة في الكومة) إضافةً إلى متغيرات الدالة المحلية، وعندما ينتهي تنفيذ الدالة، تُزال هذه القيم من المكدس.

يتكفل نظام الملكية بتتبع الأجزاء التي تستخدم البيانات من الكومة ضمن شيفرتك البرمجية، وتقليـل كميـة البيانات المُكررة ضمن الكومة، وإزالة أي بيانات غير مُستخدمة منها حتى لا تنفد من المساحة. عندما تفهم نظـام الملكية لن تحتاج للتفكير بالمكدس والكومة كثيرًا، فكل ما عليك معرفته هو أن الهدف من نظام الملكية هـو إدارة بيانات الكومة، وسيساعدك هذا الأمر في فهم طريقة عمل هذا النظام.

4.1.1 قوانين الملكية

دعنا نبدأ أولًا بالنظر إلى قوانين الملكية، أبقِ هذه القوانين في ذهنك بينما تقرأ بقية الفصل الذي يستعرض أمثلةً توضح هذه القوانين:

- لكل قيمة في رست مالك owner.
- يجب أن يكون لكل قيمة مالك واحد في نقطة معينة من الوقت.
- تُسقط القيمة عندما يخرج المالك من النطاق scope.

4.1.2 نطاق المتغير

الآن وبعد تعرفنا إلى مبادئ رست في الفصول السابقة، لن نُضمّن الشيفرة `fn main() {` في الأمثلـة، لذا إذا كنت تتبع الأمثلة تأكد من أنك تكتب الشيفرة البرمجية داخـل دالـة `main` يـدويًا، ونتيجـةً لـذلك سـتكون أمثلتنا أقصر بعض الشيء مما سيسمح لنا بالتركيز على التفاصيل المهمة بـدلًا من الشـيفرة البرمجيـة النمطيـة المتكررة.

سننظر إلى نطاق المتغيرات في أول مثال من أمثلة الملكية، والنطاق هو مجال ضـمن البرنـامج يكـون فيـه العنصر صالحًا. ألقِ نظرةً على المتغير التالي:

```
let s = "hello";
```

يُشير المتغير `s` إلى السلسلة النصية المجردة، إذ أن قيمة السلسلة النصية مكتوبة بصورةٍ صـريحة على أنهـا نص في برنامجنا، والمتغير هذا صالح من نقطة التصريح عنه إلى نهاية النطاق الحالي. توضح الشيفرة 1 البرنامج مع تعليقات توضح مكان صلاحية المتغير `s`.

```
{                      // المتغير غير صالح هنا إذ لم يُصرّح عنه بعد
    let s = "hello";   // المتغير صالح من هذه النقطة فصاعدًا

    // يمكننا استخدام s في العمليات هنا
}                      // انتهى النطاق بحلول هذه النقطة ولا يمكننا استخدام s
```

[الشيفرة 1: متغير والنطاق الذي يكون فيه صالحًا]

بكلمات أخرى، هناك نقطتان مهمتان حاليًا:

- عندما يصبح المتغير `s` ضمن النطاق، يصبح صالحًا.
- يبقى المتغير صالحًا حتى مغادرته النطاق.

لحد اللحظة، العلاقة بين النطاقات والمتغيرات هي علاقة مشابهة للعلاقة التي تجدها في لغات البرمجة الأخرى، وسنبني على أساس هذا الفهم النوع `String`.

4.1.3 النوع String

نحتاج نوعًا أكثر تعقيدًا من الأنواع التي غطيناها سابقًا وذلك لتوضيح قوانين الملكية، إذ كانت الأنواع السابقة جميعها من أحجام معروفة ويمكن تخزينها في المكدس وإزالتها عند انتهاء نطاقها، كما أنه من الممكن نسخها بكل سهولة إلى متغير آخر جديد يمثّل نسخةً مستقلةً وذلك إذا احتاج جزءٌ ما من الشيفرة البرمجية استخدام المتغير ذاته ضمن نطاق آخر، إلا أننا بحاجة إلى النظر لأنواع البيانات المخزنة في الكومة حتى نكون قادرين على معرفة ما تفعله رست لتنظيف البيانات هذه ويمثل النوع `String` مثالًا رائعًا لهذا الاستخدام.

سنركز على أجزاء النوع `String` التي ترتبط مباشرةً بالملكية، وتنطبق هذه الجوانب أيضًا على أنواع البيانات المعقدة الأخرى سواءٌ كانت هذه الأنواع موجودةً في المكتبة القياسية أو كانت مبنيةً من قبلك، وسنناقش النوع `String` بتعمّق أكبر لاحقًا.

رأينا مسبقًا السلاسل النصية المجردة (وهي السلاسل النصية المكتوبة بين علامتي تنصيص " " بصورةٍ صريحة)، إذ تُكتب قيمة السلسلة النصية يدويًا إلى البرنامج. السلاسل النصية المجردة مفيدةٌ إلا أنها غير مناسبة لكل الحالات، مثل تلك التي نريد فيها استخدام النص ويعود السبب في ذلك إلى أنها غير قابلة للتعديل immutable، والسبب الآخر هو أنه لا يمكننا معرفة قيمة كل سلسلة نصية عندما نكتب شيفرتنا البرمجية، فعلى سبيل المثال، ماذا لو أردنا أخذ الدخل من المستخدم وتخزينه؟ تملك رست لمثل هذه الحالات نوع سلسلة نصية آخر يدعى `String`، ويدير هذا النوع البيانات باستخدام الكومة، وبالتالي يمكنه تخزين كمية غير معروفة من النص عند وقت التصريف. يُمكنك إنشاء `String` من سلسلة نصية مجردة باستخدام دالة `from` كما يلي:

```
let s = String::from("hello");
```

يسمح لنا عامل النقطتين المزدوجتين :: بتسمية الدالة الجزئية from ضمن فضاء الأسماء namespace وأن تندرج تحت النوع String بدلًا من استخدام اسم مشابه مثل string_from، وسنناقش هذه الطريقة في الكتابة أكثر لاحقًا، بالإضافة للتكلم عن فضاءات الأسماء وإنشائها.

يُمكن تعديل mutate هذا النوع من السلاسل النصية:

```
let mut s = String::from("hello");

s.push_str(", world!"); // يُضيف push_str() سلسلةً نصية مجردة إلى النوع String

println!("{}", s); // سيطبع هذا السطر `hello, world!`
```

إذًا، ما الفارق هنا؟ كيف يمكننا تعديل النوع String بينما لا يمكننا تعديل السلاسل النصية المجردة؟ الفارق هنا هو بكيفية تعامل كل من النوعين مع الذاكرة.

4.1.4 الذاكرة وحجزها

نعرف محتويات السلسلة النصية في حال كانت السلسلة النصية مجرّدة عند وقت التصريف، وذلك لأن النص مكتوب في الشيفرة البرمجية بصورةٍ صريحة في الملف النهائي التنفيذي، وهذا السبب في كون السلاسل النصية المجردة سريعة وفعالة، إلا أن هذه الخصائص تأتي من حقيقة أن السلاسل النصية المجردة غير قابلة للتعديل immutable، ولا يمكننا لسوء الحظ أن نضع جزءًا من الذاكرة في الملف التنفيذي الثنائي لكل قطعة من النص، وذلك إذا كان النص ذو حجم غير معلوم عند وقت التصريف كما أن حجمه قد يتغير خلال عمل البرنامج.

نحتاج إلى تحديد مساحة من الذاكرة ضمن الكومة عند استخدام نوع String وذلك لدعم إمكانية تعديله وجعله سلسلةً نصيةً قابلة للزيادة والنقصان، بحيث تكون هذه المساحة التي تخزن البيانات غير معلومة الحجم عند وقت التصريف، وهذا يعني:

- يجب أن تُطلب الذاكرة من مُحدد الذاكرة عند وقت التشغيل.
- نحتاج طريقة لإعادة الذاكرة إلى محدد الذاكرة عندما ننتهي من استخدام String الخاص بنا.

يُنجز المتطلّب الأول عن طريق استدعاء String::from، إذ تُطلب الذاكرة التي يحتاجها ضمنيًا، وهذا الأمر موجود في معظم لغات البرمجة.

أما المتطلب الثاني فهو مختلفٌ بعض الشيء، إذ يراقب كانس المهملات -أو اختصارًا GC- الذاكرة غير المُستخدمة بعد الآن ويحررها، ولا حاجة للمبرمج بالتفكير بهذا الأمر، بينما تكون مسؤوليتنا في اللغات التي لا

تحتوي على كانس المهملات هي العثور على المساحة غير المُستخدمة بعد الآن وأن نستدعي الشيفرة البرمجيـة بصورةٍ صريحة لتحرير تلك المساحة، كما هو الحال عندما استدعينا شيفرة برمجية لحجزها، ولطالمـا كـانت هـذه المهمة صعبة على المبرمجين، فإذا نسينا تحرير الذاكرة فنحن نهدر الذاكرة وإذا حررنا الذاكرة مبكرًا فهذا يعـني أن قيمة المتغير أصبحت غير صالحة للاستخدام، بينما نحصل على خطأ إذا حررنا الـذاكرة نفسـها لأكـثر من مـرة، إذ علينا استخدام تعليمة `allocate` واحدة فقط مصحوبةً مع تعليمة `free` واحدة لكل حيز ذاكرة نستخدمه.

تسلك لغة رست سلوكًا مختلفًا، إذ تُحرر الذاكرة أوتوماتيكيًا عندما يغـادر المتغـير الـذي يملـك تلـك الـذاكرة النطاق. إليك إصدارًا من الشيفرة 1 نستخدم فيه النوع `String` بدلًا من السلسلة النصية المجردة:

```
{
    let s = String::from("hello"); // المتغير s صالح من هذه النقطة فصاعدًا

    // يمكننا إنجاز العمليات باستخدام المتغير s هنا
}                                  // انتهى النطاق ولم يعد المتغير s صالحًا
```

نستعيد الذاكرة التي يستخدمها `String` من محدد الذاكرة عندما يخرج المتحول `s` من النطاق، إذ تستدعي رست دالةً مميزةً بالنيابة عنا عند خروج متحول ما من النطـاق وهـذه الدالـة هي `drop`، وتُسـتدعى تلقائيًـا عنـد الوصول إلى قوس الإغلاق المعقوص `}`.

> يُدعى نمط تحرير الموارد في نهاية دورة حياة العنصر في لغة C++ أحيانًا "اكتساب الموارد هو تهيئتها "Resource Acquisition Is Initialization" -أو اختصارًا RAII- ودالة `drop` في رست هي مشابهة لأنماط RAII التي قد استخدمتها سابقًا.

لهذا النمط تأثير كبير في طريقة كتابة شيفرة رست البرمجية، وقد يبدو بسيطًا للوقت الحـالي إلا أن سـلوك الشيفرة البرمجية قد يكون غير متوقعًا في الحالات الأكثر تعقيدًا عندما يوجـد عـدة متغـيرات تسـتخدم البيانـات المحجوزة على الكومة، دعنا ننظر إلى بعض من هذه الحالات الآن.

١. طرق التفاعل مع البيانات والمتغيرات: النقل

يُمكن لعدة متغيرات أن تتفاعل مع نفس البيانات بطرق مختلفة في رست، دعنا ننظـر إلى الشـيفرة 2 على أنها مثال يستخدم عددًا صحيحًا.

```
let x = 5;
let y = x;
```

[الشيفرة 2: إسناد قيمة العدد الصحيح إلى المتغيرين x و y]

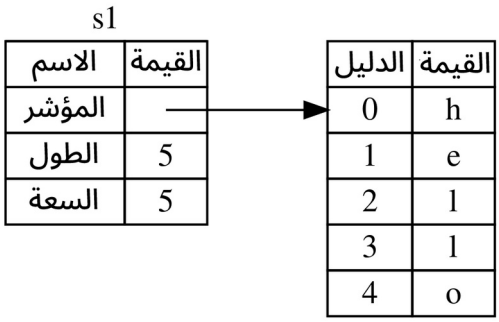
يمكنك غالبًا تخمين ما الذي تؤديه الشيفرة البرمجية السابقة: إسناد القيمة 5 إلى x ومن ثم نسخ القيمة x وإسنادها إلى القيمة y، وبالتالي لدينا متغيرين x و y وقيمة كل منهما تساوي إلى 5، وهذا ما يحدث فعلًا، لأن الأعداد الصحيحة هي قيم بسيطة بحجم معروف وثابت وبالتالي يُمكن إضافة القيمتين 5 إلى المكدس.

لننظر الآن إلى إصدار String من الشيفرة السابقة:

```
let s1 = String::from("hello");
let s2 = s1;
```

تبدو الشيفرة البرمجية هذه شبيهة بسابقتها، وقد نفترض هنا أنها تعمل بالطريقة ذاتها، ألا وهي: ينسخ السطر الثاني القيمة المخزنة في المتغير s1 ويُسندها إلى s2 إلا أن هذا الأمر غير صحيح.

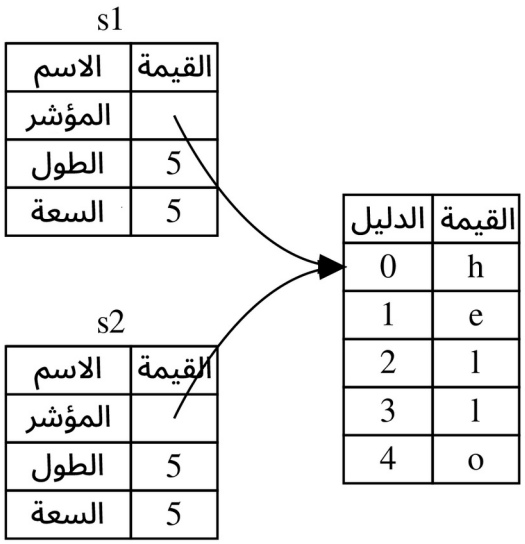
انظر إلى الشكل 1 لرؤية ما الذي يحصل بدقة للنوع String، إذ يتكون هذا النوع من ثلاثة أجزاء موضحة ضمن الجدول اليساري وهي المؤشر ptr الذي يشير إلى الذاكرة التي تُخزن السلسلة النصية وطول السلسلة النصية len وسعتها capacity، وتُخزّن مجموعة المعلومات هذه في المكدس، بينما يمثّل الجدول اليميني الذاكرة في الكومة التي تخزن محتوى السلسلة النصية.



الشكل 1: مخطط توضيحي لما تبدو عليه الذاكرة عند استخدام String يخزن القيمة "hello" المُسندة إلى s1

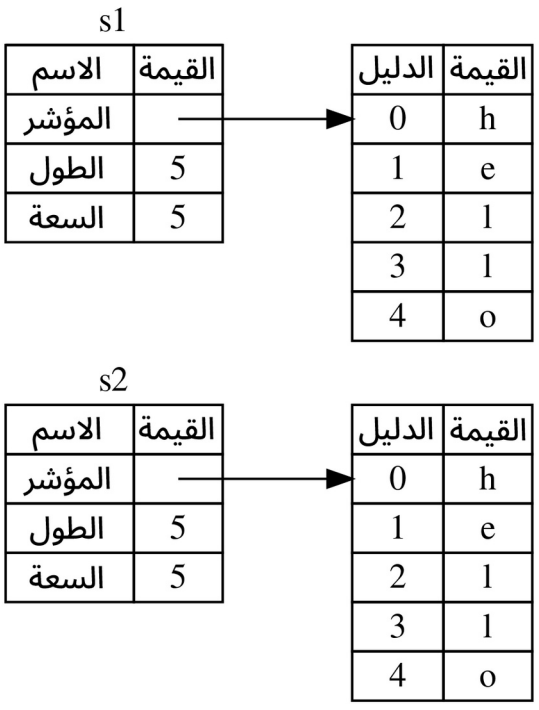
يدل الطول على كمية الذاكرة المُستهلكة بالبايت وهي الحيز الذي يشغله محتوى String، بينما تدل السعة على كمية الذاكرة المستهلكة بالكامل التي تلقّاها String من مُحدد الذاكرة، والفرق بين الطول والسعة مهم، إلا أننا سنهمل السعة لأنها غير مهمة في السياق الحالي.

تُنسخ بيانات String عندما نُسند s1 إلى s2، وهذا يعني أننا ننسخ المؤشر والطول والسعة الموجودين في المكدس ولا ننسخ البيانات الموجودة في الكومة التي يشير إليها المؤشر، بكلمات أخرى، يبدو تمثيل الذاكرة بعد النسخ كما هو موضح في الشكل 2.



الشكل 2: تمثيل الذاكرة للمتغير s2 الذي يحتوي على نسخة من مؤشر وطول وسعة s1

تمثيل الذاكرة **غير مطابق** للشكل 3 وقد ينطبق هذا التمثيل إذا نسخت رست محتويات بيانات الكومة أيضًا، وإذا فعلت رست ذلك، فستكون عملية الإسناد `s2 = s1` عمليةً مكلفةً وستؤثر سلبًا على أداء وقت التشغيل إذا كانت البيانات الموجودة في الكومة كبيرة.



الشكل 3: احتمال آخر لما قد تبدو عليه الذاكرة بعد عملية الإسناد s2 = s1 وذلك إذا نسخت رست محتويات الكومة أيضًا

قلنا سابقًا أن رست تستدعي الدالة `drop` تلقائيًا عندما يغادر متغيرٌ ما النطاق، وتحرّر الذاكرة الموجودة في الكومة لذلك المتغير، إلا أن الشكل 2 يوضح أن كلا المؤشرين يشيران إلى الموقع ذاته، ويمثّل هذا مشكلةً

واضحة، إذ عندما يغادر كلًا من s1 وs2 النطاق، فهذا يعني أن الذاكرة في الكومة ستُحرّر مرتين، وهذا خطأ تحرير ذاكرة مزدوج double free error شائع، وهو خطأ من أخطاء أمان الذاكرة الذي ذكرناه سابقًا، إذ يؤدي تحرير الذاكرة نفسها مرتين إلى فساد في الذاكرة مما قد يسبب ثغرات أمنية.

تنظر رست إلى s1 بكونه غير صالح بعد السطر let s2 = s1 وذلك لضمان أمان الذاكرة، وبالتالي لا يتوجب على رست تحرير أي شيء عندما يغادر المتحول s1 النطاق. انظر ما الذي يحدث عندما نحاول استخدام s1 بعد إنشاء s2 (لن تعمل الشيفرة البرمجية):

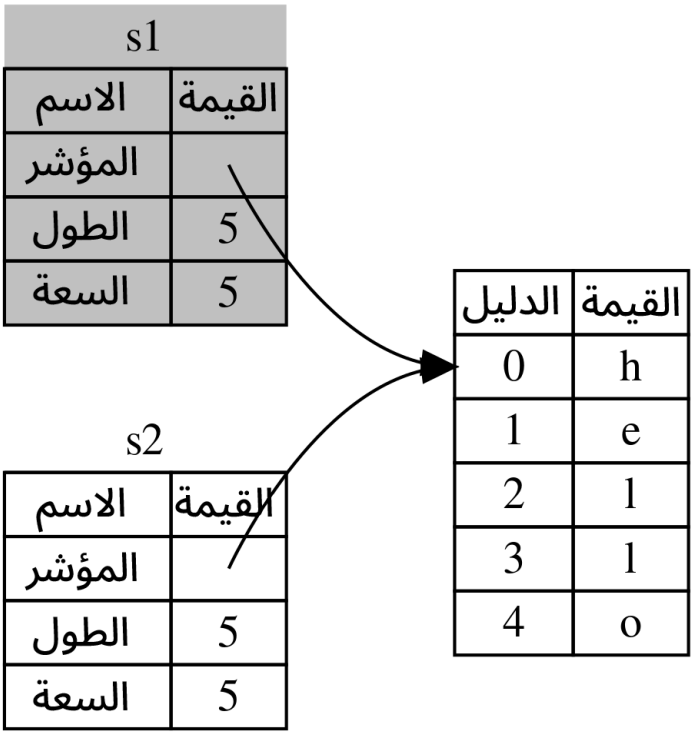
```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

سنحصل على خطأ شبيه بالخطأ التالي لأن رست يمنعك من استخدام المرجع غير الصالح بعد الآن:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:28
  |
|     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does
not implement the `Copy` trait
|     let s2 = s1;
  |              -- value moved here
|
|     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl`
(in Nightly builds, run with -Z macro-backtrace for more info)

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership` due to previous error
```

لعلك سمعت بمصطلح النسخ السطحي shallow copy والنسخ العميق deep copy خلال عملك على لغة برمجة أخرى؛ إذ يُشير مصطلح النسخ السطحي إلى عملية نسخ مؤشر وطول وسعة السلسلة النصية دون

البيانات الموجودة في الكومة، إلا أن رست تسمّي هذه العملية بالنقل move لأنها تُزيل صلاحية المتغـير الأول.
في هذا المثال، نقول أن s1 نُقِلَ إلى s2، والنتيجة الحاصلة موضحة في الشكل 4.



الشكل 4: تمثيل الذاكرة بعد إزالة صلاحية المتغير s1

يحلّ هذا الأمر مشكلتنا، وذلك بجعل المتغير s2 صالحًا فقط، وعند مغادرته للنطاق فإن المساحة تُحرر بنـاءً عليه فقط.

إضافةً لما سبق، هناك خيارٌ تصميمي ملمّح إليـه بواسـطة هـذا الحـل، ألا وهـو أن رسـت لن تُنشـئ نُسَـخًا عميقة من بياناتك تلقائيًا، وبالتالي لن يكون أي نسخ تلقائي مكلفًا بالنسبة لأداء وقت التشغيل.

ب. طرق التفاعل مع البيانات والمتغيرات: الاستنساخ

يُمكننا استخدام تابع شائع يـدعى clone إذا أردنـا نسـخ البيانـات الموجـودة في الكومـة الـتي تعـود للنـوع String نسخًا عميقًا إضافةً لبيانات المكدس، وسنناقش طريقـة كتابـة التوابـع لاحقًـا، إلا أنـك غالبًـا مـا رأيت استخدامًا للتوابع مسبقًا بالنظر إلى أنها شائعة في العديد من لغات البرمجة.

إليك مثالًا عمليًا عن تابع clone:

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

تعمل الشيفرة البرمجية بنجاح، وتولد النتيجة الموضحة في الشكل 3، إذ تُنسخ محتويات الكومة.

عند رؤيتك لاستدعاء التابع `clone`، عليك أن تتوقع تنفيذ شيفرة برمجية إضافية وأن الشيفرة البرمجية ستكون مكلفة التنفيذ، وأن استخدام التابع هو دلالة بصرية أيضًا على حدوث شيء مختلف.

ج. نسخ بيانات المكدس فقط

هناك تفصيلٌ آخر لم نتكلم بخصوصه بعد. تستخدم الشيفرة البرمجية التالية (جزء من الشيفرة 2) أعدادًا صحيحة، وهي شيفرة برمجية صالحة:

```
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```

إلا أن الشيفرة البرمجية تبدو مناقضة لما تعلمنا مسبقًا، إذ أن `x` صالح ولم يُنقل إلى المتغير `y` على الرغم من عدم استدعائنا للتابع `clone`.

السبب في هذا هو أن الأنواع المشابهة للأعداد الصحيحة المؤلفة من حجم معروف عند وقت التصريف تُخزن كاملًا في المكدس، ولذلك يمكننا نسخها فعليًا بصورةٍ أسرع، ولا فائدة في منع `x` من أن يكون صالحًا في هذه الحالة بعد إنشاء المتغير `y`، وبكلمات أخرى ليس هناك أي فرق بين النسخ السطحي والعميق هنا، لذا لن يُغيّر استدعاء التابع `clone` أي شيء مقارنةً بالنسخ السطحي الاعتيادي، ولذلك يمكننا الاستغناء عنه.

لدى لغة رست طريقةً مميزةً تدعى سمة trait النسخ `Copy`، التي تمكّننا من وضعها على الأنواع المخزنة في المكدس كما هو الحال في الأعداد الصحيحة (سنتكلم بالتفصيل عن السمات لاحقًا). إذا استخدم نوعٌ ما السمة `Copy`، فهذا يعني أن جميع المتغيرات التي تستخدم هذا النوع لن تُنقل وستُنسخ بدلًا من ذلك مما يجعل منها صالحة حتى بعد إسنادها إلى متغير آخر.

لن تسمح لنا رست بتطبيق السمة `Copy` إذا كان النوع -أو أي من أجزاء النوع- يحتوي على السمة `Drop`، إذ أننا سنحصل على خطأ وقت التصريف إذا كان النوع بحاجة لشيء مميز للحدوث عند خروج القيمة من النطاق وأضفنا السمة `Copy` إلى ذلك النوع، إن أردت تعلم المزيد عن إضافة السمة `Copy` إلى النوع لتطبيقها، ألقِ نظرةً على الملحق (ت) قسم السمات المُشتقة derivable traits.

إذًا، ما هي الأنواع التي تقبل تطبيق السمة `Copy`؟ يمكنك النظر إلى توثيق النوع للتأكد من ذلك، لكن تنص القاعدة العامة على أن أي مجموعةٍ من القيم البسيطة المُفردة تقبل السمة `Copy`، إضافةً إلى أي شيء لا يتطلب تحديد الذاكرة، أو ليس أي نوع من أنواع الموارد. إليك بعض الأنواع التي تقبل تطبيق `Copy`:

- كل أنواع الأعداد الصحيحة مثل `u32`.

- الأنواع البوليانية `bool` التي تحمل القيمتين `true` و `false`.
- جميع أنواع أعداد الفاصلة العشرية مثل `f64`.
- نوع المحرف `char`.
- الصفوف tuples إذا احتوى الصف فقط على الأنواع التي يمكن تطبيق `Copy` عليها، على سبيل المثال يُمكن تطبيق `Copy` على `(i32, i32)`، بينما لا يمكن تطبيق `Copy` على `(i32, String)`.

4.1.5 الملكية والدوال

تشبه طريقة تمرير قيمة إلى دالة إسناد assign قيمة إلى متغير، إذ أن تمرير القيمة إلى المتغير سـينقلها أو ينسخها كما هو الحال عند إسناد القيمة، توضح الشيفرة 3 مثالًا عن بعض الطرق التي توضح أين يخـرج المتغـير من النطاق.

اسم الملف: src/main.rs

```
fn main() {
    let s = String::from("hello");  // يدخل المتغير s إلى النطاق

    takes_ownership(s);             // تُنقل قيمة s إلى الدالة ولا تعود صالحة للاستخدام هنا

    let x = 5;                      // يدخل المتغير x إلى النطاق

    makes_copy(x);
// يُنقل المتغير x إلى الدالة إلا أن i32 تملك السمة Copy لذا من الممكن استخدام المتغير x بعد هذه النقطة

} // نُقِلَت s خارج النطاق ولا شيء مميز يحدث لأن قيمة x يغادر المتحول

fn takes_ownership(some_string: String) { // يدخل المتغير some_string إلى النطاق
    println!("{}", some_string);
} // يُغادر المتغير some_string النطاق هنا وتُستدعى drop، وتُحرر الذاكرة الخاصة بالمتغير

fn makes_copy(some_integer: i32) { // يدخل المتغير some_integer إلى النطاق
    println!("{}", some_integer);
} // يغادر المتغير some_integer النطاق ولا يحصل أي شيء مثير للاهتمام
```

[الشيفرة 3: استخدام الدوال مع الملكية والنطاق]

ستعرض لنا رست خطأً عند وقت التصريف إذا حاولنا استخدام a بعد استدعاء takes_ownership، ويحمينا هذا التفقد الساكن static من بعض الأخطاء. حاول إضافة شيفرة برمجية تستخدم s و x إلى الدالة main ولاحظ أين يمكنك استخدامهما وأين تمنعك قوانين الملكية من استخدامهما.

4.1.6 القيم المعادة والنطاق

يُمكن أن تحول عملية إعادة القيمة ملكيتها أيضًا، توضح الشيفرة 4 مثالًا عن دالة تُعيد قيمة بصورةٍ مشابهة للشيفرة 3.

اسم الملف: src/main.rs

```
fn main() {
    // تنقل الدالة gives_ownership قيمتها المُعادة إلى s1
    let s1 = gives_ownership();

    let s2 = String::from("hello");     // يدخل المتغير s2 إلى النطاق

    let s3 = takes_and_gives_back(s2);
// يُنقل المتغير s2 إلى takes_and_gives_back الذي ينقل قيمته المعادة بدوره إلى المتغير s3

}
//  يُغادر s3 النطاق من هنا ويُحرر من الذاكرة باستخدام drop، ولا يحصل أي شيء للمتغير s2 لأنه
نُقل، بينما يغادر s1 النطاق أيضًا ويُحرر من الذاكرة باستخدام drop

fn gives_ownership() -> String {
// تنقل الدالة gives_ownership قيمتها المعادة إلى الدالة التي استدعتها

    let some_string = String::from("yours");
    // يدخل المتغير some_string إلى النطاق

    some_string                 // يُعاد some_string ويُنقل إلى الدالة المُستدعاة

}

// تأخذ هذه الدالة سلسلة نصية وتُعيد سلسلة نصية أخرى
fn takes_and_gives_back(a_string: String) -> String { //  يدخل a_string
إلى النطاق
```

```
    a_string  // يُعاد a_string ويُنقل إلى الدالة المُستدعاة
}
```

[الشيفرة 4: تحويل ملكية القيمة المُعادة]

تتبع ملكية المتغير نفس النمط في كل مرة، وهو: "إسناد قيمة إلى متغير آخر ينقلها"، وعندما يخرج متغير يتضمن على بيانات ضمن الكومة من النطاق، تُحرر قيمته باستخدام drop إلا إذا نُقلت ملكية البيانات إلى متغير آخر.

على الرغم من نجاح هذه العملية، إلا أن عملية أخذ الملكية ومن ثم إعادتها عند كل دالة عملية رتيبة بعض الشيء. ماذا لو أردنا أن نسمح لدالة ما باستخدام القيمة دون الحصول على ملكيتها؟ إنه أمر مزعج جدًا أن أي شيء نمرره سيحتاج أيضًا لإعادة تمريره مجددًا إذا أردنا استخدامه من جديد، بالإضافة إلى أي بيانات نحصل عليها ضمن متن الدالة التي قد نحتاج أن نُعيدها أيضًا.

تسمح لنا رست بإعادة عدّة قيم باستخدام مجموعة كما هو موضح في الشيفرة 5.

اسم الملف: src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // يُعيد len() طول السلسلة النصية

    (s, length)
}
```

[الشيفرة 5: إعادة الملكية إلى المعاملات]

هذه العملية طويلة قليلًا وتتطلب كثيرًا من الجهد لشيء يُشاع استخدامه، ولحسن حظنا لدى رست ميزة لاستخدام القيمة دون تحويل ملكيتها وتدعى **المراجع references**، وسنستعرضها في القسم التالي.

4.2 المراجع References والاستعارة Borrowing

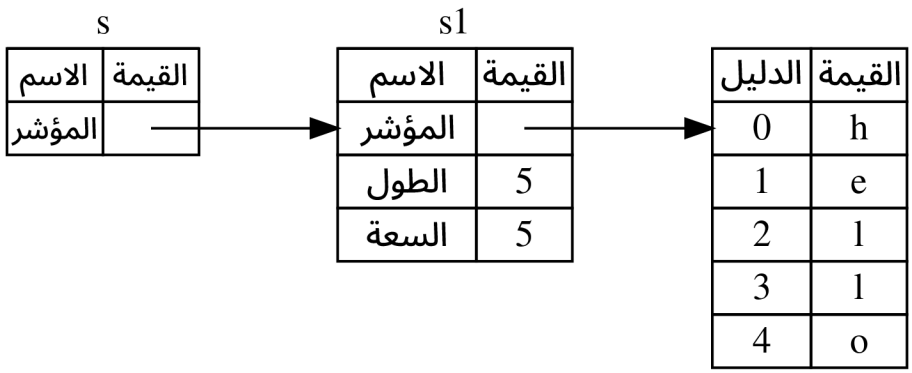
كانت مشكلتنا باستخدام الصف tuple في القسم السابق (الشيفرة 5) هو أنه علينا إعادة النوع `String` إلى القيمة المُستدعية ليتسنى لنا استخدام النوع `String` حتى بعد استدعاء الدالة `calculate_length`، وذلك لأن النوع `String` نُقل إلى `calculate_length`. بدلًا مما سبق يمكننا استخدام مرجع reference إلى القيمة `String`؛ والمرجع هو أشبه بالمؤشر pointer، إذ يُمثل عنوانًا يمكنك اتباعه للوصول إلى البيانات المخزنة ضمن العنوان المذكور، وتعود ملكية هذه البيانات إلى متغيرات أخرى مختلفة. من المضمون للمراجع - على عكس المؤشرات- أن تُشير إلى قيمة صالحة لنوع معين طوال دورة حياة المرجع.

إليك كيفية تعريف الدالة `calculate_length` واستخدامها مع احتوائها على مرجع لكائن بمثابة معاملٍ لها بدلًا من أخذ ملكية القيمة:

اسم الملف: src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

لاحظ أولًا أننا أزلنا الشيفرة البرمجية الخاصة بالصف في تصريح المتغير وقيمة الدالة المُعادة، كما أننا مررنا أيضًا `&s1` إلى الدالة `calculate_length` وأخذنا في تعريفها `&String` بدلًا من `String` وتُمثّل هذه الإشارة `&` المرجع، وتسمح لك بالإشارة إلى قيمة دون أخذ ملكيتها، ويوضح الشكل التالي هذا المفهوم.



الشكل 5: مخطط يوضح String s& الذي يُشير إلى String s1

> * عكس عملية المرجع باستخدام & هي التحصيل dereferencing ويمكن تحقيقها باستخدام عامل التحصيل وسنرى بعض استخدامات التحصيل بالإضافة لتفاصيل العملية لاحقًا.

دعنا نأخذ نظرةً أعمق على استدعاء الدالة هنا:

```
    let s1 = String::from("hello");

    let len = calculate_length(&s1);
```

تسمح لك الكتابة `&s1` بإنشاء مرجع يشير إلى القيمة `s1` إلا أنه لا ينقل الملكية، وبالتالي -وبما أنه لا يملكها- لن تُسقط القيمة (باستخدام `drop`) التي يشير إليها عندما يتوقف المرجع عن استخدامها.

وبالمثل، تستخدم شارة الدالة الرمز `&` للإشارة إلى أن نوع المعامل `s` هو مرجع. دعنا نضيف بعض التعليقات التوضيحية:

```
fn calculate_length(s: &String) -> usize { // يمثل s مرجعًا إلى String
    s.len()
} // يخرج s من النطاق هنا إلا أنه لا يُسقط لأنه لا يمتلك القيمة التي يشير إليها
```

النطاق الذي يحتوي على المتغير `s` هو مماثل لأي نطاق معامل دالة، إلا أن القيمة المُشار إليها باستخدام المرجع لا تُسقَط عندما نتوقف عن استخدام `s` وذلك لأن `s` لا يملك القيمة. لا نحتاج لإعادة القيم عندما تحتوي الدوال على مراجع مثل معاملات بدلًا من القيم الفعلية حتى نستطيع منح الملكية مجددًا، وذلك لأننا لم ننقل الملكية في المقام الأول.

نطلق على عملية إنشاء المراجع عملية الاستعارة borrowing، فكما الحال في الحيـاة الواقعيـة، إذا امتلـك شخصٌ ما غرضًا، فيمكنك استعارته منه، وعند الانتهاء من الغرض عليـك أن تُعيـده إلى ذلـك الشـخص لأنـك لا تملك الغرض.

ما الذي يحصل إذًا عندما نحاول التعديل على شيء مُستعار؟ جرّب تنفيذ الشيفرة 6 (تحذير: لن تعمل)

اسم الملف: src/main.rs

```
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

[الشيفرة 6: محاولة التعديل على قيمة مُستعارة]

وسيظهر الخطأ التالي:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind
a `&` reference
 --> src/main.rs:8:5
  |
| fn change(some_string: &String) {
  |                        ------- help: consider changing this to be
a mutable reference: `&mut String`
|     some_string.push_str(", world");
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `some_string` is a `&`
reference, so the data it refers to cannot be borrowed as mutable

For more information about this error, try `rustc --explain E0596`.
error: could not compile `ownership` due to previous error
```

المراجع غير قابلة للتعديل immutable افتراضيًا كما هو الحال مع المتغيرات، لذا لا يُمكنـك التعـديل على شيء باستخدام المرجع.

4.2.1 المراجع القابلة للتعديل

يُمكننا تصحيح الشيفرة 6 السابقة لكي تسمح لنا بتعديل قيمة مُستعارة باستخدام تعديلات بسيطة، وذلك باستخدامنا مرجعًا قابلًا للتعديل mutable reference:

اسم الملف: src/main.rs

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

علينا أولًا أن نعدل المتغير `s` ليصبح `mut`، ثم نُنشئ مرجعًا قابلًا للتعديل باستخدام `&mut s` عند نقطة استدعاء الدالة `chang`، ومن ثم تحديث شارة الدالة حتى تقبل مرجعًا قابلًا للتعديل بكتابة `some_string: &mut String`، إذ يدلنا هذا بكل وضوح على أن الدالة `chang` ستُعدّل من القيمة التي استعارتها.

للمراجع القابلة للتعديل قيدٌ واحدٌ كبير، وهو أنه لا يمكنك الحصول على أكثر من مرجع إلى قيمة إذا كان لتلك القيمة مرجعًا قابلًا للتعديل. نُحاول في الشيفرة البرمجية التالية إنشاء مرجعين قابلَين للتعديل يشيران إلى s:

اسم الملف: src/main.rs

```
    let mut s = String::from("hello");

    let r1 = &mut s;
    let r2 = &mut s;

    println!("{}, {}", r1, r2);
```

إليك الخطأ:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0499]: cannot borrow `s` as mutable more than once at a time
```

```
 --> src/main.rs:5:14
  |
|     let r1 = &mut s;
  |              ------ first mutable borrow occurs here
|     let r2 = &mut s;
  |              ^^^^^^ second mutable borrow occurs here
|
|     println!("{}, {}", r1, r2);
  |                        -- first borrow later used here

For more information about this error, try `rustc --explain E0499`.
error: could not compile `ownership` due to previous error
```

يدل هذا الخطأ على أن الشيفرة البرمجية غير صالحة لأنه لا يُمكننا استعارة القيمة `s` على أنها قيمة قابلة للتعديل أكثر من مرة واحدة، إذ نستعير القيمة القابلة للتعديل للمرة الأولى في `r1` ويجب أن نحافظ على الاستعارة حتى تُستخدم القيمة في `println!`، إلا أننا حاولنا أيضًا إنشاء مرجع قابل للتعديل آخر في `r2` يستعير نفس البيانات الموجودة في `r1` وذلك بين إنشاء المرجع القابل للتعديل الأول وبين استخدامه.

يسمح القيد الذي يمنع وجود عدة مراجع قابلة للتعديل تشير لنفس البيانات في نفس الوقت بتعديل البيانات ولكن بطريقة مُقيّدة جدًا، وهو شيء يعاني منه معظم متعلمي لغة رست الجدُد وذلك لأن معظم اللغات تسمح لك بتعديل ما تشاء. الميزة من هذا القيد هو أن رست تمنع سباق البيانات data races عند وقت التصريف، وسباق البيانات هو مشابه لحالة السباق race condition ويحدث عند حدوث أحد الحالات الثلاث:

- محاولة مؤشرين أو أكثر الوصول إلى نفس البيانات في نفس الوقت.
- استخدام واحد من المؤشرات على الأقل للكتابة إلى البيانات.
- عدم وجود آلية مُستخدمة لمزامنة الوصول إلى البيانات.

تتسبب سباقات البيانات بسلوك غير معرّف undefined behaviour وقد يكون تشخيص المشكلة وتصحيحها صعبًا عندما تحاول تتبع الخطأ عند وقت التشغيل، وتمنع رست هذه المشكلة برفض تصريف الشيفرة البرمجية التي تحتوي على سباقات البيانات.

يُمكننا استخدام الأقواس المعقوصة curly brackets لإنشاء نطاق جديد، مما يسمح بوجود مراجع قابلة للتعديل، إلا أن الشرط هو عدم تواجد المراجع القابلة للتعديل ضمن النطاق في ذات الوقت:

```
    let mut s = String::from("hello");
```

```
    {
        let r1 = &mut s;
    } // يخرج r1 من النطاق هنا وبالتالي يمكننا إنشاء مرجع جديد دون أي مشاكل

    let r2 = &mut s;
```

تُجبرنا رست على قاعدة مشابهة لجمع المراجع القابلة وغير القابلة للتعديل. تولّد الشيفرة البرمجية التالية خطأً:

```
    let mut s = String::from("hello");

    let r1 = &s; // لا توجد مشكلة
    let r2 = &s; // لا توجد مشكلة
    let r3 = &mut s; // هناك مشكلة كبيرة

    println!("{}, {}, and {}", r1, r2, r3);
```

إليك الخطأ:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed
as immutable
 --> src/main.rs:6:14
  |
|     let r1 = &s; // no problem
  |              -- immutable borrow occurs here
|     let r2 = &s; // no problem
|     let r3 = &mut s; // BIG PROBLEM
  |              ^^^^^^ mutable borrow occurs here
|
|     println!("{}, {}, and {}", r1, r2, r3);
  |                                -- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` due to previous error
```

كما أنه لا يمكننا إنشاء مرجع قابل للتعديل بينما لدينا مرجع غير قابل للتعديل يشير إلى القيمة ذاتها.

لا يتوقع مستخدموا المراجع غير القابلة للتعديل بـأن تتغـير القيمـة من تلقـاء نفسـها، لكن تسـمح المراجـع المتعددة غير القابلة للتعديل بذلك لأنـه لا يوجـد أي شـيء يقـرأ البيانـات ولديـه القـدرة على التـأثير على أي من المراجع التي تقرأ البيانات.

لاحظ بأن نطاق المراجع يبدأ من مكان إنشائها ويستمر إلى آخـر مكـان اسـتُخدم فيـه المرجـع، فعلى سـبيل المثال، يمكن تصريف الشيفرة البرمجية التالية بنجاح لأن اسـتخدام المراجـع غـير القابلـة للتعـديل الأخـير في `println!` يحدث قبل إنشاء المرجع القابل للتعديل:

```
    let mut s = String::from("hello");

    let r1 = &s; // لا يوجد مشكلة
    let r2 = &s; // لا يوجد مشكلة
    println!("{} and {}", r1, r2);
    // لن يُستخدم المتغيرين r1 و r2 بعد هذه النقطة

    let r3 = &mut s; // لا يوجد مشكلة
    println!("{}", r3);
```

ينتهي نطاق كل من المرجعَين غير القابلَين للتعديل `r1` و `r2` بعد `println!` وهي آخر نقطة لاستخدامهما قبل إنشاء المرجع القابل للتعديل `r3`. لا تتداخل النطاقات ولذلك تكون الشيفرة البرمجية صـالحة، وتـدعى قـدرة المصرف على معرفة المراجع التي لا تُستخدم بعد الآن في نهاية النطاق باسم دورات الحياة غير المُعجمية Non-Lexical Lifetimes -أو اختصارًا NLL -ويمكنك القراءة عنها من هنا.

على الرغم من كون أخطاء الاستعارة أخطاءً مثيرةً للإحباط في بعض الأحيان، إلا أنه يجب عليك أن تتـذكر أن مصرّف رست يُشير إلى خطأ قبل أوانه (عند وقت التصريف بـدلًا من وقت التشـغيل) ويوضـح لـك بالتفصـيل مكان المشكلة حتى لا تضطر إلى تعقب المشكلة إذا كانت بياناتك تحتوي قيمًا مختلفة لتوقعاتك.

4.2.2 المراجع المعلقة

من السهل في اللغات التي تدعم المؤشرات أن تُنشئ مؤشرات معلقة dangling pointer بصورةٍ خاطئة؛ وهي مؤشرات تشير إلى موضع في الذاكرة مُعطًى لشخص آخر وذلك بتحرير المساحة مع المحافظة على المؤشر الذي يشير إلى الذاكرة، وهذا غـير ممكن الحصـول في رسـت، فعلى النقيض تمامًـا يضـمن المصـرف ألا تكـون جميع المراجع مُعلّقة، لأنه سيتأكد من عدم مغادرة البيانات التي يشير إليها المرجع النطاق قبل أن يُغادر المرجـع النطاق أولًا.

دعنا نجرّب إنشاء مرجع معلّق لملاحظة كيف تمنع رست وجودها بخطأ عند وقت التصريف:

اسم الملف: src/main.rs

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

إليك الخطأ:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0106]: missing lifetime specifier
 --> src/main.rs:5:16
  |
| fn dangle() -> &String {
  |                ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but
there is no value for it to be borrowed from
help: consider using the `'static` lifetime
  |
| fn dangle() -> &'static String {
  |                ~~~~~~~~

For more information about this error, try `rustc --explain E0106`.
error: could not compile `ownership` due to previous error
```

يُشير هذا الخطأ إلى الميزة الـتي لم نكتشـفها بعـد، ألا وهي دورات الحيـاة lifetimes، وسنناقشـها بتوسُّـع لاحقًا، ولكن إذا تغاضينا عن الجزء الخاص بدورات الحياة، فإن رسالة الخطأ تحتوي سبب المشكلة:

```
this function's return type contains a borrowed value, but there is no
value
for it to be borrowed from
```

دعنا نأخذ نظرةً أقرب لما يحدث في كل مرحلة من مراحل الدالة `dangle`:

اسم الملف: src/main.rs

```
fn main() {
    let reference_to_nothing = dangle();
}
fn dangle() -> &String { // تُعيد الدالة dangle مرجعًا إلى النوع String

    let s = String::from("hello"); // يشكل s السلسلة String الجديدة

    &s // نُعيد المرجع إلى النوع String واسمه s
} // يخرج s من النطاق هنا ويُسقَط وتُحرّر ذاكرته. خطر!
```

ستُحرّر s عند الانتهاء من تنفيذ الشيفرة البرمجية داخل الدالة dangle، لأن s مُنشـأة بـداخل dangle، إلا أننا حاولنا إعادة مرجع إليها وهذا يعني أن المرجع سيشير إلى قيمة String غير صالحة، وهـذا مـا يجب تجنُّب حدوثه، بالتالي لن تسمح لنا رست بفعل ذلك.

يكمن الحل هنا بإعادة String مباشرةً:

```
fn main() {
    let string = no_dangle();
}
fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

يعمل هذا الحل دون أي مشاكل، إذ تُنقل الملكية ولا يُحرّر أي حيز من الذاكرة.

4.2.3 قوانين المراجع

دعنا نلخص أهم النقاط التي ناقشناها بخصوص المراجع:

- يُمكنك في نقطة من الزمن استخدام مرجع واحد قابل للتعديل **أو** عدة مراجع غير قابلة للتعديل.
- يجب أن تكون المراجع صالحة على الدوام.

سننظر في الفقرات التالية إلى نوع مختلف من المراجع هو الشرائح slices.

4.3 الشرائح Slices

4.3.1 نوع الشريحة

تسمح لك الشرائح بالإشارة reference إلى سلسلة متتابعة من العناصر ضمن تجميعة collection بدلًا من الإشارة إلى كامل التجميعة، وتشبه الشريحة المرجع ولذا فهو لا يحتوي على ملكية.

إليك مشكلةً برمجيةً صغيرة: اكتب دالةً تأخذ سلسلةً نصيةً من كلمات مفصول ما بينها بمسافات وتُعيد الكلمة الأولى التي تجدها ضمن تلك السلسلة النصية، إذا لم تعثر الدالة على مسافة ضمن السلسلة النصية، فهذا يعني أن السلسلة النصية مؤلفةٌ من كلمة واحدة فقط وعليها أن تُعيد كامل السلسلة النصية.

لنرى كيف يمكننا كتابة بصمة signature الدالة دون استخدام الشرائح، وذلك حتى نفهم المشكلة التي تحلها الشرائح:

```
fn first_word(s: &String) -> ?
```

للدالة `first_word` معامل `&String`، وهذا ما لا بأس فيه لأننا لا نحتاج الملكية عليه، لكن ما الذي يجب أن تُعيده الدالة؟ لا توجد لدينا أي طريقة لتحديد جزء من السلسلة النصية، إلا أننا نستطيع إعادة دليل index نهاية الكلمة بالاستفادة من المسافة، لنجرّب هذه الطريقة كما هو موضح في الشيفرة 7.

اسم الملف: src/main.rs

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

[الشيفرة 7: الدالة `first_word` تُعيد قيمة دليل بحجم بايت إلى معامل `String`]

علينا أن نحوّل `String` إلى مصفوفة من البايتات لأننا بحاجة للمرور بعناصر `String` عنصرًا تلو الآخر للبحث عن مسافات، وذلك باستخدام التابع `as_bytes`:

```
let bytes = s.as_bytes();
```

من ثم نُنشئ مُكرّرًا iterator على مصفوفة البايتات باستخدام تابع `iter`:

```
for (i, &item) in bytes.iter().enumerate() {
```

سنُناقش المُكرّرات بالتفصيل لاحقًا، أما الآن فكل ما عليك معرفته هو أن `iter` تابع يُعيد كل عنصر في تجميعة، وأن `enumerate` يُغلّف wraps نتيجة `iter` ويُعيد كل عنصر على أنه جزءٌ من الصف tuple بدلًا من ذلك، إذ يمثّل العنصر الأول من الصف المُعاد من التابع `enumerate` دليلًا، بينما يمثل العنصر الثاني مرجعًا إلى العنصر في التجميعة، وهذه الطريقة أسهل من حساب أدلة العناصر بأنفسنا.

يُمكننا استخدام الأنماط patterns لتفكيك الصف وذلك بالنظر إلى أن التابع `enumerate` يُعيد صفًا، وسنناقش الأنماط بالتفصيل لاحقًا. نُحدد في حلقة `for` النمط الذي يحتوي على `i`، والذي يمثل الدليل الموجود في الصف و `&item` للبايت الوحيد الموجود في الصف، ونستخدم `&` في النمط لأننا نحصل على مرجع للعنصر من `iter().enumerate()`..

نبحث عن البايت الذي يمثل المسافة داخل حلقة `for` باستخدام البايت المجرّد، وإن وجدنا مسافةً نُعيد مكانها، وإلا فنعيد طول السلسلة النصية باستخدام `s.len()`:

```
        if item == b' ' {
            return i;
        }
    }

    s.len()
```

لدينا طريقةٌ الآن لمعرفة دليل نهاية الكلمة الأولى في السلسلة النصية، إلا أن هناك مشكلةً في هذه الطريقة؛ إذ أننا نُعيد النوع `usize` بصورةٍ منفردة إلا أنه ذو معنى فقط في سياق `&String`، بكلمات أخرى، لا يوجد أي ضمان أن القيمة ستكون صالحة في المستقبل نظرًا لأنها قيمة منفصلة عن `String`. ألقِ نظرةً على الشيفرة 8 التي تستخدم الدالة `first_word` من الشيفرة 7.

اسم الملف: src/main.rs

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s); // word ستُخزّن القيمة 5 في

```

```
    s.clear(); // تُفرغ هذه التعليمة السلسلة النصية وتجعلها مساوية إلى القيمة ""

    // للمتغير word القيمة 5 هنا إلا أنه لا يوجد أي سلسلة نصية تجعل من هذه القيمة ذات معنى،
وبالتالي القيمة عديمة الفائدة!
}
```

[الشيفرة 8: تخزين القيمة من استدعاء الدالة `first_word` ومن ثمّ تغيير محتويات `String`]

يُصرَّف البرنامج السابق دون أي أخطاء وسيعمل دون مشاكل إذا استخدمنا `word` بعد استدعاء `s.clear()`، وذلك لأن `word` ليست مرتبطة بحالة `s`إطلاقًا، إذ ما زالت تحتوي `word` على القيمة 5 حتى بعد استدعاء `s.clear()`، ويمكننا استخدام القيمة 5 مع المتغير `s` حتى نحاول استخراج الكلمة الأولى إلا أن ذلك سيتسبب بخطأ لأن محتويات `s` تغيرت منذ أن خزّننا 5 في `word`.

تُعد مراقبة الدليل الموجود في `word` خشيةً من فقدان صلاحيته بالنسبة للمتغير `s` عمليةً رتيبةً ومعرضةً للخطأ، وستصبح أسوأ إذا كتبنا دالة `second_word` تكون بصمتها على النحو التالي:

```
fn second_word(s: &String) -> (usize, usize) {
```

الآن وبما أننا نتتبع دليل البداية والنهاية، فهذا يعني أنه لدينا قيم أكثر لحسابها من البيانات في حالة معينة، إلا أن هذه القيم غير مرتبطة بحالة ما إطلاقًا، أصبح لدينا الآن إذًا ثلاثة متغيرات غير مرتبطة مع بعضها بعضًا ويجب أن نربطها.

لدى رست لحسن الحظ الحل لهذه المشكلة، وهي سلسلة الشرائح النصية string slices.

4.3.2 شرائح السلاسل النصية

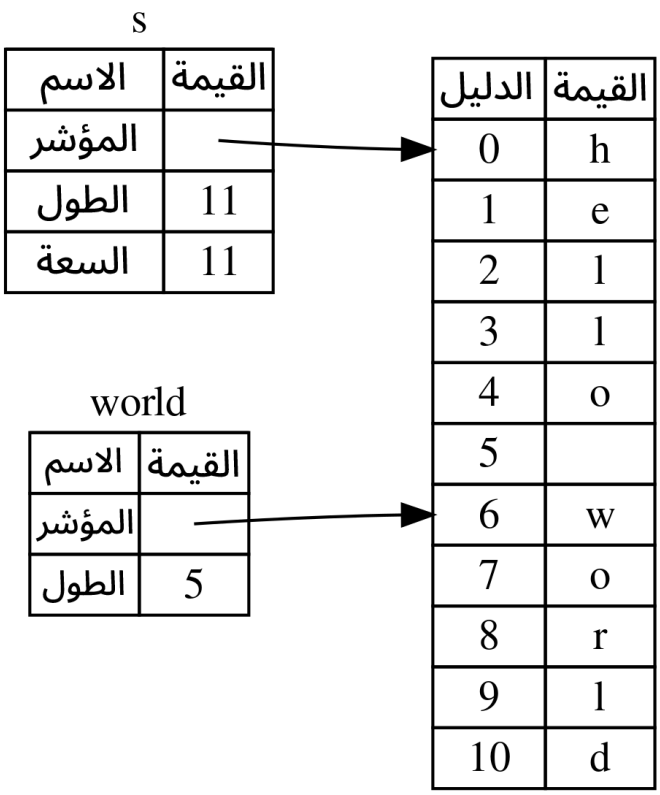
شرائح السلاسل النصية هي مرجعٌ لجزء من النوع `String` وتُكتب بالشكل التالي:

```
fn main() {
    let s = String::from("hello world");

    let hello = &s[0..5];
    let world = &s[6..11];
}
```

نستخدم المرجع `hello` الذي يمثّل مرجعًا لجزء من النوع `String` بدلًا من استخدام مرجع لكامل النوع، والجزء مُحدّد باستخدام `[0..5]` بت. نُنشئ هنا شرائحًا باستخدام مجال باستخدام الأقواس وذلك بتحديد دليل البداية ودليل النهاية بالشكل التالي: `[starting_index..ending_index]`؛ إذ يُمثل `starting_index` موضع بداية الشريحة؛ بينما يمثل `ending_index` الموضع الذي يلي موضع نهاية

الشريحة. يُخزّن هيكل بيانات الشريحة داخليًا كلًا من موضع البداية وطول الشريحة، الذي تحصل عليه من طرح `starting_index` من `ending_index`، لذا في حالة `let world = &s[6..11]` نحصل على شريحة باسم `world` تحتوي على مؤشر يشير إلى البايت الموجود في الدليل 6 للسلسلة `s` بقيمة طول مساوية إلى 5.

يوضح الشكل 6 العملية السابقة.



الشكل 6: شريحة سلسلة نصية تُشير إلى جزء من String

يُمكنك إهمال دليل البدء قبل النقطتين `..` في المجال إذا أردت البدء من الدليل صفر، أي أن الشريحتين التاليتن متماثلتان:

```
let s = String::from("hello");

let slice = &s[0..2];
let slice = &s[..2];
```

كما يمكنك إهمال دليل النهاية إذا أردت أن تُضمّن السلسلة النصية إلى النهاية، أي أن الشريحتين التاليتين متماثلتان:

```
let s = String::from("hello");

let len = s.len();
```

```
let slice = &s[3..len];
let slice = &s[3..];
```

أخيرًا، يمكنك إهمال كل من دليل البداية ودليل النهاية ضمن المجال إذا أردت الحصول على كامل السلسلة النصية، والشريحتان التاليتان متماثلتان:

```
let s = String::from("hello");

let len = s.len();

let slice = &s[0..len];
let slice = &s[..];
```

> يجب أن تمثّل أدلّة شرائح السلاسل النصية محرفًا صالحًا من ترميز UTF-8، وإلا سيتوقف البرنامج ويعرض خطأً إذا حاولت إنشاء شريحة سلسلة نصية منتصف محرف متعدد البايتات multibyte character، ونفرض هنا في هذا القسم أن المحارف بترميز آسكي ASCII فقط بهدف البساطة، وسنناقش مفصّلًا التعامل مع محارف بترميز UTF-8 لاحقًا.

بعد تعرّفنا لما سبق، دعنا نكتب دالة تُعيد شريحة نسميها `first_word`، والنوع الذي يمثّل شريحة السلسلة النصية هو `&str`:

اسم الملف: src/main.rs

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
fn main() {}
```

نحصل في البرنامج السابق على دليل نهاية الكلمة بصورةٍ مشابهة لما فعلناه في الشيفرة 7 وذلك بالبحث عن أوّل ظهور لمسافة، وعندما نجد هذه المسافة نُعيد شريحة سلسلة نصية باستخدام بداية السلسلة النصية مثل دليل بداية ودليل المسافة مثل دليل نهاية.

نحصل على قيمة واحدة متعلقة بالبيانات التي لدينا بعد استدعاء الدالة `first_word`، وتتألف القيمة من مرجع إلى نقطة البداية لشريحة السلسلة النصية وعدد العناصر في تلك الشريحة.

يمكننا أن نجعل الدالة `second_word` تُعيد شريحة أيضًا:

```
fn second_word(s: &String) -> &str {
```

أصبح لدينا الآن واجهة برمجية API واضحة صعب العبث فيها، إذ سيتأكد المصرّف من أن مراجع النوع `String` هي مراجع صالحة. أتتذكر الخطأ الذي واجهناه في البرنامج الموجود في الشيفرة 8 عندما حصلنا على دليل نهاية الكلمة الأولى ومن ثمّ مسحنا السلسلة النصية مما جعل الدليل غير صالح؟ كانت الشيفرة تلك غير صحيحة منطقيًا ولكننا لم نحصل على أي أخطاء مباشرة واضحة، وستظهر المشكلة لاحقًا إذا حاولت استخدام دليل السلسلة النصية الفارغة، إلا أن شرائح السلاسل النصية تجعل من هذا الخطأ مستحيلًا وستعلمنا بحدوث خطأ في شيفرتنا البرمجية في وقت مبكّر، وعلى سبيل المثال، نحصل على خطأ وقت التصريف إذا استخدمنا إصدار شريحة السلسلة النصية من الدالة `first_word`.

اسم الملف: src/main.rs

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);
```

```
    s.clear(); // خطأ!

    println!("the first word is: {}", word);
}
```

إليك خطأ التصريف:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed
as immutable
  --> src/main.rs:18:5
   |
|     let word = first_word(&s);
   |                           -- immutable borrow occurs here
|
|     s.clear(); // error!
   |     ^^^^^^^^^ mutable borrow occurs here
|
|     println!("the first word is: {}", word);
   |                                       ---- immutable borrow later
used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` due to previous error
```

تذكّر من قواعد الاستعارة borrowing rules أنه لا يمكننا أخذ مرجع قابل للتعديل من شيء إذا كان لدينا مرجع غير قابل للتعديل لهذا الشيء مسبقًا، ولأن `clear` بحاجة لحذف محتويات `String`، فهي بحاجة للحصول على مرجع قابل للتعديل، يستخدم `println!` بعد الاستدعاء للدالة `clear` المرجع في `word`، لذلك لا بدّ للمرجع غير القابل للتعديل أن يكون صالحًا بحلول تلك النقطة ولذلك تمنع رست وجود مرجع قابل للتعديل في `clear` ومرجع غير قابل للتعديل في `word` في الوقت ذاته مما يتسبب بفشل عملية التصريف. لم تكتفي رست بجعل الواجهة البرمجية أسهل للتعامل بل أقصَت صنفًا كاملًا من الأخطاء ممكنة الحدوث عند وقت التصريف.

ا. السلاسل النصية المجردة هي شرائح

تذكر أننا تحدثنا عن السلاسل النصية المجردة بكونها تُخزّن بداخل الملف التنفيذي الثنائي، ويمكننا الآن فهم السلاسل النصية المجردة بوضوح بما أننا نعرف الشرائح:

```
let s = "Hello, world!";
```

نوع s هنا هو &str وهي شريحة تُشير إلى جزء محدد من الملف الثنائي، وهذا السبب في كـون السلاسـل النصية غير قابلة للتعديل، وبالتالي يكون المرجع &str مرجعًا غير قابل للتعديل.

ب. شرائح السلاسل النصية مثل معاملات

تدلنا معرفة أنه يُمكننا أخذ شرائح من السلاسل النصية المجردة والقيم من النوع String على أنه نسـتطيع إجراء تحسين واحد إضافي على first_word وهو بصمة الدالة:

```
fn first_word(s: &String) -> &str {
```

قد يكتب مبرمج لغة رست خبير بصمة الدالة السابقة الموضحة في الشيفرة 9 بـدلًا من ذلـك والسـبب في هذا هو أن البصمة السابقة تسمح لنا باستخدام الدالة ذاتها على قيمتَي &String و &str.

```
fn first_word(s: &str) -> &str {
```

[الشيفرة 9: تحسين دالة first_word باستخدام شريحة سلسلة نصية لنوع المُعامل s]

يُمكننا تمرير شريحة السلسلة النصية مباشرةً في هذه الحالة، فعلى سبيل المثال يمكننـا تمريـر شـريحة من النوع String أو مرجع إليه إذا كان لدينا النوع String في معـاملات الدالـة، وهـذا الأمـر ممكن بفضـل مـيزة التحصيل القسري deref corecions التي سنتكلم عنها بالتفصيل لاحقًا. يجعـل تعريـف الدالـة لتأخـذ شـريحة سلسلة نصية بدلًا من مرجع إلى النوع String من واجهتنا البرمجية شاملة الاستخدام أكثر ومفيدةً دون خسـارة أي من وظائفها.

اسم الملف: src/main.rs

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
fn main() {
```

```
    let my_string = String::from("hello world");

    // تعمل الدالة first_word على شرائح النوع String سواءً كانت شريحة جزئية أو شريحة تشكل كامل السلسلة
    let word = first_word(&my_string[0..6]);
    let word = first_word(&my_string[..]);
    // تعمل الدالة first_word على مراجع النوع String والمساوية إلى كامل شرائح String
    let word = first_word(&my_string);

    let my_string_literal = "hello world";

    // تعمل الدالة first_word على شرائح سلاسل نصية مجردة سواءً كانت مجردة أو جزئية
    let word = first_word(&my_string_literal[0..6]);
    let word = first_word(&my_string_literal[..]);

    // بما أن السلاسل النصية المجردة هي شرائح سلاسل نصية بالأصل، فالتعليمة التالية تعمل أيضًا دون طريقة كتابة الشريحة
    let word = first_word(my_string_literal);
}
```

4.3.3 الشرائح الأخرى

شرائح السلاسل النصية هي شرائح خاصة بالسلاسل النصية كما قد تتوقع، إلا أن هناك أنـواع شـرائح عامـة أكثر، ألقِ نظرةً على المصفوفة التالية:

```
let a = [1, 2, 3, 4, 5];
```

قد نحتاج لاستخدام مرجع يشير إلى جزء من مصفوفة بطريقة مماثلة للسلسلة النصية، ويمكننا تحقيق ذلك الأمر وفق ما يلي:

```
let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];

assert_eq!(slice, &[2, 3]);
```

للشريحة النوع `&[i32]` وتعمل بطريقة مماثلة لشريحة السلسلة النصية وذلك بتخزين مرجع للعنصر الأول وطول الشريحة، وستستخدم هذا النوع من الشرائح لكافة أنواع التجميعات الأخرى، وسنناقش هذه التجميعات بالتفصيل عندما نتحدث عن الأشعة لاحقًا.

4.4 خاتمة

تضمن مفاهيم مثل الملكية ownership والاستعارة borrowing والشرائح slices أمان الذاكرة في برامج رست عند وقت التصريف، وتُعطيك لغة رست التحكم على استخدام الذاكرة بالطريقة ذاتها في لغات برمجة النظم الأخرى، إلا أن وجود مالك للبيانات يجعل من تحرير الذاكرة بسيطًا عند مغادرة المالك من النطاق مما يعني أنه ليس عليك كتابة شيفرة برمجية وتشخيص أخطائها للحصول على هذا النوع من التحكم.

تؤثر الملكية على كيفية عمل كثيرٍ من أجزاء لغة رست، ولذلك سنتحدث عن هذه المفاهيم بتعمُّق أكبر ضمن الكتاب. دعنا ننتقل الآن إلى الفصل الخامس وننظر إلى كيفية تجميع أجزاء من البيانات مع بعضها البعض باستخدام الهياكل `struct`.

دورة تطوير التطبيقات
باستخدام لغة بايثون

احترف البرمجة وتطوير التطبيقات مع أكاديمية حسوب
والتحق بسوق العمل فور انتهائك من الدورة

التحق بالدورة الآن

5. استخدام الهياكل لتنظيم البيانات

الهيكل struct أو البنية structure هو نوع بيانات مُخصّص يسمح لنا باستخدام عدة قيم بأسماء مختلفة في مجموعة واحدة ذات معنًى ما. يشبه **الهيكل** سمات attributes بيانات الكائن وفقًا لمفهوم البرمجة كائنية التوجه أو OOP.

سنقارن في هذا الفصل بين الصفوف tuples والهياكل، ونستعرض كل منها بناءً على ما تعلمته سابقًا، وسنوضح الحالات التي يكون فيها استخدام الهياكل خيارًا أفضل لتجميع البيانات، وكذلك كيفية تعريف وإنشاء الهياكل، إضافةً إلى تعريف الدوال المرتبطة بها وبالأخص الدوال التي تحدد السلوك المرتبط بنوع الهيكل، والتي تُدعى **التوابع methods**. تُعدّ الهياكل والمُعدّدات enums (سنناقشها لاحقًا) من لبنات بناء نوع بيانات جديد ضمن نطاق برنامجك وذلك للاستفادة الكاملة من خاصية التحقق من الأنواع في رست عند وقت التصريف.

5.1 تعريف وإنشاء نسخة من الهياكل

تُشابه الهياكل الصفوف التي ناقشناها سابقًا وذلك من حيث تخزينها لعدة قيم مترابطة، إذ يمكن للهياكل أن تُخزّن أنواع بيانات مختلفة كما هو الحال في الصفوف، إلا أننا نسمّي كل جزء من البيانات ضمن الهيكل بحيث يكون الهدف منها واضحًا على عكس الصفوف، وهذا يعني أن الهياكل أكثر مرونة من الصفوف إذ أننا لا نعتمد على ترتيب البيانات بعد الآن لتحديد القيمة التي نريدها.

نستخدم الكلمة المفتاحية `struct` لتعريف الهيكل ونُلحقها باسمه، ويجب أن يصِف اسم الهيكل استخدام البيانات التي يجمعها ويحتويها، ومن ثمّ نستخدم الأقواس المعقوصة curly brackets لتعريف أسماء وأنواع البيانات التي يحتويها الهيكل وتُدعى هذه البيانات باسم **الحقول fields**، فعلى سبيل المثال توضح الشيفرة 1 هيكلًا يحتوي داخله معلومات تخص معلومات عن حساب مستخدم.

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

[الشيفرة 1: تعريف الهيكل User]

نُنشئ نسخةً instance من الهيكل الذي عرّفناه حتى نستطيع استخدامه، ومن ثم نحدّد قيمًا لكل حقل ضمنه. نستطيع إنشاء نسخة من الهيكل عن طريق ذكر اسم الهيكل ومن ثم إضافة أقواس معقوصة تحتوي على ثنائيات من التنسيق `key: value`، إذ يمثّل المفتاح `key` اسم الحقل، بينما تمثّل القيمة `value` البيانات التي نريد تخزينها في ذلك الحقل، وليس من الواجب علينا ذكر الحقول بالترتيب المذكور في تصريح الهيكل؛ أي بكلمات أخرى، يُعدّ تعريف الهيكل بمثابة قالب عام للنوع وتملأ النُسخ ذلك القالب ببيانات معينة لإنشاء قيمة من نوع الهيكل، ويمكن مثلًا التصريح عن مستخدم ما كما هو موضح في الشيفرة 2.

```
fn main() {
    let user1 = User {
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
        sign_in_count: 1,
    };
}
```

[الشيفرة 2: إنشاء نسخة من الهيكل User]

نستخدم النقطة (.) للحصول على قيمة محددة من هيكل ما؛ فإذا أردنا مثلًا الحصول على البريد الإلكتروني الخاص بالمستخدم فقط، فيمكننا كتابة `user1.email` عندما نريد استخدام تلك القيمة؛ وإذا كانت النسخة تلك قابلة للتعديل mutable، فيمكننا تغيير القيمة باستخدام الطريقة ذاتها أيضًا وإسناد الحقل إلى قيمة جديدة. توضح الشيفرة 3 كيفية تغيير القيمة في الحقل `email` الخاصة بالهيكل `User` القابل للتعديل.

```
fn main() {
    let mut user1 = User {
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
```

```
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");
}
```

[الشيفرة 3: تعديل قيمة الحقل email الخاصة بنسخة من الهيكل User]

لاحظ أنه يجب أن تكون كامل النسخة قابلة للتعديل، إذ لا تسمح لنا رست بتعيين حقول معينة من الهيكل على أنها حقول قابلة للتعديل. يمكننا إنشاء نسخة جديدة من الهيكل في آخر تعبير من الدالة، بحيث تُعيد الدالة نسخةً جديدةً من ذلك الهيكل كما هو الحال في التعابير الأخرى.

توضح الشيفرة 4 الدالة `build_user` التي تُعيد نسخةً من الهيكل `User` باستخدام اسم مستخدم وبريد إلكتروني يُمرّران إليها، إذ يحصل الحقل `active` على القيمة `true`، بينما يحصل الحقل `sign_in_count` على القيمة "1".

```
fn build_user(email: String, username: String) -> User {
    User {
        email: email,
        username: username,
        active: true,
        sign_in_count: 1,
    }
}
```

[الشيفرة 4: الدالة build_user التي تأخذ بريد إلكتروني واسم مستخدم ومن ثمّ تُعيد نسخة من الهيكل User]

من المنطقي أن نُسمّي معاملات الدالة وفق أسماء حقول الهيكل، إلا أن تكرار كتابة أسماء الحقول والمتغيرات `email` و `username` يصبح رتيبًا بعض الشيء، وقد يصبح الأمر مزعجًا مع زيادة عدد حقول الهيكل، إلا أن هناك اختصارًا لذلك لحسن الحظ.

5.1.1 ضبط قيمة حقول الهيكل بطريقة مختصرة

يُمكننا استخدام طريقة مختصرة في ضبط قيمة حقول الهيكل بما أن أسماء معاملات الدالة وأسماء حقول الهيكل متماثلة في الشيفرة 4، ولاستخدام هذه الطريقة نُعيد كتابة الدالة `build_user` بحيث تؤدي الغرض ذاته دون تكرار أي من أسماء الحقول `email` و `username` كما هو موضح في الشيفرة 5.

```
fn build_user(email: String, username: String) -> User {
```

```
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

[الشيفرة 5: دالة build_user تستخدم طريقة إسناد قيم الحقول المختصر لأن لمعاملات email و username الاسم ذاته لحقول الهيكل]

نُنشئ هنا نسخةً جديدةً من الهيكل `User` الـذي يحتـوي على هيكـل يـدعى `email`، ونضـبط قيمـة الحقـل `email` إلى قيمة المعامل `email` المُمرّر إلى الدالـة `build_user`، وذلـك لأن للحقـل والمعامـل الاسـم ذاتـه وبذلك نستطيع كتابة `email` بدلًا من كتابة `email: email`.

5.1.2 إنشاء نسخ من نسخ أخرى عن طريق صيغة تحديث الهيكل

من المفيد غالبًا إنشاء نسخة جديدة من الهيكل، الذي يتضمن معظم القيم من نسخةٍ أخرى ولكن مع بعض التغييرات وذلك باستخدام صيغة تحديث الهيكل struct update syntax.

تظهر الشيفرة 6 كيفية إنشـاء نسـخة مسـتخدم `user` جديـد في المسـتخدم `user2` بصـورةٍ منتظمـة دون استخدام صيغة تحديث الهيكل، إذ ضبطنا قيمة جديدة لحقـل `email` بينمـا اسـتخدمنا نفس القيم للمسـتخدم `user1` المُنشأ في الشيفرة 5.

```
fn main() {
    // --snip--

    let user2 = User {
        active: user1.active,
        username: user1.username,
        email: String::from("another@example.com"),
        sign_in_count: user1.sign_in_count,
    };
}
```

[الشيفرة 6: إنشاء نسخة user باستخدام إحدى قيم المستخدم user1]

يمكننا باستخدام صيغة تحديث الهيكل تحقيق نفس التأثير وبشيفرة أقصر، كما هو موضح في الشيفرة 7، إذ تدل الصيغة `..` على أن باقي الحقول غير المضبوطة ينبغي أن يكون لها نفس قيم الحقول في النسخة المحددة.

```
fn main() {
    // --snip--

    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    };
}
```

[الشيفرة 7: استخدام صيغة تحديث الهيكل لضبط قيمة email لنسخة هيكل المستخدم User واستخدام بقية قيم المستخدم user1]

تنشئ الشيفرة 7 أيضًا نسخةً في الهيكل `user2` بنفس قيم الحقول `username` و `active` و `sign_in_count` للهيكل `user1` ولكن لها قيمة `email` مختلفة. يجب أن يأتي المستخدم `..user1` أخيرًا ليدل على أن الحقول المتبقية يجب أن تحصل على قيمها من الحقول المقابلة في الهيكل `user1`، ولكن يمكننا تحديد قيم أي حقل من الحقول دون النظر إلى ترتيب الحقول الموجود في تعريف الهيكل.

تستخدم صيغة تحديث الهيكل الإسناد =، لأنه ينقل البيانات تمامًا كما هو الحال في القسم "طرق التفاعل مع البيانات والمتغيرات: النقل" من الفصل السابق. لم يعد بإمكاننا في هذا المثال استخدام `user1` بعد إنشاء `user2` بسبب نقل السلسلة النصية `String` الموجودة في الحقل `username` من المستخدم `user1` إلى `user2`. إذا أعطينا قيم سلسلة نصية جديدة إلى `user2` لكلا الحقلين `username` و `email` واستخدمنا فقط قيم الحقلين `active` و `sign_in_count` من الهيكل `user1`، سيبقى الهيكل `user1` في هذه الحالة صالحًا بعد إنشاء `user2`. تكون أنواع بيانات الحقلين `active` و `sign_in_count` بحيث تنفّذ السمة `Copy`، وبالتالي سينطبق هنا الأسلوب الذي ناقشناه في القسم الأول من الفصل المشار إليه آنفًا.

5.1.3 استخدام هياكل الصفوف دون حقول مسماة لإنشاء أنواع مختلفة

تدعم رست الهياكل التي تبدو مشابهةً للصفوف، وتُدعى بهياكل الصفوف tuple structs، ولهياكل الصفوف حقول مشابهة للهياكل العادية إلا أنها عديمة التسمية، وإنما لكل حقل نوع بيانات فقط. هياكل الصفوف مفيدة عندما تريد تسمية صف ما وأن تجعل الصف من نوع مختلف عن الصفوف الأخرى، أو عندما لا تكون تسمية كل حقل في الهيكل الاعتيادي عملية ضرورية.

لتعريف هيكل صف، نبدأ بالكلمة المفتاحية `struct` ومن ثم اسم الصف متبوعًا بالأنواع الموجودة في الصف. على سبيل المثال، نعرّف هنا هيكلا صف بالاسم `Color` و `Point` ونستخدمهما:

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

لاحظ أن `black` و `origin` من نوعين مختلفين، وذلك لأنهما نسختان من هياكل صف مختلفة، إذ يُعدّ كل هيكل تُعرفه نوعًا مختلفًا حتى لو كانت الحقول داخل الهيكل من نوع مماثل لهيكل آخر، على سبيل المثال، لا يمكن لدالة تأخذ النوع `Color` وسيطًا أن تأخذ النوع `Point` على الرغم من أن النوعين يتألفان من قيم من النوع `i32`. إضافةً لما سبق، يُماثل تصرف هياكل الصفوف تصرف الصفوف؛ إذ يمكنك تفكيكها إلى قطع متفرقة أو الوصول إلى قيم العناصر المختلفة بداخلها عن طريق استخدام النقطة (.) متبوعةً بدليل العنصر، وهكذا.

5.1.4 الهياكل الشبيهة بالوحدات بدون أي حقول

يُمكنك أيضًا تعريف هياكل لا تحتوي على أي حقول، وتدعى **الهياكل الشبيهة بالوحدات unit-like structs** لأنها مشابهة لنوع الوحدة unit type `()` الذي تحدثنا عنه في الفصل 3. يُمكن أن نستفيد من الهياكل الشبيهة بالوحدات عندما نريد تطبيق سمة trait على نوع ما ولكننا لا نملك أي بيانات نريد تخزينها داخل النوع، وسنناقش مفهوم السمات لاحقًا.

إليك مثالًا عن تصريح وإنشاء نسخة من هيكل شبيه بالوحدات يدعى `AlwaysEqual`:

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

تُستخدم الكلمة المفتاحية `struct` لتعريف `AlwaysEqual`، ثم تُلحق بالاسم الذي نريده ومن ثم الفاصلة المنقوطة، ولا حاجة هنا للأقواس إطلاقًا. يمكننا بعد ذلك الحصول على نسخة من الهيكل `AlwaysEqual` في المتغير `subject` بطريقة مماثلة، وذلك باستخدام الاسم الذي عرّفناه مسبقًا دون أي أقواس عادية أو معقوصة. نستطيع تطبيق سلوك على النوع فيما بعد بحيث تُساوي كل نسخة من `AlwaysEqual` قيمة أي نوع مُسند

إليه، والتي من الممكن أن تكون قيمةً معروفة بهدف التجريب، وفي هذه الحالة لن نحتاج إلى أي بيانات لتطبيق هذا السلوك، وسترى لاحقًا كيف بإمكاننا تعريف السمات وتطبيقها على أي نوع بما فيه الهياكل الشبيهة بالوحدات.

5.1.5 ملكية بيانات الهيكل

استخدمنا في الشيفرة 1 النوع `String` في الهيكل `User` الموجود بدلًا عن استخدام نوع شريحة سلسلة نصية string slice type بالشكل `&str`، وهذا استخدام مقصود لأننا نريد لكل نسخة من هذا الهيكل أن تمتلك جميع بياناتها وأن تكون بياناتها صالحة طالما الهيكل بكامله صالح.

من الممكن للهياكل أيضًا أن تُخزِّن مرجعًا لبيانات يمتلكها شيء آخر، إلا أن ذلك يتطلب استخدام دورات الحياة lifetimes وهي ميزة في رست سنتحدث عنها لاحقًا، إذ تضمن دورات الحياة أن البيانات المُشار إليها باستخدام الهيكل هي بيانات صالحة طالما الهيكل صالح. دعنا نفترض أنك تريد تخزين مرجع في هيكل ما دون تحديد دورات الحياة كما يلي (لن ينجح الأمر):

اسم الملف: src/main.rs

```
struct User {
    active: bool,
    username: &str,
    email: &str,
    sign_in_count: u64,
}

fn main() {
    let user1 = User {
        email: "someone@example.com",
        username: "someusername123",
        active: true,
        sign_in_count: 1,
    };
}
```

سيشكو المصرّف ويخبرك أنه بحاجة محدّدات دورات الحياة lifetime specifiers:

```
$ cargo run
   Compiling structs v0.1.0 (file:///projects/structs)
error[E0106]: missing lifetime specifier
```

```
 --> src/main.rs:3:15
  |
|     username: &str,
  |               ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
~ struct User<'a> {
|     active: bool,
~     username: &'a str,
  |

error[E0106]: missing lifetime specifier
 --> src/main.rs:4:12
  |
|     email: &str,
  |            ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
~ struct User<'a> {
|     active: bool,
|     username: &str,
~     email: &'a str,
  |

For more information about this error, try `rustc --explain E0106`.
error: could not compile `structs` due to 2 previous errors
```

سنناقش كيفية حل هذه المشكلة لاحقًا، بحيث يمكنك تخزين المراجع في الهياكل، إلا أننا سنستخدم حاليًا الأنواع المملوكة owned types مثل `String` بدلًا عن المراجع مثل `&str` لتجنب هذه المشكلة.

5.2 مثال على برنامج يستخدم الهياكل

دعنا نكتب برنامجًا يحسب مساحة مستطيل حتى نفهم فائدة استخدام الهياكل وأين نستطيع الاستفادة منها. نبدأ أوّلًا بكتابة البرنامج باستخدام متغيرات منفردة، ومن ثمّ نعيد كتابة البرنامج باستخدام الهياكل بدلًا من ذلك.

نُنشئ مشروعًا ثنائيًا جديدًا باستخدام كارجو Cargo باسم "rectangles"، ويأخذ هذا البرنامج طول وعـرض المستطيل بالبيكسل pixel ويحسب مساحة المستطيل. توضح الشـيفرة 8 برنامجًـا قصـيرًا ينفـذ ذلـك بإحـدى الطرق ضمن الملف "src/main.rs".

```
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        "The area of the rectangle is {} square pixels.",
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

[الشيفرة 8: حساب مساحة المستطيل المُحدّد بمتغيّرَي الطول والعرض]

دعنا الآن نُنفّذ البرنامج باستخدام الأمر `cargo run`:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target/debug/rectangles`
The area of the rectangle is 1500 square pixels.
```

ينجح برنامجنا في حساب مساحة المستطيل باستدعاء الدالـة `area` باسـتخدام بُعـدَي المسـتطيل، إلا أنـه يمكننا تعديل الشيفرة البرمجية لتصبح أكثر وضوحًا وسهلة القراءة.

تكمن المشكلة في شيفرتنا البرمجية الحالية في بصمة signature الدالة `area`:

```
fn area(width: u32, height: u32) -> u32 {
```

يُفترض أن تحسب الدالة `area` مساحة مستطيل واحد، إلا أن الدالة التي كتبناها تأخذ مُعـاملَين وليس من الواضح حاليًا أن المُعاملين مرتبطان فيما بينهما، ومن الأفضل هنا أن نجمع قيمة الطـول والعـرض سـويًّا، وقـد ناقشنا كيف قد نفعل ذلك سابقًا باستخدام الصفوف.

5.2.1 إعادة كتابة البرنامج باستخدام الصفوف

توضح الشيفرة 9 إصدارًا آخر من برنامجنا باستخدام الصفوف.

```
fn main() {
    let rect1 = (30, 50);

    println!(
        "The area of the rectangle is {} square pixels.",
        area(rect1)
    );
}

fn area(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}
```

[الشيفرة 9: تخزين طول وعرض المستطيل في صف]

البرنامج الحالي أفضل، إذ تسمح لنا الصفوف بتنظيم القيم على نحوٍ أفضل، إضافةً إلى أننا نمرّر للدالــة الآن معاملًا واحدًا، إلا أن هذا الإصدار أقل وضوحًا، لأن الصفوف لا تُسـمّي عناصـرها وبالتـالي علينـا اسـتخدام دليـل العنصر للوصول إلى القيمة التي نريدها مما يجعل من حساباتنا أقل وضوحًا.

الخلط بين الطول والعرض في حساب المساحة ليس بالأمر الجلل، إلا أن التمييز بين القيمتين يصبح مهمًّا في حال أردنا رسم المستطيل على الشاشة، وفي هذه الحالة علينا تذكّر أن قيمة العرض `width` مُخزّنة في دليل الصف "0" وقيمة العرض `height` مخزّنة في دليل الصف "1"، وقد يكون هذا الأمر صعبًا إذا أراد أحدٌ التعـديل على شيفرتنا البرمجية، أي أن شيفرتنا البرمجية لن تكون مُعبّرة وواضحة مما يرفع نسبة الأخطاء واردة الحدوث.

5.2.2 إعادة كتابة البرامج باستخدام الهياكل وبوضوح أكبر

نستخدم الهياكل حتى نجعل من بياناتنا ذات معنى بتسميتها، إذ يمكننا تحويل الصف المُستخدَم سابقًا إلى هيكل باسم له، إضافةً إلى اسمٍ لكل حقلٍ داخله كما هو موضح في الشيفرة 10.

اسم الملف: src/main.rs

```
struct Rectangle {
    width: u32,
    height: u32,
```

```
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

[الشيفرة 10: تعريف الهيكل Rectangle]

عرّفنا هنا هيكلًا اسمه `Rectangle`، ثم عرّفنا داخل الأقواس المعقوصة حقلَي الهيكل `width` و `height` من النوع `u32`، وننشئ بعدها نسخةً من الهيكل ضمن الدالة `main` بعرض 30 بيكسل وطول 50 بيكسل.

الدالة `area` في هذا الإصدار من البرنامج معرّفة بمعامل واحد، وقد سمّيناه `rectangle` وهو من نوع الهيكل `Rectangle` القابل للإستعارة دون تعديل، وكما ذكرنا سابقًا، يمكننا استعارة borrow الهيكل بدلًا من الحصول على ملكيته وبهذه الطريقة تحافظ الدالة `main` على ملكيته ويمكننا استخدامه عن طريق النسخة `rect1` وهذا هو السبب في استخدامنا للرمز `&` في بصمة الدالة وعند استدعائها.

تُستخدَم الدالة `area` في الوصول لحقلَي نسخة الهيكل `Rectangle` وهُما `width` و `height`، وتدلّ بصمة الدالة هنا على ما نريد فعله بوضوح: احسب مساحة `Rectangle` باستخدام قيمتَي الحقلين `width` و `height`، وهذا يدلّ قارئ الشيفرة البرمجية على أن القيمتين مترابطتين فيما بينهما ويُعطي اسمًا واصفًا واضحًا لكل من القيمتُين بدلًا من استخدام قيم دليل الصف "0" و"1" كما سبق. وهذا الإصدار الأوضح حتى اللحظة.

5.2.3 بعض الإضافات المفيدة باستخدام السمات المشتقة

سيكون من المفيد أن نطبع نسخةً من الهيكل `Rectangle` عند تشخيص أخطاء برنامجنا للتحقـق من قيم الحقول، نُحاول في الشيفرة 11 فعل ذلك باستخدام الماكرو println! كما عهدنا في الفصول السابقة إلا أن هـذا الأمر لن ينجح.

اسم الملف: src/main.rs

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {}", rect1);
}
```

[الشيفرة 11: محاولة طباعة نسخة من الهيكل Rectangle]

نحصل على رسالة الخطأ التالية عندما نُصرّف الشيفرة البرمجية السابقة:

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
```

يمكّننا الماكرو `println!` من تنسيق الخرج بعدّة أشكال، إلا أن التنسيق الافتراضي هو التنسيق المعروف باسم `Display` الذي يستخدم أسلوب كتابة الأقواس المعقوصة، وهو تنسيق موجّه لمستخدم البرنامج، وتستخدم أنواع البيانات الأولية primitive التي استعرضناها حتى الآن تنسيق `Display` افتراضيًا، وذلك لأن هناك طريقةً واحدةً لعرض القيمة "1" -أو أي قيمة نوع أولي آخر- للمستخدم، لكن الأمر مختلف مع الهياكل إذ أن هناك عدّة احتمالات لعرض البيانات التي بداخلها؛ هل تريد الطباعة مع الفواصل أم بدونها؟ هل تريد طباعة الأقواس المعقوصة؟ هل تريد عرض جميع الحقول؟ وبسبب هذا لا تحاول رست تخمين الطريقة التي نريد عرض الهيكل بها ولا يوجد أي تطبيق لطباعة الهيكل باستخدام النمط `Display` في الماكرو `println!` باستخدام الأقواس المعقوصة `{}`.

إذا قرأنا رسالة الخطأ نجد الملاحظة المفيدة التالية:

```
   = help: the trait `std::fmt::Display` is not implemented for
`Rectangle`
   = note: in format strings you may be able to use `{:?}` (or {:#?}
for pretty-print) instead
```

تخبرنا الملاحظة أنه يجب علينا استخدام تنسيق محدّد لطباعة الهيكل، لنجرّب ذلك! يصبح استدعاء الماكرو `println!` بالشكل التالي:

```
println!("rect1 is {:?}, rect1);
```

يدّل المحدد `:?` داخل الأقواس المعقوصة على أننا نريد استخدام تنسيق طباعة يدعى Debug، إذ يُمكّننا هذا النمط من طباعة الهيكل بطريقة مفيدة للمطورين عند تشخيص أخطاء الشيفرة البرمجية.

صرّف الشيفرة البرمجية باستخدام التغيير السابقة، ونفّذ البرنامج. حصلنا على خطأ من جديد.

```
error[E0277]: `Rectangle` doesn't implement `Debug`
```

إلا أن المصرف يساعدنا مجددًا بملاحظة مفيدة:

```
   = help: the trait `Debug` is not implemented for `Rectangle`
   = note: add `#[derive(Debug)]` to `Rectangle` or manually `impl
Debug for Rectangle`
```

لا تُضمّن رست إمكانية طباعة المعلومات المتعلقة بتشخيص الأخطاء، إذ عليك أن تحدّد صراحةً أنك تريد استخدام هذه الوظيفة ضمن الهيكل الذي تريد طباعته، ولفعل ذلك نُضيف السمة الخارجية `#[derive(Debug)]` قبل تعريف الهيكل كما هو موضح في الشيفرة 12.

اسم الملف: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
```

```
    println!("rect1 is {:?}", rect1);
}
```

[الشيفرة 12: إضافة سمة للهيكل للحصول على السمة المُشتقة Debug وطباعة نسخة من الهيكل Rectangle باستخدام تنسيق تشخيص الأخطاء]

لن نحصل على أي أخطاء أخرى عندما ننفذ البرنامج الآن، وسنحصل على الخرج التالي:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 is Rectangle { width: 30, height: 50 }
```

رائع، فعلى الرغم من أن تنسيق الخرج ليس جميلًا إلا أنه يعرض لنا جميع حقول النسخة، وهذا سيساعدنا بالتأكيد خلال عملية تشخيص الأخطاء. من المفيد استخدام المُحدد `{:#?}` بدلًا من المحدد `{:?}` عندما يكون لدينا هياكل ضخمة، ونحصل على الخرج بالطريقة التالية عند استخدام المحدد السابق:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 is Rectangle {
    width: 30,
    height: 50,
}
```

يمكننا طباعة القيم بطريقة أخرى، وهي باستخدام الماكرو dbg!، إذ يأخذ هذا الماكرو ملكية التعبير ويطبع الملف ورقم السطر الذي ورد فيه الماكرو، إضافةً إلى القيمة الناتجة من التعبير، ثمّ يُعيد الملكية للقيمة.

> يطبع استدعاء الماكرو `dbg!` الخرج إلى مجرى أخطاء الطرفية القياسي standard error console stream أو كما يُدعى "stderr" على عكس الماكرو `println!` الذي يطبع الخرج إلى مجرى خرج الطرفية القياسي standard output console stream أو كما يُدعى "stdout"، وسنتحدث بصورةٍ موسعة عن "stderr" و "stdout" لاحقًا.

إليك مثالًا عمّا سيبدو عليه برنامجنا إذا كُنّا مهتمين بمعرفة القيمة المُسندة إلى الحقل `width` إضافةً إلى قيم كامل الهيكل في النسخة `rect1`:

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let scale = 2;
    let rect1 = Rectangle {
        width: dbg!(30 * scale),
        height: 50,
    };

    dbg!(&rect1);
}
```

يُمكننا كتابة `dbg!` حول التعبير `30 * scale` وسيحصل الحقل `width` على القيمة ذاتها في حالة عـدم استخدامنا لاستدعاء `dbg!` هنا لأن `dbg!` يُعيد الملكية إلى قيمة التعبير، إلا أننا لا نريـد للمـاكرو `dbg!` أن يأخـذ ملكية `rect1`، لذلك سنستخدم مرجعًا إلى `rect1` في الاستدعاء الثاني. إليك ما سيبدو عليه الخرج عند تنفيـذ البرنامج:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.61s
     Running `target/debug/rectangles`
[src/main.rs:10] 30 * scale = 60
[src/main.rs:14] &rect1 = Rectangle {
    width: 60,
    height: 50,
}
```

يُمكننا ملاحظة أن الجزء الأول من الخرج طُبِعَ نتيجةً للسطر البرمجي العاشر ضمن الملف src/main.rs، إذ أردنا تشخيص الخطأ في التعبير `30 * scale` والقيمة 60 الناتجة عنه (يطبع تنسيق `Debug` فقط القيمة في حالة الأعداد الصحيحة)، بينما يطبع الاستدعاء الثاني للماكرو `dbg!` الوارد في السطر الرابع عشـر ضـمن الملـف src/main.rs قيمة `&rect1` الذي هو بدوره نسخةٌ من الهيكل `Rectangle`، ويستخدم الخرج تنسـيق الطباعـة

Debug ضمن النوع Rectangle. يُمكن للماكرو dbg! أن يكون مفيدًا في العديد من الحالات التي تريد فيها معرفة ما الذي تفعله شيفرتك البرمجية.

تزوّدنا رست بعدد من السمات الأخرى بجانب السمة Debug، ونستطيع استخدامها بواسطة السمة derive مما يُمكّننا من إضافة سلوكيات مفيدة إلى أنواع البيانات المُخصصة، نوضح السمات وسلوكياتها في الملحق (ت)، وسنغطي كيفية إنشاء سمات مُخصصة لاحقًا. هناك العديد من السمات الأخرى بجانب derive، للحصول على معلومات أكثر انظر هنا.

الدالة area الموجودة لدينا الآن ضيقة الاستخدام، إذ أنها تحسب فقط مساحة المستطيل ومن المفيد أن نربط سلوكها بصورةٍ أكبر مع الهيكل Rectangle لأنها لا تعمل مع أي نوع آخر، وسننظر لاحقًا إلى إصدارات أخرى من هذا البرنامج التي ستحوّل الدالة area إلى تابع area ضمن النوع Rectangle.

5.3 استخدام التوابع methods

التوابع methods مشابهة للدوال functions، إذ نُصرّح عنها باستخدام الكلمة المفتاحية fn متبوعةً باسم التابع، ويمكن للتوابع أن تمتلك عدّة معاملات وأن تُعيد قيمةً ما، ويحتوي التابع بداخله على جزء من شيفرة برمجية تعمل عند استدعاء التابع في مكان آخر، إلا أن التوابع -على عكس الدوال- تُعرّف داخل الهيكل (أو داخل المعدّد enum، أو كائن سمة trait object وهو ما سنتكلم عنه لاحقًا)، ويكون المعامل الأول دائمًا هو self الذي يمثّل نسخةً من الهيكل التي يُستدعى التابع منها.

5.3.1 تعريف التوابع

دعنا نُعدّل الدالة area التي تأخذ نسخةً من الهيكل Rectangle معاملًا لها، ونُنشئ بدلًا من ذلك تابع area معرّف داخل الهيكل Rectangle كما هو موضح في الشيفرة 13.

اسم الملف: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
```

```
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

[الشيفرة 13: تعريف التابع area داخل الهيكل Rectangle]

نبدأ بكتابة كتلة تطبيق implementation `impl` داخل الهيكل `Rectangle` حتى نستطيع تعريف الدالة `area` داخل سياق الهيكل، وسيكون كل شيء ضمن هذه الكتلة مرتبطًا بالنوع `Rectangle`، من ثمّ ننقل الدالة إلى داخل أقواس الكتلة `impl` ونعدّل المعامل الأول -وفي هذه الحالة هو المعامل الوحيد- ليصبح `self` في بصمة الدالة وأي مكان آخر ضمنها. ننتقل إلى الدالة `main` وعوضًا عن استدعاء الدالة `area` وتمرير `rect1` مثل وسيط، سنستخدم طريقة كتابة التابع لاستدعاء التابع `area` على نسخة الهيكل `Rectangle`، إذ تتلخّص الطريقة بإضافة نقطة (.) بعد نسخة الهيكل متبوعةً باسم التابع ومن ثم القوسين وبداخلهما أي وسطاء.

نستخدم `&self` في بصمة الدالة `area` عوضًا عن `rectangle: &Rectangle`، وفي الحقيقة `&self` هو اختصار إلى `self: &Self`، ويمثّل `Self` داخل الكتلة `impl` اسمًا مستعارًا للنوع الذي يحتوي داخله الكتلة `impl`،وهي في هذه الحالة `Rectangle`. يجب على التوابع أن تحتوي على وسيط باسم `self` من النوع `Self` مثل مُعامل أوّل، إذ تسمح لك رست باختصار الاسم إلى `self` في المعامل الأول للتابع، لاحظ أنّنا ما زلنا بحاجة الرمز `&` أمام الاسم المختصر `self` وذلك للإشارة إلى أن التابع يستعير نسخةً من النوع `Self` كما فعلنا سابقًا باستخدام `rectangle: &Rectangle`. يمكن للتوابع أن تمتلك الاسم `self` أو أن تستعير `self` على نحوٍ غير قابل للتعديل -كما فعلنا هنا- أو أن تستعير `self` مع إمكانية تعديل كما هو الحال في أي مُعامل آخر.

اخترنا `&self` هنا مثل معامل للسبب ذاته الذي دفعنا لاستخدام `&Rectangle` في إصدار البرنامج السابق، وهو أننا لا نريد أخذ الملكيّة بل نريد الاكتفاء فقط بقراءة البيانات الموجودة في الهيكل دون التعديل عليها. إذا أردنا تغيير النسخة التي استدعينا التابع عليها مثل جزء من وظيفة التابع، فيجب علينا استخدام `&mut self` مثل معامل أوّل بدلًا من ذلك. من النادر أن تجد تابعًا يأخذ ملكية نسخة ما باستخدام `self` فقط

في المعامل الأول وتُستخدم هذه الطريقة عادةً عندما يحوِّل التابع الوسيط `self` إلى شيء آخر وتريد أن تمنع مستدعي التابع من استخدام النسخة الأصل بعد عملية التحويل.

السبب الرئيسي في استخدامنا التوابع بدلًا من الدوال هو التنظيم، إضافةً إلى أننا لا نُكرّر نوع الوسيط `self` في كل بصمةٍ للتابع، إذ سمحت لنا التوابع بوضع جميع الأشياء التي يمكننا فعلها ضمن نسخة النوع `impl` داخل كتلة واحدة عوضًا عن إجبار قارئ الشيفرة البرمجية بالبحث عن الدوال التي تشير لإمكانيات الهيكل `Rectangle` بنفسه في أماكن مختلفة من المكتبة التي كتبناها.

لاحظ أنه يمكننا اختيار اسم التابع على نحوٍ مماثل لاسم حقول الهيكل، على سبيل المثال يمكننا تعريف تابع داخل الهيكل `Rectangle` باستخدام اسم الحقل `width`:

اسم الملف: src/main.rs

```
impl Rectangle {
    fn width(&self) -> bool {
        self.width > 0
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    if rect1.width() {
        println!("The rectangle has a nonzero width; it is {}",
rect1.width);
    }
}
```

نختار هنا أن نجعل التابع `width` يُعيد القيمة `true` إذا كانت القيمة `width` في نسخة الهيكل أكبر من 0، وإلا فالقيمة `false` إذا كانت القيمة مساوية إلى الصفر، ويمكننا الاستفادة من الحقل داخل التابع الذي يحمل الاسم ذاته لأي هدف كان، ثم ننتقل إلى الدالة `main` ونستخدم التابع بالشكل `rect1.width` مع الأقواس حتى تعلم رست أننا نقصد التابع `width`، إذ ستعلم رست أننا نقصد الحقل `width`، إذا لم نستخدم الأقواس.

قد نحتاج في بعض الأحيان عندما نُعطي التابع الاسم ذاته لحقل ما أن يُعيد التابع هذا القيمة الموجودة في الحقل ولا شيء آخر، وتُدعى التوابع من هذا النوع بالتوابع الجالبة getters ولا تُطبّقها رست تلقائيًا كما هو

الحال في معظم اللغات الأخرى. تُعد التوابع الجالبة مفيدة لأنها تُمكّنك من جعل الحقل خاصًّا private وجعل التابع عامًا public في ذات الوقت مما يمكنك من الوصول إلى الحقل وقراءته فقط بمثابة جزء من الواجهة البرمجية API العامة للنوع، وسنناقش معنى خاص وعام وكيفية جعل الحقل أو التابع خاصًا أو عامًا لاحقًا.

ا. أين العامل '->' ؟

لدى لغة سي C و C++ معاملان مختلفان لاستدعاء التوابع، إذ يُمكنك استخدام `.` إذا أردت استدعاء التابع على الكائن مباشرةً، أو استخدام المعامل `->` إذا أردت استدعاء التابع على مؤشر يُشير إلى الكائن وتريد أن تحصل dereference على المؤشر عن الكائن أولًا؛ أي بكلمات أخرى، إذا كان `object` مؤشرًا فكتابة `object->something()` مشابهة إلى `(*object).something()`.

لا يوجد في رست مُكافئ للمعامل `->`، بل لدى رست ميزة تُدعى بالمرجع لعنوان الذاكرة والتحصيل التلقائي automatic referencing and dereferencing بدلًا من ذلك، واستدعاء التوابع هو واحدة من الأجزاء في لغة رست التي تتبع هذا السلوك.

إليك كيفية عمل هذه الميزة: عند استدعاء التابع باستخدام `object.something()` تُضيف رست `&` أو `&mut` أو `*` تلقائيًا حتى يُطابق `object` بصمة التابع، أي بكلمات أخرى، السطرين البرمجيين متماثلين، إلا أن السطر البرمجي الأول يبدو أكثر ترتيبًا:

```
p1.distance(&p2);
(&p1).distance(&p2);
```

يعمل سلوك المرجع لعنوان الذاكرة التلقائي لأن للتوابع مستقبل receiver واضح ألا وهو النوع `self`، وتستطيع رست باستخدام المستقبل الواضح واسم التابع أن تعرف دون شك إذا ما كان التابع يقرأ (`&self`) أو يعدّل (`&mut self`) أو يستهلك (`self`)، وتُعدّ حقيقة أن رست تجعل من الاستعارة مباشرة لمستقبل التابع من أهم أجزاء ميزة الملكية في رست.

5.3.2 التوابع التي تحتوي على عدة معاملات

دعنا نتدرب على استخدام التوابع بتطبيق تابع ثاني ضمن الهيكل `Rectangle`، ونريد في هذه المرة أن تأخذ نسخةٌ من الهيكل `Rectangle` نسخةً أخرى من الهيكل ذاته وأن تُعيد `true` إذا كانت النسخة الثانية تتسع كاملةً داخل النسخة الأولى (`self`) وإلا فيجب أن تُعيد `false`، وسنستطيع كتابة الشيفرة 14 بعد تعريف التابع `can_hold`.

اسم الملف: src/main.rs

```
fn main() {
    let rect1 = Rectangle {
```

```
        width: 30,
        height: 50,
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

[الشيفرة 14: استخدام التابع can_hold الذي لم نكتبه بعد]

سيبدو خرج الشيفرة البرمجية السابقة كما يلي، وذلك لأن كلا أبعاد النسخة `rect2` أصغر من أبعاد النسخة `rect1` إلا أن أبعاد النسخة `rect3` أكبر من النسخة `rect1`:

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

نعلم أننا نريد تعريف تابع، ولذلك سنكتب ذلك ضمن الكتلة `impl Rectangle`، وسيكون اسم التابع `can_hold` وسيستعير نسخةً من `Rectangle` غير قابلة للتعديل مثل معامل، ويمكننا معرفة نوع المعامل بالنظر إلى السطر البرمجي الذي سيستدعي التابع، إذ يمرر الاستدعاء `rect1.can_hold(&rect2)` الوسيط `&rect2` وهو نُسخة من الهيكل `Rectangle` مُستعارة غير قابلة للتعديل، وهذا الأمر منطقي لأننا نريد فقط أن نقرأ بيانات النسخة `rect2` ولا حاجة لنا في التعديل عليها مما سيتطلب نسخةً مُستعارةً قابلة للتعديل، إذ نريد هنا أن تحتفظ الدالة `main` بملكية `rect2` حتى نستطيع استخدامها مجددًا بعد استدعاء التابع `can_hold`.

ستكون القيمة المُعادة من التابع `can_hold` بوليانية boolean وسيتحقق تطبيقنا فيما إذا كان الطول والعرض الخاص بالمعامل `self` أكبر من الطول والعرض الخاص بنسخة `Rectangle` الأخرى. دعنا نُضيف التابع `can_hold` الجديد إلى كتلة `impl` الموجودة في الشيفرة 13 كما هو موضح في الشيفرة 15.

اسم الملف: src/main.rs

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

[الشيفرة 15: تطبيق التابع can_hold ضمن الهيكل Rectangle الذي يأخذ نسخةً أخرى من الهيكل Rectangle بمثابة معامل]

سنحصل على الخرج المطلوب عند تشغيل الشيفرة البرمجية ضمن `main` في الشيفرة 14. يمكن أن تأخذ التوابع عدة معاملات إن أردنا وذلك بإضافتها إلى بصمة التابع بعد المعامل `self`، وتعمل هذه المعاملات كما تعمل في الدوال عادةً.

5.3.3 الدوال المترابطة

تُدعى جميع الدوال المُعرّفة داخل الكتلة `impl` بالدوال المترابطة associated functions، وذلك لأنها مرتبطة بالنوع الموجود بعد `impl`، ويمكننا تعريف الدوال المترابطة التي لا تحتوي على المعامل الأول `self` (وبالتالي فهي لا تُعدّ توابعًا)، لأننا لا نحتاج إلى نسخة من النوع عند تنفيذها، وقد استخدمنا دالةً مشابهةً لهذه سابقًا، ألا وهي الدالة `String::from` والمعرّفة داخل النوع `String`.

تُستخدم الدوال المترابطة التي لا تُعدّ بمثابة توابع في الباني constructor الذي يعيد نسخةً جديدةً من الهيكل، وتُسمى عادةً الدوال المترابطة `new` لكن هذا الاسم غير مخصص في هذه اللغة، إذ يمكننا مثلًا كتابة دالة مترابطة تحتوي على معامل من بُعدٍ واحد وأن نستخدم هذا المعامل للطول والعرض وبالتالي يُسهّل هذا الأمر إنشاء مربّع باستخدام الهيكل `Rectangle` بدلًا من تحديد القيمة ذاتها مرتين:

اسم الملف: src/main.rs

```
impl Rectangle {
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
```

```
    }
}
```

تُعد كلمات `Self` المفتاحية في النوع المُعاد ومتن الدالة بمثابة أسماء مستعارة للنوع الذي يظهر بعد الكلمة المفتاحية `impl`، والتي هي في حالتنا `Rectangle`. نستخدم `::` مع اسم الهيكل لاستدعاء الدالة المترابطة، والسطر `let sq = Rectangle::square(3)` هو مثال على ذلك، ويقع فضاء أسماء namespace هذه الدالة داخل الهيكل، ويُستخدم الرمز `::` لكلٍّ من الدوال المترابطة وفضاءات الأسماء المُنشأة من قبل الوحدات modules التي سنناقشها لاحقًا.

5.3.4 كتل impl متعددة

يمكن أن يحتوي كل هيكل على عدّة كُتَل `impl`، فعلى سبيل المثال الشيفرة 15 مكافئة للشيفرة 16 التالية التي تحتوي على كل تابع داخل كتلة `impl` مختلفة.

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

[الشيفرة 16: إعادة كتابة الشيفرة 15 باستخدام كتَل impl متعددة]

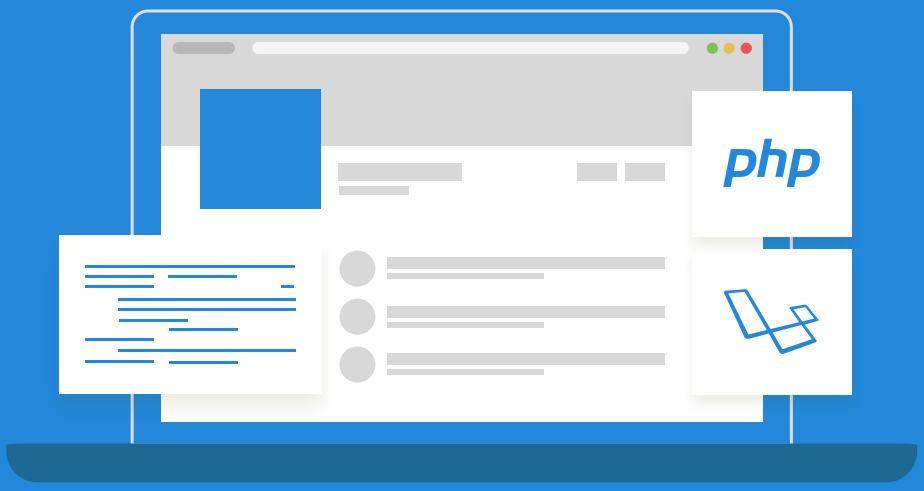
لا يوجد أي سبب لتفرقة التوابع ضمن كتل `impl` متعددة هنا، إلا أنها كتابة صحيحة وسنستعرض حالة قد تكون فيها هذه الكتابة مفيدة لاحقًا عندما نُناقش الأنواع المُعمّمة generic types والسمات.

5.4 خاتمة

تسمح لك الهياكل بإنشاء أنواع مُخصصة مفيدة لحالات استخدام برنامجك، ويمكنك باستخدام الهياكل المحافظة على بياناتك المتعلقة ببعضها مترابطةً فيما بينها وأن تُسمّي كل جزء من البيانات اسمًا معبّرًا مما ينعكس إيجابيًا على وضوح شيفرتك البرمجية. يمكنك تعريف الدوال المترابطة مع النوع داخل كتل `impl`؛ والتوابع هي نوع من الدوال المترابطة التي تسمح لك بتحديد السلوك الذي تريد من نسخة الهيكل أن تمتلكه.

ليست الهياكل الطريقة الوحيدة لإنشاء الأنواع المخصصة، وهذا ما ستتعرف إليه عندما نتكلم عن ميزة المعددات في رست بحيث تُضيف أداةً جديدةً أخرى إلى مخزونك.

دورة تطوير تطبيقات الويب
باستخدام لغة PHP



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

6. التعدادات Enums

سننظر في هذا الفصل إلى التعدادات enumerations -أو اختصارًا enums- وهي تسمح لنا بتعريف نـوع من خلال تعدّد متغيراته variants المحتملة. سنعرّف أولًا المعدّدات ونستخدمها حتى نعرف إمكانياتها وقـدرتها على ترميز البيانات، ثم سنستعرض التعداد المهم المسمى `option`، والذي يدل على قيمة إذا كان يحتـوي على شيء أو لا شيء، ومن ثمّ سـننظر إلى مطابقـة الأنمـاط pattern matching باسـتخدام التعبـير `match` الـذي يجعل من عملية تشغيل شيفرات برمجية مختلفة بحسب قيم التعداد المختلفة عملية سهلة، وأخيرًا سنتكلم عن الباني `if let` وهو طريقة ومصطلح مختصر آخر مُتاح لنا للتعامل مع التعدادات في برامجنا.

6.1 تعريف تعداد

التعدادات هي طريقـة لتعريـف أنـواع بيانـات مُخصصـة بصـورةٍ مختلفـة عن الهياكـل structs؛ إذ تمنحنـا الهياكل طريقةً لتجميع الحقول والبيانات ذات الصلة معًا، فمثلًا يُعرّف المستطيل `Rectangle` من خلال طولـه `height` وعرضه `width`؛ بينما تعطينا التعدادات الخيار للقـول أن القيمـة هي واحـدة من القيم الممكنـة، فقـد نرغب مثلًا بالقول أن المستطيل هو أحد الخيارات من مجموعة أشـكال متاحـة تتضـمن أيضًـا الـدائرة `Circle` والمثلث `Triangle`. يسمح لنا رست بترميز هذه الاحتمالات على هيئة تعداد.

دعنا ننظر إلى حالة قد تستخدم فيها التعدادات في شـيفرتك البرمجيـة ولننظـر إلى فائـدتها والأشـياء الـتي تميزهـا عن الهياكـل؛ لنقـل أن تطبيقنـا سـيتعامل مـع عنـاوين بروتوكـول الإنـترنت IP address. المعيـاران المُستخدمان حاليًا في عناوين بروتوكول الإنترنت هما الإصدار الرابع والإصدار السادس، لذلك نستطيع استخدام التعداد لجميع المتغيرات variants في الحالتين، وهذا هو السبب في تسمية التعدادات بهذا الاسم.

يمكن لأي عنوان أن يكون من الإصـدار الرابـع أو السـادس فقـط، وليس من الممكن أن يكـون العنـوان من الإصدارين معًا، وهذا ما يجعل استخدام التعداد مع عناوين الإنـترنت اسـتخدامًا مناسـبًا، لأن قيمتـه تكـون أحـد

متغيراته فقط، وكلّ من عناوين الإصدار الرابع والسادس هي عناوين إنترنت في نهاية المطاف ويجب أن تُعامل كأنها نوع واحد عندما تتعامل شيفرتنا البرمجية مع العناوين.

يمكننا التعبير عن هذا المفهوم في شيفرتنا البرمجية عن طريق تعريف التعداد `IpAddrKind` وإضافة أنواع عناوين الإنترنت الممكنة ألا وهي `V6` و `V4` والتي هي متغيرات التعداد:

```
enum IpAddrKind {
    V4,
    V6,
}
```

أصبح لدينا الآن التعداد `IpAddrKind` وهو نوع بيانات مخصص يمكننا استخدامه في أي مكان ضمن شيفرتنا البرمجية.

6.1.1 قيم التعداد

يُمكننا إنشاء نسخةٍ من كلا المتغيرَين في التعداد `IpAddrKind` على النحو التالي:

```
    let four = IpAddrKind::V4;
    let six = IpAddrKind::V6;
```

لاحظ أن فضاء أسماء متغيرات التعداد موجود ضمن المعرّف identifier، ويمكننا استخدام نقطتين مزدوجتين `::` لفصل كل من اسم المتغيّر والمعرف، وهذا الأمر مفيد لأن كلا القيمَتين `IpAddrKind::V4` و `IpAddrKind::V6` من نفس النوع وهو `IpAddrKind`. يمكننا بعد ذلك تعريف دالة تأخذ أي قيمة من النوع `IpAddrKind`:

```
fn route(ip_kind: IpAddrKind) {}
```

يمكننا بعدها استدعاء الدالة باستخدام أي من متغيّرات التعداد على النحو التالي:

```
    route(IpAddrKind::V4);
    route(IpAddrKind::V6);
```

لاستخدام التعدادات فوائد أكثر، فبالنظر إلى نوع عنوان الإنترنت الذي أنشأناه نلاحظ أنه لا توجد أي طريقة لنا لتخزين بيانات العنوان الفعلي، بل نستطيع فقط معرفة نوع العنوان، وقد تطبق ما تعلمناه عن الهياكل سابقًا بكتابة الشيفرة 1.

```
    enum IpAddrKind {
        V4,
```

```
        V6,
    }

    struct IpAddr {
        kind: IpAddrKind,
        address: String,
    }

    let home = IpAddr {
        kind: IpAddrKind::V4,
        address: String::from("127.0.0.1"),
    };

    let loopback = IpAddr {
        kind: IpAddrKind::V6,
        address: String::from("::1"),
    };
```

[الشيفرة 1: تخزين البيانات ومتغيّر النوع IpAddrKind باستخدام الهيكل struct]

عرفنا هنا هيكلًا `IpAddr` يحتوي على حقلين أحدهما `kind` من النـوع `IpAddrKind` (وهـو التعـداد الـذي عرفناه سابقًا) وحقل `address` من النوع `String`، ثم أنشأنا نسختين من هذا الهيكل، وأول نسـخة هي `home` وتحمـل القيمـة `IpAddrKind::V4` في الحقـل `kind` والقيمـة `127.0.0.1` في الحقـل `address`، وثـاني النسخ هي `loopback` وتحمل القيمة `IpAddrKind::V6` (المتغيّر الثاني من النوع `IpAddrKind`) في حقل `kind` ويحتوي على القيمة `::1` في حقل العنوان `address`، وبالتالي اسـتخدمنا هنـا هيكلًا لتجميـع القيمـتين `kind` و `address` مع بعضهما وربطنا المتغيّر مع القيمة.

يُعد تنفيذ المفهوم ذاته باستخدام التعدادات فقط أكـثر بسـاطة من ذلـك بكثـير، إذ بـدلًا من وجـود التعـداد `IpAddr` داخل الهيكل يمكننا وضع البيانات مباشرةً داخل كل متغيّر من التعداد. إليك التعريف الجديـد للتعـداد الذي يحتوي على `V4` و `V6` ويحمل كل منهما قيمةً من النوع `String`:

```
    enum IpAddr {
        V4(String),
        V6(String),
    }
```

```
    let home = IpAddr::V4(String::from("127.0.0.1"));

    let loopback = IpAddr::V6(String::from("::1"));
```

نربط البيانات إلى كل متغيّر من التعداد مباشرةً، مما يجنبنا إضافة هيكل إضافي كما يجعل من رؤية تفاصيل عمل التعداد عمليةً أكثر وضوحًا؛ إذ يصبح اسم كل متغيّر معرّف في التعداد دالةً تُنشئ نسخةً من التعداد أي أن `IpAddr::V4()` هو استدعاء لدالة تأخذ وسيطًا من النوع `String` وتُعيد نسخةً من النوع `IpAddr` ونحصل على هذه الدالة البانية constructor function تلقائيًا عند تعريف التعداد.

هناك ميزةٌ أخرى لاستخدام التعدادات عوضًا عن الهياكل، ألا وهي أن كل متغيّر يحصل على نوع مختلف وكمية مختلفة من البيانات المرتبطة به، إذ سيحصل نوع الإصدار الرابع من عنوان الإنترنت على أربع مكونات عددية تحمل قيمة تنتمي إلى المجال من 0 إلى 255. إذا أردنا تخزين عناوين الإصدار الرابع `V4` بأربع قيم من النوع `u8` مع المحافظة على إمكانية تمثيل عناوين الإصدار السادس `V6` مثل قيمة واحدة من النوع `String`، وهذا لن يكون هذا ممكنًا باستخدام الهياكل. إليك كيف تسمح لنا التعدادات بفعل ذلك:

```
    enum IpAddr {
        V4(u8, u8, u8, u8),
        V6(String),
    }

    let home = IpAddr::V4(127, 0, 0, 1);

    let loopback = IpAddr::V6(String::from("::1"));
```

استعرضنا عدة طرق في تعريف هياكل البيانات لتخزين الإصدار الرابع والسادس من عناوين بروتوكول الإنترنت، إلا أن تخزين عناوين الإنترنت في الحقيقة ممكن التخزين والترميز بسهولة عن طريق تعريف ضمن المكتبة القياسية. دعنا ننظر إلى كيفية تعريف المكتبة القياسية للنوع `IpAddr`، إذ تُعرّف المكتبة القياسية التعداد مع معدّداته بصورةٍ مماثلة لما فعلناه سابقًا إلا أنها تُضمّن عناوين الإنترنت داخل المتغيّرات ضمن هيكلين مختلفين مُعرّفَين على نحوٍ مختلف:

```
struct Ipv4Addr {
    // --snip--
}

struct Ipv6Addr {
    // --snip--
```

```
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

توضح لك الشيفرة البرمجية أنه بإمكانك تخزين أي نوع من البيانات داخل متغيّر التعداد مثل السلاسل النصية والأنواع العددية أو الهياكل، كما يمكنك أيضًا تضمين تعداد آخر داخلها. أنواع المكتبة القياسية بسيطة ويمكنك فهمها بسهولة أو كتابتها من الصفر لوحدك.

لاحظ أنه بالرغم من احتواء المكتبة القياسية على تعريف النوع `IpAddr`، إلا أنه ما زال بإمكاننا إنشاء واستخدام تعريفنا الخاص دون أي تعارض وذلك لأننا لم نُضِف تعريف المكتبة القياسية إلى نطاقنا، وسنتحدث على نحوٍ مفصل عن إضافة الأنواع إلى النطاق لاحقًا.

دعنا ننظر إلى مثال آخر على التعدادات في الشيفرة 2، إذ يحتوي هذا المثال على أنواع متعددة ضمن متغيّرات التعداد.

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

[الشيفرة 2: تعداد Message يحتوي على متغيّرات، يخزّن كل منها عدد ونوع مختلف من أنواع القيم]

يحتوي التعداد السابق على أربع متغيّرات من أنواع مختلفة:

- المتغيّر `Quit`، الذي لا يحتوي على أي بيانات مرتبطة به إطلاقًا.
- المتغيّر `Move`، الذي يحتوي على حقول مُسمّاة بصورةٍ مشابهة للهياكل.
- المتغيّر `Write`، الذي يحتوي على نوع `String` وحيد.
- المتغيّر `ChangeColor`، الذي يحتوي على ثلاث قيم من النوع `i32`.

عملية تعريف التعداد السابق في الشيفرة 2 هي عملية مشابهة لتعريف أنواع مختلفة من الهياكل، والفارق الوحيد هنا هو أن التعدادات لا تستخدم الكلمة المفتاحية `struct` وكل المتغيّرات هنا مجمعة مع بعضها بعضًا ضمن النوع `Message`. تُخزّن الهياكل التالية البيانات التي يحملها التعداد السابق:

```
struct QuitMessage; // unit هيكل وحدة
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // هيكل صف
struct ChangeColorMessage(i32, i32, i32); // هيكل صف
```

إلا أننا لن نصبح قادرين على تعريف دالة تأخذ أي من أنواع الرسائل بسهولة إذا استخدمنا هياكل مختلفة لكلّ منها نوعها المختلف كما هو الحال في التعداد `Message` المعرف في الشيفرة 2 الذي يمثّل نوعًا واحدًا فقط.

هناك تشابه آخر بين التعدادات والهياكل؛ إذ يمكننا تعريف توابع في التعدادات بطريقة مشابهة لما يحدث في الهياكل باستخدام `impl`. إليك تابعًا باسم `call` يمكننا تعريفه ضمن التعداد `Message`:

```
    impl Message {
        fn call(&self) {
            // يُعرّف محتوى التابع هنا
        }
    }

    let m = Message::Write(String::from("hello"));
    m.call();
```

تُستخدم `self` في متن التابع للحصول على القيمة المُعادة من استدعاء التابع، وفي هذا المثال نُنشئ متغيرًا `m` يحمل القيمة `Message::Write(String::from("hello"))`، وهذا ما ستكون قيمة `self` عليه داخل التابع `call` عند تنفيذ السطر `m.call()`.

دعنا ننظر إلى تعداد شائع مفيد آخر ضمن المكتبة القياسية ألا وهو `Option`.

6.1.2 التعداد Option وميزاته بما يخص القيم الفارغة

نستعرض في هذه الفقرة التعداد `Option`، وهو تعداد معرّف داخل المكتبة القياسية. يُستخدم النوع `Option` لترميز حالة شائعة عندما تكون القيمة شيئ أو لا شيء، فعلى سبيل المثال إذا طلبت أول عناصر القائمة ستحصل على قيمة (شيء) إلا إذا كانت القائمة فارغة فستحصل على لا شيء، والتعبير عن هذا المفهوم في سياق أنواع البيانات يعني أن المصرف قادر على معرفة إذا ما كنت تعاملت مع جميع الحالات التي يجب عليك التعامل معها، وتساعد هذه الميزة في منع حدوث الأخطاء الشائعة جدًا في باقي لغات البرمجة.

يُنظر إلى تصميم لغات البرمجة بسياق المزايا التي تتضمنها، إلا أن المزايا التي لن تُضمّن مهمةٌ أيضًا، إذ لا تحتوي رست على ميزة القيمة الفارغة null التي تمتلكها العديد من لغات البرمجة الأخرى؛ والقيمة الفارغة null تعني أنه لا يوجد أي قيمة هناك، وتكون المتغيرات في لغات البرمجة التي تحتوي على القيمة الفارغة في حالتين، إما حالة فارغة أو حالة غير فارغة not-null.

صرّح توني هواري Tony Hoare -مخترع القيمة الفارغة- في عام 2009 في عرض تقديمي بعنوان "المراجع الفارغة: خطأ بقيمة مليار دولار Null Reference: The Billion Dollar Mistake" ما يلي:

> أدعوها بخطئي الذي كلف مليار دولار. كنت في وقتها أصمم أول نظام للأنواع للمراجع في لغة كائنية التوجه، وكان هدفي التأكد من أن جميع استخدامات المراجع هي استخدامات آمنة بالكامل وذلك عن طريق فحص المصرف تلقائيًا، إلا أنني لم أستطع مقاومة إغراء إضافة مرجع فارغ null reference وذلك لأنه كان سهل التطبيق ببساطة. أدى ذلك فيما بعد إلى أخطاء وثغرات لا تُعد ولا تُحصى وتعطُّل للنظام، مما تسبب بخسائر بمليارات الدولار في الأربعين سنة المنصرمة.

المشكلة في القيم الفارغة هي أنك ستحصل على خطأ من نوع ما إذا حاولت استخدامها مثل قيمة غير فارغة، ومن السهل ارتكاب ذلك الخطأ لأن خاصية الفراغ وعدم الفراغ مستخدمة جدًا، إلا أن المفهوم الذي تحاول قيمة الفراغ أن تشير إليه ما زال مفيدًا ألا وهو أن القيمة الفارغة غير صالحة أو مفقودة لسبب ما.

المشكلة ليست بالمفهوم وإنما بالحل المُطبّق، ولهذا السبب لا تحتوي رست على النوع فارغ null، وإنما تحتوي على تعداد يدل على مفهوم عدم وجود قيمة أو عدم صلاحيتها ألا وهو التعداد `Option<T>` المعرف في المكتبة القياسية كما يلي:

```
enum Option<T> {
    None,
    Some(T),
}
```

التعداد `Option<T>` مفيد جدًا حتى أنه مضمّن في مقدمة البرنامج وليس عليك أن تُضيفه إلى النطاق يدويًا، كما أن متغيّراته مضمّنة أيضًا ويمكنك استخدام `Some` و `None` مباشرةً دون البدء بالبادئة `Option::`، إلا أن التعداد `Option<T>` هو تعداد مثل أي تعداد آخر و `Some(T)` و `None` هي متغيّرات من النوع `Option<T>`.

طريقة كتابة `<T>` هي ميزة من ميزات رست التي لم نتحدث عنها بعد، وهي معامل نوع مُعمّم generic type parameter وسنتكلم عن الأنواع المعممة لاحقًا، وكل ما عليك معرفته الآن هو أن `<T>` يعني أن متغيّر `Some` من التعداد `Option` يستطيع تخزين جزء من أي نوع من البيانات وأي نوع يُستخدم في مكان `T` يجعل من النوع `Option<T>` نوعًا مختلفًا. إليك بعض الأمثلة التي تستخدم قيم `Option` لتخزين عدة أنواع من البيانات:

```
    let some_number = Some(5);
     let some_char = Some('e');

     let absent_number: Option<i32> = None;
```

نوع some_number هو Option، ونوع some_string هو Option<char> وهو نوع مختلف تمامًا عن سابقه، وتستطيع رست تحديد هذه الأنواع بسبب استخدامنا للقيمة داخل متغيّر Some، إلا أن رست تتطلب منا تحديد نوع Option الكلي بالنسبة للمتغير absent_number ولا يستطيع المصرّف تحديد النوع الخاص بالمتغير Some بنفسه عن طريق النظر إلى قيمة None فقط، وعلينا إخبار رست هنا أننا نقصد أن absent_number هو من النوع Option<i32>.

نعلم أن هناك قيمة موجودة عندما يكون لدينا قيمة في Some، أما عندما يكون لدينا قيمة في None نعلم أن هذا الأمر مكافئ للقيمة الفارغة null أي أنه لا يوجد لدينا قيمة صالحة. إذًا، لمَ وجود Option<T> هو أفضل من وجود القيمة الفارغة؟

لأن Option<T> و T (إذ يمكن أن تدل T على أي نوع) هي من أنواع مختلفة ولن يسمح لنا المصرف باستخدام القيمة Option<T> على أنها قيمة صالحة. على سبيل المثال، لن تُصرَّف الشيفرة البرمجية التالية لأننا نحاول إضافة النوع i8 إلى نوع Option<i8>:

```
    let x: i8 = 5;
    let y: Option<i8> = Some(5);

    let sum = x + y;
```

إذا نفذنا الشيفرة البرمجية السابقة، نحصل على رسالة الخطأ التالية:

```
$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0277]: cannot add `Option<i8>` to `i8`
 --> src/main.rs:5:17
  |
|     let sum = x + y;
  |                 ^ no implementation for `i8 + Option<i8>`
  |
  = help: the trait `Add<Option<i8>>` is not implemented for `i8`
  = help: the following other types implement trait `Add<Rhs>`:
            <&'a f32 as Add<f32>>
```

```
              <&'a f64 as Add<f64>>
              <&'a i128 as Add<i128>>
              <&'a i16 as Add<i16>>
              <&'a i32 as Add<i32>>
              <&'a i64 as Add<i64>>
              <&'a i8 as Add<i8>>
              <&'a isize as Add<isize>>
            and 48 others

For more information about this error, try `rustc --explain E0277`.
error: could not compile `enums` due to previous error
```

تعني رسالة الخطأ السابقة أن رست لا تعرف كيفية إضافة `i8` إلى `Option<i8>` لأنهما من نوعين مختلفين. يتأكد المصرف عندما نحصل على قيمة من نوع مشابه إلى النوع `i8` في رست من أنه لدينا قيمة صالحة، ويمكننا تجاوز عملية التحقق بأمان دون الحاجة للتحقق من قيمة فارغة قبل استخدام هذه القيمة، إلا أنه يجب علينا أخذ الحيطة فقط في حال كان لدينا `Option<i8>` (أو أي نوع من قيمة نتعامل معها) وذلك إذا كان النوع لا يحمل أي قيمة وسيتأكد المصرف حينها من تعاملنا مع هذه الحالة قبل استخدام القيمة.

بكلمات أخرى، يجب عليك تحويل النوع `Option<T>` إلى `T` قبل تنفيذ عمليات النوع `T` على القيمة، ويساعدنا هذا عمومًا على تشخيص أحد أكثر المشاكل شيوعًا في القيم الفارغة ألا وهي افتراض أن قيمة ما ليست فارغة وهي في الحقيقة فارغة.

سيجعلك التخلص من المشكلة السابقة أكثر ثقة بشيفرتك البرمجية، إذ يتوجب عليك تحويل القيمة إلى النوع `Option<T>` يدويًا إذا أردت الحصول على القيمة التي من الممكن أن تكون فارغة، وعليك التعامل مع حالة كون القيمة فارغة إذا استخدمت هذه الطريقة، وتستطيع الافتراض بأمان أن أي قيمة ليست من النوع `Option<T>` ليست بقيمة فارغة، وهذا تصميم مقصود في لغة رست للحدّ من انتشار القيمة الفارغة وزيادة أمان شيفرة رست البرمجية.

كيف تُخرج القيمة `T` خارج متغيّر `Some` عندما يكون لديك قيمة من النوع `Option<T>` وتريد استخدام القيمة؟ لدى التعداد `Option<T>` العديد من التوابع المفيدة في حالات متعددة ويمكنك النظر إليها من التوثيق، وستكون معرفة هذه التوابع الخاصة بالنوع `Option<T>` مفيدةً جدًا في رحلتك مع رست.

يجب عليك كتابة شيفرة برمجية تتعامل مع كل متغيّر إذا أردت استخدام القيمة الموجودة في `Option<T>`، إذ يجب على شيفرتك البرمجية أن تُنفّذ فقط في حال كان داخل `Option<T>` قيمةً ما، والسماح لهذه الشيفرة البرمجية باستخدام القيمة `T` الداخلية، كما ينبغي وجود شيفرة برمجية أخرى تُنفّذ في حال كان هناك قيمة `None` بحيث لا تستخدم هذه الشيفرة البرمجية القيمة `T`. يسمح لنا تعبير `match` الذي

يمثل باني للتحكم بسير البرنامج باستخدام التعدادات بتنفيذ هذا الأمر تمامًا، إذ سينفذ شيفرةً برمجيةً مختلفةً بحسب المتغيّر الموجود داخل التعداد، ويمكن للشيفرة البرمجية حينها استخدام هذه البيانات داخل القيمة الموافقة.

6.2 بنية match للتحكم بسير البرنامج

لدى رست بنية construct فعالة جدًا للتحكم بسير البرنامج وتدعى ببنية `match`، إذ تسمح لك هذه البنية بمقارنة قيمة مع مجموعة من الأنماط، ثم تنفيذ شيفرة برمجية بناءً على النمط الموافق لهذه القيمة، ويمكن أن يكون النمط مشكلًا من قيمًا مجرّدة literal values، أو أسماء متغيرات، أو محرف بدل wildcard، أو أي شيء آخر، وسنتكلم لاحقًا عن جميع الأنواع المختلفة من الأنماط وعمل كل منها. تأتي قوة البنية `match` من قابلية التعبير الواضح عن الأنماط ومن أن المتصرف يتحقق من أن جميع الحالات الممكنة قد جرى التعامل معها.

انظر إلى تعبير البنية `match` بكونه آلةً لترتيب القطع النقدية المعدنية، إذ تدخل القطعة النقدية إلى مسار يحتوي على ثقوب متفاوتة الأحجام بحيث تسقط كل قطعة نقدية داخل الثقب الذي تتسع داخله، تدخل المتغيرات بالطريقة ذاتها للقطع النقدية ويجري فحصها بالنسبة لكل نمط موجود في البنية `match` و"يدخل" المتغير إلى أول كتلة برمجية يوافق نمطها مع المتغير حتى يُستخدم المتغير داخل تلك الكتلة البرمجية، وبما أننا اتخذنا القطع النقدية مثالًا للشرح دعنا نطّبق ذلك في برنامجنا باستخدام البنية `match`، إذ يُمكننا كتابة دالة تأخذ عددًا غير معين من القطع النقدية الأمريكية بطريقة مماثلة لآلة عد النقود لتحديد قيمة كل قطعة نقدية ومن ثم تُعيد قيمتها بالسنت cent، وتوضح الشيفرة 3 هذا البرنامج.

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

[الشيفرة 3: تعداد enum وتعبير match يحتوي على جميع متغايرات التعداد مثل أنماط]

دعنا نشرح التعبير `match` بالتفصيل في الدالة `value_in_cents`؛إذ يبدأ أولًا بكتابة الكلمة المفتاحية `match` متبوعة بتعبير وهو قيمة القطعة النقدية `coin` في هذه الحالة، ويبدو هذا الأمر مشابهًا لاستخدام تعابير `if` إلا أن هناك فارقًا كبيرًا؛ إذ يحتاج تعبير `if` أن يُعيد قيمة بوليانية boolean إلا أن التعبير هنا يمكن أن يُعيد قيمةً من أي نوع، فعلى سبيل المثال نحصل هنا على القيمة `coin` من نوع التعداد `Coin` الذي عرفناه في السطر الأول.

ننتقل الآن إلى أذرع arms البنية `match`، إذ يكون للذراع جزءان: نمط وشيفرة برمجية. يحتوي الذراع الأول هنا على نمط من النوع `Coin::Penny` ومن ثم العامل `=>` الذي يفصل ما بين النمط والشيفرة البرمجية الواجب تنفيذها، والشيفرة البرمجية في هذه الحالة هي فقط القيمة "1"، ومن ثم يُفصل كل ذراع من الذراع الذي يليه باستخدام الفاصلة.

تُقارن نتيجة التعبير `match` عند تنفيذه مع نمط كل ذراع بالترتيب؛ فإذا كان النمط يوافق القيمة تُنفّذ الشيفرة البرمجية المرتبطة بذلك النمط؛ وإذا لم يوافق القيمة يستمر تنفيذ البرنامج ليفحص الذراع التالية كما هو الحال في آلة عد القطع النقدية. يمكننا إضافة أذرع أخرى بقدر ما نريد، إذ تحتوي `match` في الشيفرة 3 على أربعة أذرع.

الشيفرة البرمجية المرتبطة بكل ذراع هي تعبير، ونتيجة ذلك التعبير في الذراع الموافقة للقيمة هي القيمة التي تُعاد من تعبير `match` الكامل.

لا تُستخدم عادةً الأقواس المعقوصة curly brackets، إذا كانت الشيفرة البرمجية في الذراع قصيرة كما هو الحال في الشيفرة 3، إذ تُعيد الشيفرة قيمة واحدة فقط مباشرةً، إلا أنه يجب علينا استخدام الأقواس المعقوصة إذا أردنا تنفيذ عدة أسطر برمجية داخل ذراع البنية `match`، ويكون استخدام الفاصلة بعد الذراع عندئذٍ اختياريًا. على سبيل المثال، تطبع الشيفرة البرمجية في المثال التالي "Lucky penny!" كل مرة يُستدعى فيها التابع باستخدام القيمة `Coin::Penny` إلا أنها ما زالت تُعيد أيضًا القيمة "1" في نهاية الكتلة البرمجية:

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        }
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
```

```
}
```

6.2.1 الأنماط المرتبطة مع القيم

لذراع البنية `match` ميزةٌ مفيدةٌ أخرى، وهي قدرة ربط أجزاء من القيمة لتوافق النمط، وهذه هي الطريقة المتبعة عندما نريد الحصول على قيم من متغايرات التعدادات. دعنا نعدل مثلًا متغايرات التعداد السابق بحيث نستطيع تخزين بعض البيانات داخله. سكّت الولايات المتحدة من العام 1999 إلى 2008 قطعًا نقدية بقيمة ربع دولار بتصاميم مختلفة لكلٍ من الولايات الخمسين على أحد الجوانب، ولا يوجد أي قطع نقدية أخرى تحتوي على تصاميم خاصة بالولايات. إذًا، نحتاج وضع القيمة الإضافية فقط في الأرباع، ويمكننا إضافة هذه المعلومة داخل التعداد `enum` بتغيير المتغاير `Quarter` بحيث يحتوي على القيمة `UsState` مخزنةً بداخله وتلك العملية موضحة في الشيفرة 4.

```
#[derive(Debug)] // حتى نستطيع معاينة الولاية لاحقًا
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

[الشيفرة 4: تعداد Coin يحتوي فيه المتغاير Quarter على القيمة UsState]

دعنا نتخيل أن صديقًا من أصدقائك يحاول جمع الأرباع الخمسين جميعها، ولمساعدته نستدعي اسم الولاية المرتبطة بكل ربع عند ترتيب القطع النقدية، بحيث يضيف صديقك ربعًا جديدًا إلى مجموعته إذا صادفنا ربعًا يعود لولاية ما لا يمتلكها.

نُضيف في تعبير `match` ضمن هذه الشيفرة البرمجية متغيرًا يدعى `state` إلى النمط الذي يطابق قيمة المتغاير `Coin::Quarter` وعندما تتطابق القيمة مع النوع `Coin::Quarter` يُربَط المتغير `state` مع قيمة ولاية الربع، ثم يصبح بإمكاننا استخدام `state` في الشيفرة البرمجية ضمن الذراع الخاصة بها كما يلي:

```
fn value_in_cents(coin: Coin) -> u8 {
```

```
        match coin {
            Coin::Penny => 1,
            Coin::Nickel => 5,
            Coin::Dime => 10,
            Coin::Quarter(state) => {
                println!("State quarter from {:?}!", state);
                25
            }
        }
    }
```

سـيأخذ المتغـير `coin` القيمـة `Coin::Quarter(UsState::Alaska)` إذا اسـتدعينا `value_in_cents(Coin::Quarter(UsState::Alaska))` وعنـدما نقـارن تلـك القيمـة مـع كـل من أذرع البنية `match` لن يُطابق أي منهم القيمة وسنصل إلى `Coin::Quarter(state)` وبحلول تلـك النقطـة سيكون ربط القيمـة `state` مـع النـوع `UsState::Alaska` ومن ثم يمكننـا اسـتخدام ذلـك الربـط في تعبـير `println!`، مما سيسمح لنا بالحصول على قيمة الولاية الداخلية من متغاير `Quarter` ضمن التعداد `Coin`.

6.2.2 المطابقة مع Option<T>

أردنا سابقًا الحصول على القيمة `T` الداخلية من الحالة `Some` عند استخدام `Option<T>`، إلا أنه يمكننا أيضًا التعامل مع `Option<T>` باستخدام `match` كمـا فعلنـا في معـدد `Coin`؛ إذ سـنقارن متغـايرات `Option<T>` عوضًا عن مقارنة العملات، إلا أن طريقة عمل تعبير `match` ستبقى كما هي.

دعنا نقول أننا نريد كتابة دالة تأخذ `Option<i32>` مثل وسيط وتُضيف إلى قيمته "1" إذا كان هناك قيمةٌ داخله، وإذا لم يحتوي الوسيط على قيمة يجب أن تُعيد الدالة القيمة `None` وألّا تُجري أي عمليات أخرى.

الدالة سهلة الكتابة جدًا وذلك بفضل `match` وستبدو كما هو موضح في الشيفرة 5.

```
        fn plus_one(x: Option<i32>) -> Option<i32> {
            match x {
                None => None,
                Some(i) => Some(i + 1),
            }
        }

        let five = Some(5);
        let six = plus_one(five);
```

```
    let none = plus_one(None);
```

[الشيفرة 5: دالة تستخدم تعبير match على المتغير Option]

دعنا نفحص التنفيذ الأول للدالة `plus_one` بالتفصيل، إذ يأخذ المتغير `x` الموجود داخل دالـة `plus_one` القيمة `Some(5)` عند الاستدعاء `plus_one(five)` ومن ثم نقارن تلك القيمة مع كل ذراع ضمن `match`.

```
            None => None,
```

لا تُطابق القيمة `Some(5)` النمط الأول `None` التالي، لذا نستمر بمحاولة النمط الذي يليه.

```
            Some(i) => Some(i + 1),
```

هل يُطابق `Some(5)` النمط `Some(i)`؟ نعم. لدينا المتغـاير ذاتـه وترتبـط `i` بالقيمـة الـتي تحتويهـا `Some` وبالتالي يأخذ `i` القيمة 5. تُنفّذ الشيفرة البرمجية الموجودة في ذراع `match` الموافقة، وبالتالي نُضـيف "1" إلى قيمة `i` ونُنشأ قيمة `Some` جديدة باستخدام الناتج "6" الإجمالي داخله.

دعنا ننظر إلى الاستدعاء الثاني للدالة `plus_one` في الشيفرة 5، إذ يأخـذ المتغـير `x` القيمـة `None`. نـدخل البنية `match` ونُقارن مع الذراع الأول.

```
            None => None,
```

حصلنا على مطابقة. ليس هنـاك أي قيمـة لنضـيفها لـذا يتوقـف البرنـامج ويُعيـد القيمـة `None` على يمين المعامل `=>` ولا تحدث أي مقارنة أخرى لأننا حصلنا على مطابقة مع الذراع الأولى.

استخدام البنية `match` مع التعدادات مفيد في العديد من الحالات، وستجد نمـط اسـتخدام البنيـة `match` مع التعداد شائعًا في لغة رست، إذ يُربط متغير إلى قيمة داخـل `match`، ثم تُنفّـذ الشـيفرة البرمجيـة الموافقـة، وقد يكون هذا الأسلوب معقدًا بعض الشيء إلا أنك ستتمنى لو أنه موجود في كـل لغـات البرمجـة حالمـا تعتـاد عليه إذ أنه الخيار المفضل لكثير من مبرمجي لغة رست.

6.2.3 يجب أن تكون بنى match شاملة

هناك جانب آخر من البنية `match` لم نناقشه بعد. ألقِ نظرةً على الإصدار التالي من دالـة `plus_one` الـذي يحتوي على خطأ برمجي ولن يُصرّف:

```
    fn plus_one(x: Option<i32>) -> Option<i32> {
        match x {
            Some(i) => Some(i + 1),
        }
```

```
    }
```

لم نشمل حالة `None` في بنية `match` السابقة وبالتالي ستتسبب شيفرتنا السابقة بخطأ برمجي، إلا أن رست تنبهنا على هذا الخطأ لحسن الحظ، إذ أنك ستحصل على رسالة الخطأ التالية إذا حاولت تصريف الشيفرة البرمجية:

```
$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0004]: non-exhaustive patterns: `None` not covered
   --> src/main.rs:3:15
    |
|         match x {
    |               ^ pattern `None` not covered
    |
note: `Option<i32>` defined here
    = note: the matched value is of type `Option<i32>`
help: ensure that all possible cases are being handled by adding a
match arm with a wildcard pattern or an explicit pattern as shown
    |
~             Some(i) => Some(i + 1),
~             None => todo!(),
    |

For more information about this error, try `rustc --explain E0004`.
error: could not compile `enums` due to previous error
```

تعلم رست أننا لم نغطي جميع الحالات الممكنة وحتى أنها تعرف الأنماط التي نسيناها؛ إذ أن الأنماط في لغة رست شاملة بحيث أنها يجب أن تشمل أي حالة ممكنة حتى تكون الشيفرة البرمجية صالحة، وبالأخص في حالة `Option<T>`، إذ تذكرنا رست بالتعامل مع حالة `None` صراحةً، وتحمينا من افتراض أننا حصلنا على قيمة بينما نحن نملك قيمة فارغة في الحقيقة مما يجعل الخطأ ذو مليارات الدولارات الذي ناقشناه لاحقًا مستحيل الحدوث.

6.2.4 التعامل مع جميع الأنماط والمحرف _ المؤقت

يمكننا أن نحدد تنفيذ بعض الأمور المميزة لبعض القيم في التعدادات بينما تُنفَّذ شيفرةٌ برمجيةٌ محددة لجميع القيم الممكنة الأخرى. تخيل أننا نبرمج لعبة نرمي فيها النرد وعندما نحصل على القيمة 3 لا يتحرك اللاعب بل يحصل على قبعة جميلة جديدة، بينما إذا حصلت على 7 فإن اللاعب يفقد تلك القبعة الجميلة،

ويتحرك اللاعب بباقي الحالات وفقًا للقيمة التي حصلنا عليها من النرد على الرقعة. نحاول تطبيق منطق اللعبة السابقة باستخدام البنية `match` بحيث يكون نتيجة النرد مكتوبة صراحةً عوضًا عن الحصول عليها عشوائيًا، ونمثل جميع الدوال الأخرى دون متن لها، وذلك لأن التطبيق الفعلي لكل منها خارج نطاق نقاشنا هنا:

```
    let dice_roll = 9;
    match dice_roll {
=> add_fancy_hat(),
=> remove_fancy_hat(),
        other => move_player(other),
    }

    fn add_fancy_hat() {}
    fn remove_fancy_hat() {}
    fn move_player(num_spaces: u8) {}
```

نستخدم قيمًا مجرّدة مثل نمط لكل من الذراعين الأوليين، بينما نستخدم نمطًا للذراع الأخيرة يغطّي كل النتائج الأخرى المحتملة؛ والنمط هو اسم المتغير الذي اخترناه ألا وهو `other`، إذ ستستخدم الشيفرة البرمجية المتغيّر `other` وتمرره إلى الدالة `move_player`.

تُصرَّف الشيفرة البرمجية السابقة بنجاح على الرغم من أننا لم نُضيف جميع القيم الممكنة التي يمكن تمثيلها باستخدام النوع `u8` وذلك لأن النمط الأخير سيطابق أي قيمة لم تُدرج صراحةً قبله، وهذا النمط يحقق شرط البنية `match` في كونها شاملة على جميع القيم المحتملة. لاحظ أنه يجب علينا إضافة ذراعًا آخرًا لمعالجة جميع الحالات catch-all، لأن الأنماط تُقيَّم بالترتيب، وستحذرنا رست إذا أضفنا أي ذراع آخر بعد ذراع معالجة جميع الحالات، وذلك لأن هذا الذراع الإضافي لن يُطابق إطلاقًا.

لدى رست نمطٌ يمكننا استخدامه لاستخدام القيمة التي نحصل عليها من نمط الحصول على جميع الحالات، ألا وهو النمط `_` المميز الذي يطابق أي قيمة ولا يُربط مع القيمة. تعلم رست عند استخدامه أننا لن نستخدم القيمة، لذا لن تحذرنا بخصوص المتغير غير المُستعمل.

دعنا نغيّر من قوانين اللعبة بحيث يجب عليك إعادة رمي النرد مجددًا إذا حصلت على نتيجة مغايرة عن 3 أو 7، ولا نحتاج هنا لاستخدام القيم صراحةً وكل ما علينا هو وضع المحرف المميز `_` بدلًا من اسم المتغيّر `other`:

```
    let dice_roll = 9;
    match dice_roll {
=> add_fancy_hat(),
=> remove_fancy_hat(),
```

```
            _ => reroll(),
        }

        fn add_fancy_hat() {}
        fn remove_fancy_hat() {}
        fn reroll() {}
```

تحقق الشيفرة البرمجية السابقة شرط بنية `match`، الذي ينص على وجوب شموليتها، إذ أننا نتجاهل جميع القيم الأخرى في الذراع الأخير مما يعني أننا لم ننسى أي احتمال.

يمكننا استخدام قيمة الواحدة unit value (نوع الصف الفارغ empty tuple type الذي ذكرناه سابقًا) مع ذراع المحرف المميز _، إذا أردنا تغيير قوانين اللعبة مجددًا بحيث لا يحصل أي شيء عندما تحصل على قيمة مغايرة عن القيمة 3 أو 7 :

```
        let dice_roll = 9;
        match dice_roll {
    => add_fancy_hat(),
    => remove_fancy_hat(),
            _ => (),
        }

        fn add_fancy_hat() {}
        fn remove_fancy_hat() {}
```

نخبر رست هنا صراحةً أننا لن نستخدم أي قيمة أخرى لا تتوافق مع النمطين السابقين للذراع الأخيرة، وأننا لا نريد تنفيذ أي شيفرة برمجية في هذه الحالة أيضًا.

هناك المزيد من الأنماط التي سنغطيها لاحقًا، إلا أننا سنتحدث الآن عن `if let` التي قد تكون مفيدةً في حالات يكون فيها استخدام التعبير `match` يتطلب صياغةً طويلة.

6.3 التحكم بسير البرنامج باستخدام if let

تسمح لك `if let` بجمع كل من `if` و `let` بطريقة أكثر اختصارًا مقارنةً باستخدام الأنماط و `match` مع تجاهل القيم الأخرى التي لا تهمنا. ألقِ نظرةً على البرنامج الموجود في الشيفرة 6 الذي يُطابق قيمة `Option<u8>` في المتغير `config_max` بحثًا عن قيمة واحدة ألا وهي متغاير `Some`.

```
        let config_max = Some(3u8);
        match config_max {
```

```
        Some(max) => println!("The maximum is configured to be {}",
max),
        _ => (),
    }
```

[الشيفرة 6: بنية match تنفذ شيفرة برمجية فقط في حالة الحصول على القيمة Some]

إذا كانت القيمة هي Some، نطبع القيمة داخل المتغاير Some من خلال ربطها مع المتغير max في النمـط، إلا أننا لا نريد فعـل أي شـيء إذا حصـلنا على القيمـة None ولتحقيـق شـرط البنيـة match بكونهـا تشـمل كـل الاحتمالات نُضيف _ => () بعد معالجة متغاير واحد، وهذا الأسلوب في الكتابة مزعج وهناك بديل أفضل.

بدلًا مما سبق، نستطيع اختصار الكتابة عن طريق اسـتخدام if let، وتوضـح الشـيفرة البرمجيـة التاليـة استخدامها، إذ تؤدي الغرض ذاته مقارنةً بالشيفرة السابقة الموجودة في الشيفرة 6 باستخدام match:

```
    let config_max = Some(3u8);
    if let Some(max) = config_max {
        println!("The maximum is configured to be {}", max);
    }
```

نستخدم مع if let نمطًا وتعبيرًا مفصولَين بإشارة مساواة، ويعمل هذا بطريقة مماثلة للبنية match، إذ يُعطى التعبير إلى match ويمثّل النمط الذراع الأولى له، وفي هذه الحالة هو Some(max)، وبالتالي تُربط قيمـة المتغير max إلى القيمة الموجودة داخل Some، ويمكننا بعد ذلك اسـتخدام max داخـل كتلـة if let بطريقـة مماثلة لاستخدامنا max في ذراع البنية match الموافقة، ولن تُنفَّذ كتلة if let إذا لم تطـابق القيمـة النمـط الخاص بها.

نقلّل -باستخدامنا if let- من كتابة السطور البرمجية ومحاذاتها ونتفـادى كتابـة سـطور برمجيـة نمطيـة مكرّرة، إلا أننا نفقد ميزة التفقد من شمولية الحالات كما هو الحال في بنية match، إذ يعتمـد اختيـار match أو if let على الشيء الذي تريد تنفيذه باستخدامهما والحالة الموجودة أمامنا، بحيث نقايض الكتابة المختصـرة بإمكانية التأكد من شمولية الحالات الموجودة.

بكلمات أخرى، يمكنك النظر إلى if let بكونها طريقـةً أنيقـةً لكتابـة البنيـة match الـتي تنفّـذ الشـيفرة البرمجية عندما تتطابق قيمة مع نمط واحد محدّد ومن ثم تتجاهل جميع القيم الأخرى.

يمكننا تضمين else مع if let، إذ أن كتلة else مطابقة لحالة _ في تعبير match المساوي لكـل من if let و else. تذكّر مثال تعريف التعداد Coin في الشيفرة 4، إذ كان متغـاير Quarter يحمـل قيمـةً من النوع UsState، ونستطيع استخدام تعبير match عندها إذا أردنا عدّ جميع الأرباع الـتي رأيناهـا بينمـا نُعلن عن ولاية كل من الأرباع كما يلي:

```
    let mut count = 0;
    match coin {
        Coin::Quarter(state) => println!("State quarter from {:?}!",
state),
        _ => count += 1,
    }
```

أو يمكننا استخدام تعبير `if let` و `else` بدلًا من ذلك كما يلي:

```
    let mut count = 0;
    if let Coin::Quarter(state) = coin {
        println!("State quarter from {:?}!", state);
    } else {
        count += 1;
    }
```

إذًا، إذا صادفت موقفًا احتاج فيه برنامجك إلى المنطق الخاص بالبنية `match`، وكانت كتابة البنية طويلة ورتيبة تذكّر وجود `if let` في مخزونك أيضًا.

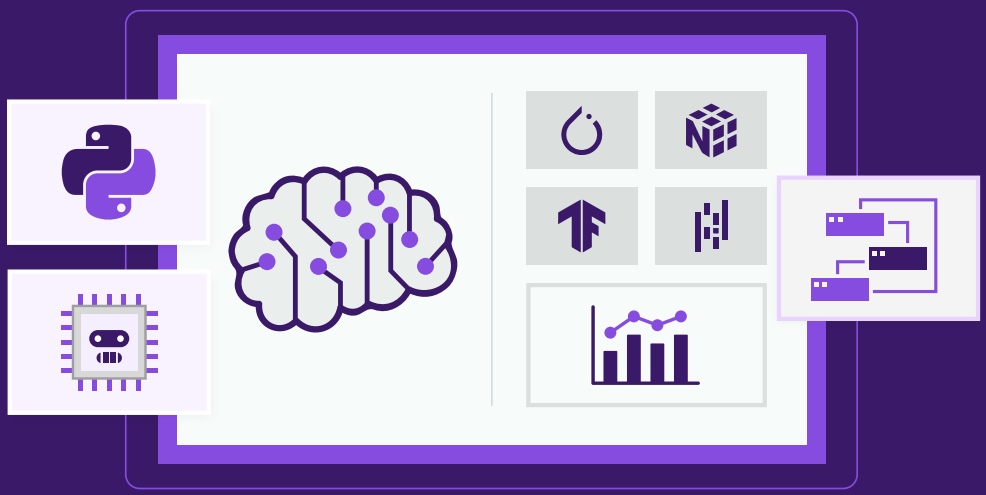
6.4 خاتمة

غطّينا الآن كيفية استخدام التعدادات enums لإنشاء أنواع مُخصصة يمكن أن تمثّل مجموعة من القيم المتعددة، كما استعرضنا نوع المكتبة القياسية `Option<T>` الذي يمكّننا من استخدام نظام الأنواع لتجنب الأخطاء. يمكننا استخدام `match` أو `if let` عندما يحتوي التعداد على قيم داخله لاستخراج واستخدام تلك القيم بحسب الحالات التي نريد معالجتها.

يمكن لبرامج رست الخاصة بك الآن الدلالة على مفاهيم في نطاق استخدامك باستخدام الهياكل والتعدادات. يضمن إنشاء أنواع مخصصة لاستخدامها ضمن واجهتك البرمجية أمان النوع إذ سيتأكد المصرف من حصول الدوال المُستخدمة على نوعٍ محدّد من القيم.

دعنا الآن ننتقل إلى إنشاء وحدات رست حتى نكون قادرين على إنشاء واجهات برمجية منظمة يمكن استخدامها بوضوح وتعرض فقط المعلومات التي سيحتاجها مستخدموها.

دورة الذكاء الاصطناعي



تعلم الذكاء الاصطناعي وتعلم الآلة والتعلم العميق
وتحليل البيانات، وأضفها إلى تطبيقاتك

التحق بالدورة الآن

7. إدارة المشاريع الكبيرة عبر الحزم والوحدات والوحدات المصرفة

تزداد أهمية تنظيم شيفرتك البرمجية مع زيادة تعقيد برامجك، إذ تصبح متابعة كامل الشيفرة البرمجيــة في ذهنك مستحيلةً بحلول تلك النقطة ويمكنـك جعـل الشـيفرة البرمجيـة الخاصـة بمشـروعك واضـحة عن طريـق تجميع الشيفرات البرمجية المتعلقة ببعضها مع بعضها بعضًا وفصل الشيفرات البرمجية غير المتعلقة ببعضها.

كانت البرامج التي كتبناها لحد اللحظة موجودةً ضمن وحـدة module واحـدة وداخـل ملـف واحـد، إلا أنـك تستطيع تنظيم شيفرتك البرمجية مع زيـادة تعقيـد مشـروعك عن طريـق تجزئتهـا إلى وحـدات متعـددة ومن ثم ملفات متعددة. يمكن أن تحتوي الحزمة package على عدة وحدات ثنائية مُصرَّفة binary crates وعلى وحدة مكتبـة مُصـرّفة اختياريـة، ويمكنـك الحصـول على أجـزاء الحزمـة ضـمن وحـدات مُصـرّفة منفصـلة عن طريـق اعتماديات dependencies خارجية. يُغطّي هذا الفصل جميع الطرق المستخدمة في المشاريع الكبـيرة، ويقـدّم كارجو Cargo مساحات العمل workspaces التي سننظر إليها في الفصل الرابع عشر، للمشاريع الكبـيرة الـتي تعتمد على مجموعة من الحزم المترابطة التي يجري تطويرها معًا.

بالإضافة إلى ميزة التجميع، يسمح لك تغليف encapsulation تفاصيل التطبيق بإعادة استخدام الشــيفرة البرمجية ضمن مراحل أعلى؛ إذ يمكن لشيفرة برمجية ما استدعاء شيفرة برمجية أخرى بعد تطبيقـك لعمليـة مـا وذلك باستخدام واجهة الشيفرة البرمجية دون معرفة كيفية عمل التطبيق على نحوٍ دقيـق. تُعـرّف الطريقـة الـتي تكتب فيها الشيفرة البرمجية أيّ الأجزاء منها التي ستكون عامة بحيث تسـتخدمها شـيفرات برمجيـة أخـرى وأي أجزاء ستكون تفاصيل عن التطبيق بصورةٍ خاصة لتحتفظ بها وبحقّ تعـديلها، وهـذه طريقـة أخـرى في الحـدّ من كمية التفاصيل التي يجب عليك إبقاؤها في ذهنك ضمن عملية التطوير.

مصطلح آخر مرتبط بهذا الموضوع هو النطاق scope، إذ يُعرف بأنه النطاق الذي تمتلك فيه شيفرة برمجية ما مجموعةً من الأسماء المعرفة ضمنه. يجب على المصّرف والمبرمجين أن يعلموا إذا كان اسم ما في مكان معين يشير إلى متغيّر أو دالة أو تعداد أو ثابت أو عنصر آخر وما الذي يعنيه ذلك العنصر خلال عملية قراءة وكتابة وتصريف الشيفرة البرمجية. يُمكنك إنشاء النطاقات وتغيير الأسماء الموجودة فيها، ولا يمكنك استخدام الاسم ذاته مرتين داخل ذات النطاق، وهناك بعض الأدوات المتاحة لحلّ مشكلة تضارب الأسماء.

لدى رست مجموعةً من المزايا التي تسمح لك بإدارة وتنظيم شيفرتك البرمجية بما فيها معرفة أيّ التفاصيل المتاحة وأي التفاصيل الخاصة وما هي الأسماء الموجودة في كل نطاق ضمن برنامجك، ويُشار إلى هذه المزايا مجتمعة باسم نظام الوحدة module system ويتضمن:

- الحزم: ميزة في كارجو Cargo تسمح لك ببناء وتجربة ومشاركة الوحدات المصرّفة.
- الوحدات المصرّفة: شجرة من الوحدات التي تُنتِج مكتبةً أو ملفًا تنفيذيًا.
- الوحدات وكلمة `use` المفتاحية: تسمح لك بالتحكم بتنظيم شيفرتك البرمجية ونطاقها وخصوصية المسارات paths.
- المسارات: طريقة في تسمية العناصر مثل الهياكل أو الدوال أو الوحدات.

سنغطّي في هذا الفصل جميع هذه المزايا وسنناقش كيف تتفاعل مع بعضها بعضًا، إضافةً إلى شرحنا كيفية استخدامها لإدارة النطاق، وبنهاية الفصل سيصبح لديك فهم كامل عن نظام الوحدة وستكون قادرًا على التعامل مع النطاقات كأنك محترف.

7.1 الحزم packages والوحدات المصرفة crates

سنُغطّي أولى أجزاء نظام الوحدة ألا وهو الحزم packages والوحدات المصرّفة crates.

الوحدة مُصرَّفة هي الجزء الأصغر من الشيفرة البرمجية التي يستطيع المصرّف التعرف عليها في المرة الواحدة. حتى إذا شغّلنا `rustc` بدلًا من `cargo` ومررنا ملفًا مصدريًا للشيفرة، سيتعامل المصرّف مع هذا الملف على أنه وحدة مُصرَّفة. يمكن أن تحتوي الوحدات المُصرَّفة وحدات modules يمكن أن تكون معرّفة في ملفات أخرى مُصرّفة مع تلك الوحدة المُصرَّفة، كما سنرى في الأقسام التالية.

يُمكن أن تكون الوحدة المُصرَّفة ثنائية binary أو أن وحدة مكتبة مُصرَّفة library، وتُعد الوحدات المُصرَّفة الثنائية برامجًا يُمكنك تصريفها إلى ملف تنفيذي ومن ثم تشغيلها مثل برامج سطر الأوامر command line programs أو الخوادم servers، ويجب أن تحتوي الوحدات المُصرَّفة الثنائية على دالة تدعى `main` تُعرِّف ما الذي يحدث عند تشغيل الملف التنفيذي، وجميع الوحدات المُصرَّفة التي أنشأناها حتى اللحظة هي وحدات مصرّفة ثنائية.

لا تحتوي الوحدات المكتبية المصرّفة على دالة `main` ولا تُصرَّف إلى ملف تنفيذي، وإنما تعرِّف وظائف بُنيَت بهدف الاستفادة منها من خلال مشاركتها من قبل عدّة مشاريع، ونذكر منها على سبيل المثال وحدة المكتبة المصرفة `rand` المُستخدمة في الفصل 2، والتي تقدّم خاصية توليد الأرقام العشوائية. عندما يذكر مستخدمو رست مصطلح وحدة مصرّفة، فهم يقصدون وحدة المكتبة المصرّفة، ويعتمدون مفهوم الوحدة المصرّفة على نحو تبادلي لمصطلح "المكتبة" السائد في مفهوم البرمجة العام.

وحدة الجذر المصرّفة crate root هي الملف المصدري الذي يبدأ منه مصرّف رست ويشكّل وحدة الجذر للوحدة المصرّفة crate، وسنشرح الوحدات بتفصيلٍ أكبر في الفقرات اللاحقة.

الحزمة هي مجمع لوحدة مصرّفة واحدة أو أكثر، وتقدّم مجموعةً من الوظائف، وتحتوي على ملف "Cargo.toml" يصِف كيفية بناء الوحدات المصرّفة داخله. كارجو Cargo هي حزمة تحتوي على وحدة ثنائية مُصرّفة مخصصة لأداة سطر الأوامر المُستخدمة لبناء شيفرتك البرمجية، وتحتوي أيضًا على وحدة مكتبة مصرّفة تعتمد عليها الوحدة الثنائية المُصرّفة. يمكن أن تعتمد المشاريع الأخرى على وحدة مكتبة كارجو المُصرّفة لاستخدام نفس المنطق المُستخدم في أداة سطر أوامر كارجو.

هناك عدّة قواعد تُملي ما على الحزمة أن تحتويه، إذ يمكن أن تحتوي الحزمة على وحدة مكتبة مُصرّفة واحدة فقط، ويمكن أن تحتوي على عدد من الوحدات الثنائية المُصرّفة بحسب حاجتك إلا أنها يجب أن تحتوي على الأقل على وحدةٍ مصرّفة واحدةٍ على الأقل بغضّ النظر عن نوعها سواءٌ كانت وحدة ثنائية أو وحدة مكتبة.

دعنا ننظر ما الذي يحدث عندما نُنشئ حزمة. نُدخل أوّلًا الأمر `cargo new`:

```
$ cargo new my-project
     Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
main.rs
```

بعد تنفيذ الأمر السابق، نستخدم الأمر `ls` لنرى ماذا أنشأ كارجو، إذ سنجد ضمن مجلد المشروع ملفًا اسمه Cargo.toml، والذي يمنحنا حزمة، ونجد أيضًا مجلد "src" يحتوي على الملف "main.rs". وإذا نظرنا إلى Cargo.toml فلن نجد هناك أي ذكر للملف src/main.rs، وذلك لأن كارجو يتبع اصطلاحًا معيّنًا بحيث يكون src/main.rs الوحدة الجذر للوحدة الثنائية المصرّفة باستخدام نفس اسم الحزمة. يعلم كارجو أيضًا أنه إذا احتوى مجلد الحزمة على src/lib.rs فهذا يعني أن الحزمة تحتوي على وحدة مكتبة مصرّفة باسم الحزمة ذاته و src/lib.rs هي الوحدة الجذر في هذه الحالة. يُمرّر كارجو ملفات الوحدة الجذر المصرّفة إلى `rustc` لبناء وحدة مكتبة مصرّفة أو وحدة ثنائية مصرّفة.

لدينا هنا في هذه الحالة حزمة تحتوي src/main.rs فقط، وهذا يعني أنها تحتوي وحدة ثنائية مُصرّفة تُدعى `my-project`، وفي حالة احتواء المشروع على src/main.rs و src/lib.rs في ذات الوقت فهذا يعني أنه يحتوي على وحدتين مصرفتين، وحدة مكتبة مصرّفة ووحدة ثنائية مُصرّفة ويحتوي كلاهما على الاسم ذاته الخاص بالحزمة، ويمكن أن تحتوي الحزمة عدّة وحدات ثنائية مُصرّفة من خلال وضع الملفات في المجلد src/bin، بحيث يُمثّل كل ملف داخل هذا المجلد وحدة ثنائية مُصرّفة منفصلة.

7.2 تعريف الوحدات للتحكم بالنطاق والخصوصية

سننتقل في هذا القسم للتكلم عن الوحدات modules والأجزاء الأخرى من نظام الوحدة، والتي تُسمى **المسارات paths**، وهي تسمح لك بتسمية العناصر؛ وسنتطرق أيضًا إلى الكلمة المفتاحية `use` التي تُضيف مسارًا إلى النطاق؛ والكلمة المفتاحية `pub` التي تجعل من العناصر عامة public؛ كما سنناقش الكلمة المفتاحية `as` والحزم الخارجية وعامل `glob`.

لكن أولًا دعنا نبدأ بمجموعة من القوانين التي تُساعدك بتنظيم شيفرتك البرمجية مستقبلًا، ومن ثمّ سنشرح كل قاعدة من القواعد بالتفصيل.

7.2.1 مرجع سريع للوحدات

إليك كيف تعمل كل من المسارات والوحدات وكلمتَي `use` و `pub` المفتاحيتَين في المصرّف وكيف يُنظّم المطوّرون شيفرتهم البرمجية. سنستعرض مثالًا عن كلٍ من القواعد الموجودة أدناه وقد ترى هذه الفقرة مفيدةً ويمكن استعمالها مثل مرجع سريع في المستقبل لتذكّرك بكيفية عمل الوحدات.

- ابدأ من وحدة الجذر المصرّفة crate root: عند تصريف وحدة مصرّفة ما، ينظر المصرّف أولًا إلى ملف وحدة الجذر المُصرّفة (عادةً src/lib.rs لوحدة المكتبة المصرّفة و src/main.rs للوحدة الثنائية المصرّفة).
- **التصريح عن الوحدات:** يُمكنك التصريح في ملف وحدة الجذر المصرّفة عن وحدة جديدة باسم معيّن وليكن "garden" بكتابة السطر البرمجي `mod garden;`، وسيبحث عندها المصرّف عن الشيفرة البرمجية داخل الوحدة في هذه الأماكن:
  - ضمن السطر ذاته inline: أي ضمن السطر الخاص بالتعليمة `mod garden` ضمن الأقواس المعقوصة عوضًا عن الفاصلة المنقوطة.
  - في الملف src/garden.rs.
  - في الملف src/garden/mod.rs.

- **التصريح عن الوحدات الفرعية submodules:** يُمكنك التصريح عن وحدات فرعية لأي ملف غير وحدة الجذر المصرّفة، إذ يمكنك مثلًا التصريح عن `mod vegtables` في الملف src/garden.rs، وسيبحث المصرّف عن شيفرة الوحدة الفرعية في المجلد المُسمى للوحدة الأصل في هذه الأماكن:
  - ضمن السطر ذاته inline: أي ضمن السطر الخاص بالتعليمة `mod vegetables` ضمن الأقواس المعقوصة عوضًا عن الفاصلة المنقوطة.
  - في الملف src/garden/vegetables.rs.
  - في الملف src/garden/vegetables/mod.rs.
- **المسارات التي ستُشفّر في الوحدات:** بمجرد أن تصبح الوحدة جزءًا من الوحدة المصرّفة، يمكنك الرجوع إلى الشيفرة البرمجية الموجودة في تلك الوحدة من أي مكانٍ آخر في نفس الوحدة المصرّفة، طالما تسمح قواعد الخصوصية بذلك، وذلك باستخدام المسار الواصل إلى هذه الشيفرة. يمكنك مثلًا العثور على نوع `Asparagus` في وحدة الخضار عند المسار `crate::garden::vegetables::Asparagus`.
- **الخاص private والعام public**: الشيفرة البرمجية الموجودة داخل الوحدة هي خاصة بالنسبة للوحدة الأصل افتراضيًا، ولجعل الوحدة عامة يجب أن نصرح عنها باستخدام `pub mod` بدلًا من `mod`، ولجعل العناصر الموجودة داخل الوحدة العامة عامة أيضًا نستخدم `pub` قبل التصريح عنها.
- **الكلمة المفتاحية: `use`** تُنشئ الكلمة المفتاحية `use` داخل النطاق اختصارًا للعناصر لتجنب استخدام المسارات الطويلة، إذ يمكنك في أي نطاق اختصار المسار `crate::garden::vegetables::Asparagus` باستخدام `use` عن طريق كتابة `use crate::garden::vegetables::Asparagus;` ومن ثمّ يمكنك كتابة `Asparagus` مباشرةً ضمن النطاق دون الحاجة لكتابة كامل المسار.

إليك وحدة ثنائية مصرّفة اسمها `backyard` توضح القوانين السابقة. نُسمّي مسار الوحدة المصرّفة أيضًا `backyard` ويحتوي ذلك المسار على هذه الملفات والمسارات:

```
backyard
├── Cargo.lock
├── Cargo.toml
└── src
    ├── garden
    │   └── vegetables.rs
    ├── garden.rs
    └── main.rs
```

في هذه الحالة، يحتوي ملف وحدة الجذر المصرّفة src/main.rs على التالي:

اسم الملف: src/main.rs

```
use crate::garden::vegetables::Asparagus;

pub mod garden;

fn main() {
    let plant = Asparagus {};
    println!("I'm growing {:?}!", plant);
}
```

يعني السطر البرمجي `pub mod garden;` بأن المصرّف سيضمّ الشيفرة البرمجية التي سيجدها في src/garden.rs، وهي:

اسم الملف: src/garden.rs

```
pub mod vegetables;
```

ويعني السطر `pub mod vegetables;` بأنّ الشيفرة البرمجية الموجودة في الملف src/garden/vegetables.rs ستُتضمّن أيضًا:

```
#[derive(Debug)]
pub struct Asparagus {}
```

لننظر إلى ما سبق عمليًا مع بتطبيق القوانين التي ذكرناها سابقًا.

7.2.2 تجميع الشيفرات البرمجية المرتبطة ببعضها في الوحدات

تسمح لنا الوحدات بتنظيم الشيفرة البرمجية ضمن وحدات مصرّفة على شكل مجموعات لزيادة سهولة القراءة والاستخدام، وتتحكم الوحدات أيضًا بخصوصية privacy العناصر داخلها، لأن الشيفرات داخل الوحدات خاصة افتراضيًا، والعناصر الخاصة هي تفاصيل داخلية تطبيقية خاصة وغير مُتاحة للاستخدام الخارجي. يمكننا اختيار إنشاء وحدات وعناصر وجعلها عامة، وهذا يسمح باستخدام الشيفرات الخارجية والاعتماد عليها.

دعنا نكتب وحدة مكتبة مصرّفة تزوّدنا بخصائص مطعم مثلًا، وسنعرّف داخلها بصمات signatures الدوال، لكن سنترك محتوى كل منها فارغًا لنركّز على جانب تنظيم الشيفرة البرمجية بدلًا من التطبيق الفعلي لمطعم.

يُشار في مجال المطاعم إلى بعض الأجزاء بكونها أجزاء على الواجهة الأمامية front of house بينما يُشار إلى أجزاء أخرى بأجزاء في الخلفية back of house؛ وأجزاء الواجهة الأمامية هي حيث يتواجد الزبائن، أي جانب حجز الزبائن للمقاعد وأخذ الطلبات والمدفوعات منهم وتحضير النادل للمشروبات؛ بينما أجزاء الواجهة هي حيث يتواجد الطهاة وتُطهى الأطباق وتُنظّف الصحون والأعمال الإدارية التي يجريها المدير.

من أجل هيكلة الوحدة المصرّفة بطريقة مماثلة للمطعم، نلجأ لتنظيم الدوال في وحدات متداخلة. لننشئ مكتبةً جديدة ونسميها `restaurant` بتنفيذ الأمر `cargo new restaurant --lib`، ثمّ نكتب الشيفرة 1 داخل الملف src/lib.rs لتعريف بعض الوحدات وبصمات الدوال.

اسم الملف: src/lib.rs

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

[الشيفرة 1: وحدة front_of_house تحتوي على وحدات أخرى، وتحتوي هذه الوحدات الأخرى بدورها على دوال]

نعرّف الوحدة عن طريق البدء بكتابة الكلمة المفتاحية `mod` ومن ثم تحديد اسم الوحدة (في حالتنا هذه الاسم هو `front_of_house`) ونضيف بعد ذلك أقواس معقوصة حول متن الوحدة. يمكننا إنشاء وحدات أخرى داخل وحدة ما وهذه هي الحالة مع الوحدات `hosting` و `serving`، يمكن للوحدات أيضًا أن تحتوي داخلها على تعريفات لعناصر أخرى، مثل الهياكل structs والتعدادات enums والثوابت constants والسمات traits أو الدوال كما هو موجود في الشيفرة 1.

يمكننا تجميع التعريفات المرتبطة ببعضها بعضًا باستخدام الوحدات وتسمية العامل الذي يربطهم، وبالتالي سيكون من الأسهل للمبرمجين الذين يستخدمون الشيفرة البرمجية هذه أن يعثروا على التعريفات التي يريدوها

لأنه من الممكن تصفح الشيفرة البرمجية بناءً على المجموعات بدلًا من قراءة كل التعريفات، كما أنه سيكون من السهل إضافة مزايا جديدة إلى الشيفرة البرمجية لأن المبرمج سيعرف أين يجب إضافة الشيفرة البرمجيــة بحيث يحافظ على تنظيمها.

ذكرنا سابقًا أن src/main.rs و src/lib.rs هي وحدات جذر مُصرَّفة، والسبب في تسـميتهما بـذلك هـو أن محتويات أيّ منهما يشكّل وحدةً تدعى `crate` في جذر هيكل الوحدة الخاص بالوحدة المصرّفة وكما تُعرف أيضًا بشجرة الوحدة module tree.

توضح الشيفرة 2 شجرة الوحدة الخاصة بهيكل الشيفرة 1.

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

[الشيفرة 2: شجرة الوحدة للشيفرة 1]

توضح الشجرة بأن بعض الوحدات متداخلة مع وحدات أخرى (على سبيل المثال الوحدة `hosting` متداخلة مع `front_of_house`)، كما توضح الشجرة أيضًـا أن بعض الوحـدات أشـقاء siblings لبعضـها، بمعـنى أنهـا معرفة داخل الوحدة ذاتها (`hosting` و `serving` معرفتان داخل `front_of_house`). لإبقـاء تشـبيه شـجرة العائلة، إذا كانت الوحدة (أ) محتـواة داخـل الوحـدة (ب) نقـول بـأن الوحـدة (أ) هي ابن child للوحـدة (ب) وأن الوحــدة (ب) هي الأب parent للوحــدة (أ). لاحــظ أن كامــل الشــجرة محتــواة داخــل الوحــدة الضــمنية المسماة `crate`.

قد تذكرك شجرة الوحدة بشـجرة مسـارات الملفـات على حاسـبك، وهـذه مقارنـة ملائمـة، إذ أننـا نسـتخدم الوحدات لتنظيم الشيفرة البرمجية بنفس الطريقة التي نستخدم فيها المجلدات لتنظيم الملفات، وكما هو الحال في الملفات داخل المجلدات نحتاج إلى طريق لإيجاد الوحدات.

7.3 المسارات paths وشجرة الوحدة module tree

يُستخدم المسار path بنفس الطريقة المُستخدمة عند التنقل ضمن نظام الملفات في الحاسوب حتى ندلّ رست على مكان وجود عنصر ما ضمن شجرة الوحدة module tree، وبالتالي علينا معرفة مسار الدالة أولًا إذا أردنا استدعائها.

قد يكون المسار واحدًا من النوعين التاليين:

- المسار المُطلق absolute path: وهو المسار الكامل بدءًا من جذر الوحدة المصرّفة؛ إذ أن المسار المُطلق لشيفرة برمجية موجودة في وحدة مصرّفة خارجية تبدأ باسم الوحدة المصرّفة بينما يبدأ مسار شيفرة برمجية داخل الوحدة المصرّفة الحالية المُستخدمة بالكلمة `crate`.
- المسار النسبي relative path: ويبدأ هذا المسار من الوحدة المصرّفة الحالية المُستخدمة ويستخدم الكلمة المفتاحية `self`، أو `super`، أو معرّف ينتمي إلى الوحدة نفسها.

يُتبع كلا نوعَي المسارات السابقَين بمعرّف identifier واحد، أو أكثر ويفصل بينهما نقطتان مزدوجتان `::`.

بالعودة إلى الشيفرة 1، لنفترض أننا نريد استدعاء الدالة `add_to_waitlist`، وهذا يقتضي أن نسأل أنفسنا: ما هو مسار الدالة `add_to_waitlist`؟ تحتوي الشيفرة 3 على الشيفرة 1 مع إزالة بعض الوحدات والدوال.

سنستعرض طريقتين لاستدعاء الدالة `add_to_waitlist` من دالة جديدة معرّفة في جذر الوحدة المصرّفة وهي `eat_at_restaurant` في هذه الحالة، والمسارات الموجودة في كل من الطريقتين صالحة إلا أن هناك مشكلة أخرى ستمنع مثالنا من أن يُصرَّف كما هو، وسنفسّر ذلك قريبًا.

تشكل الدالة `eat_at_restaurant` جزءًا من الواجهة البرمجية العامة public API الخاصة بوحدة المكتبة المصرّفة library crate، لذا نُضيف الكلمة المفتاحية `pub` إليها، وسنناقش المزيد من التفاصيل بخصوص `pub` في الفقرة التالية.

اسم الملف: src/lib.rs

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
```

```
    // مسار مطلق
    crate::front_of_house::hosting::add_to_waitlist();

    // مسار نسبي
    front_of_house::hosting::add_to_waitlist();
}
```

[الشيفرة 3: استدعاء الدالة add_to_waitlist باستخدام المسار المُطلق والمسار النسبي]

نستخدم مسارًا مُطلقًا عندما نريد استدعاء الدالة add_to_waitlist داخل eat_at_restaurant للمرة الأولى، ويمكننا استخدام المسار المطلق بدءًا بالكلمة المفتاحية crate بالنظر إلى أن الدالة add_to_waitlist معرفة في نفس الوحدة المصرّفة الخاصة بالدالة eat_at_restaurant، ومن ثمّ نضمّن كل من الوحدات الأخرى الموجودة في شجرة الوحدة module tree حتى الوصول إلى الدالة add_to_waitlist. يمكنك تخيّل هذا الأمر بصورةٍ مشابهة لنظام الملفات على حاسوبك، إذ نكتب المسار /front_of_house/hosting/add_to_waitlist لتشغيل البرنامج add_to_waitlist، إلا أننا نستخدم الكلمة المفتاحية crate للدلالة على جذر الوحدة المصرّفة بدلًا من استخدام / في بداية المسار، وهو أمرٌ مشابه لكتابة / في سطر الأوامر حتى تصل إلى جذر نظام الملفات على حاسوبك.

نستخدم المسار النسبي في المرة الثانية التي نستدعي add_to_waitlist في eat_at_restaurant، ويبدأ المسار هنا بالاسم front_of_house، وهو اسم الوحدة المعرفة ضمن نفس مستوى eat_at_restaurant في شجرة الوحدة، وسيستخدم نظام الملفات المكافئ المسار front_of_house/hosting/add_to_waitlist بدءًا باسم الوحدة مما يعني أن المسار هو مسار نسبي.

يجب الاختيار بين المسار المطلق والنسبي بناءً على مشروعك ويعتمد على إذا ما كنت ستميل غالبًا إلى نقل شيفرة تعريف العنصر البرمجية وتفريقها عن الشيفرة البرمجية التي تستخدم ذلك العنصر، أو إذا كنت ستبقيهما سويًا. على سبيل المثال، إذا أردنا نقل الوحدة front_of_house والدالة eat_at_restaurant إلى وحدة باسم customer_experience، سنضطر إلى تحديث المسار المطلق الخاص بالدالة add_to_waitlist، إلا أن المسار سيبقى صالحًا في حال استخدمنا المسار النسبي، لكن إذا نقلنا الدالة eat_at_restaurant بصورةٍ منفصلة إلى وحدة جديدة تدعى dining فلن يكون المسار النسبي لاستدعاء الدالة add_to_waitlist صالحًا بعد الآن ويجب تحديثه، إلا أن المسار المطلق سيبقى صالحًا. نفضّل هنا المسارات المطلقة، لأننا على الأغلب سنحرّك تعريف الشيفرة البرمجية واستدعاء العناصر على نحوٍ متفرق عن بعضهما بعضًا.

دعنا نجرّب تصريف الشيفرة 3 ونقرأ الخطأ ونحاول معرفة السـبب في عـدم قابليـة تصـريفها. نحصـل على الخطأ الموضح في الشيفرة 4.

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: module `hosting` is private
 --> src/lib.rs:9:28
  |
|     crate::front_of_house::hosting::add_to_waitlist();
  |                            ^^^^^^^ private module
  |
note: the module `hosting` is defined here
 --> src/lib.rs:2:5
  |
|     mod hosting {
  |     ^^^^^^^^^^

error[E0603]: module `hosting` is private
  --> src/lib.rs:12:21
   |
|     front_of_house::hosting::add_to_waitlist();
   |                     ^^^^^^^ private module
   |
note: the module `hosting` is defined here
  --> src/lib.rs:2:5
   |
|     mod hosting {
   |     ^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors
```

[الشيفرة 4: أخطاء المصرف الناجمة عن تصريف الشيفرة 3]

تدلنا رسالة الخطأ على أن الوحدة `hosting` هي وحدة خاصة، بمعنى أنه على الرغم من اسـتخدامنا للمسـار الصحيح الخاص بوحدة `hosting` ودالة `add_to_waitlist` إلا أن رست لن تسمح لنـا باسـتخدامهما لأنـه لا يوجد لدينا سماحية الوصول إلى هذه الأجزاء الخاصة. جميع العناصر في رست (دوال وتوابع وهياكـل وتعـدادات

ووحدات وثوابت) هي خاصة بالوحدة الأب (الأصل) فقط افتراضيًا، وإذا أردت جعل عنصر ما مثل دالة أو هيكل خاصًا، فعليك وضعه داخل الوحدة.

لا يمكن لعناصر في وحدة أب استخدام العناصر الخاصة داخل الوحدات التابعة لها، إلا أن وحدات الابن يمكن أن تستخدم العناصر الموجودة في الوحدات الأب، وهذا لأن الوحدات الابن تغلّف وتُخفي تفاصيل تطبيقها وتستطيع الوحدات الابن رؤية السياق الخاص بتعريفها، وحتى نستمرّ في تشبيهنا السابق للمطعم، تخيل أن قوانين الخصوصية مشابهة للمكاتب الإدارية للمطعم، فالذي يحصل في هذه المكاتب هو معزول عن زبائن المطعم، لكن يستطيع مدراء المطعم رؤية أي شيء في المطعم.

تختار رست بأن يعمل نظام الوحدة على هذا النحو، بحيث يكون إخفاء تفاصيل التطبيق الداخلي هو الحالة الافتراضية، وبالتالي تستطيع بهذه الطريقة معرفة أي الأجزاء من الشيفرة الداخلية التي يمكنك تغييرها دون التسبب بأخطاء في الشيفرة الخارجية. مع ذلك، تعطيك رست الخيار لكشف الأجزاء الداخلية من الوحدات الابن للوحدات الخارجية باستخدام الكلمة المفتاحية `pub` لجعل العنصر عامًا.

7.3.1 كشف المسارات باستخدام الكلمة المفتاحية pub

بالعودة إلى الخطأ الموجود في الشيفرة 4 الذي كان مفاده أن الوحدة `hosting` هي وحدة خاصة، نريد أن تمتلك الدالة `eat_at_restaurant` الموجودة في الوحدة الأب وصولًا إلى الدالة `add_to_waitlist` الموجودة في الوحدة الابن، ولتحقيق ذلك نُضيف الكلمة المفتاحية `pub` إلى الوحدة `hosting` كما هو موضح في الشيفرة 5.

اسم الملف: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // مسار مطلق
    crate::front_of_house::hosting::add_to_waitlist();

    // مسار نسبي
    front_of_house::hosting::add_to_waitlist();
}
```

[الشيفرة 5: التصريح عن الوحدة hosting باستخدام الكلمة المفتاحية pub لاستخدامها داخل eat_at_restaurant]

لسوء الحظ، تتسبب الشيفرة 5 بخطأ موضح في الشيفرة 6.

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: function `add_to_waitlist` is private
 --> src/lib.rs:9:37
  |
|     crate::front_of_house::hosting::add_to_waitlist();
  |                                     ^^^^^^^^^^^^^^^ private
function
  |
note: the function `add_to_waitlist` is defined here
 --> src/lib.rs:3:9
  |
|         fn add_to_waitlist() {}
  |         ^^^^^^^^^^^^^^^^^^^

error[E0603]: function `add_to_waitlist` is private
  --> src/lib.rs:12:30
   |
|     front_of_house::hosting::add_to_waitlist();
   |                              ^^^^^^^^^^^^^^^ private function
   |
note: the function `add_to_waitlist` is defined here
  --> src/lib.rs:3:9
   |
|         fn add_to_waitlist() {}
   |         ^^^^^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors
```

[الشيفرة 6: أخطاء المصرّف الناتجة عن بناء الشيفرة 5]

ما الذي حصل؟ تجعل إضافة الكلمة المفتاحية `pub` أمام `mod hosting` من الوحـدة وحـدةً عامـةً، وبهـذا التغيير إن أمكننا الوصول إلى `front_of_house`، فهذا يعني أنه يمكننا الوصول إلى `hosting`، ولكن محتوى

hosting ما زال خاصًّا، إذ أن تحويل الوحدة إلى وحدة عامة لا يجعل من محتواها عامًا أيضًا، إذ أن استخدام الكلمة المفتاحية pub على وحدة ما يسمح الوحدات الأب بالإشارة إلى الشيفرة البرمجية فقط وليس الوصول إلى الشيفرة البرمجية الداخلية، ولأن الوحدات هي بمثابة حاويات فليس هناك الكثير لفعله بجعل الوحدة فقط عامة، وسنحتاج إلى جعل عنصر واحد أو أكثر من عناصرها عامًا أيضًا.

تدلنا الأخطاء الموجودة في الشيفرة 6 إلى أن الدالة add_to_waitlist هي دالة خاصة، وتُطبَّق قوانين الخصوصية على الهياكل والتعدادات والتوابع والدوال كما هو الحال مع الوحدات.

دعنا نجعل من الدالة add_to_waitlist دالة عامة بإضافة الكلمة المفتاحية pub قبل تعريفها كما هو موضح في الشيفرة 7.

اسم الملف: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // مسار مطلق
    crate::front_of_house::hosting::add_to_waitlist();

    // مسار نسبي
    front_of_house::hosting::add_to_waitlist();
}
```

[الشيفرة 7: إضافة الكلمة المفتاحية pub إلى mod hosting و fn add_to_waitlist مما يسمح لنا باستدعاء الدالة من eat_at_restaurant]

يمكننا الآن تصريف الشيفرة البرمجية بنجاح. دعنا ننظر إلى المسارات المطلقة والمسارات النسبية حتى نفهم لماذا تسمح لنا إضافة pub باستخدام هذه المسارات في add_to_waitlist مع مراعاة قوانين الخصوصية.

نبدأ بكتابة crate في المسار المطلق وهو جذر شجرة الوحدة الخاصة بالوحدة المصرّفة. تُعرّف الوحدة front_of_house في جذر الوحدة المصرّفة، إلا أنها ليست عامة، وذلك لأن الدالة eat_at_restaurant معرفة في الوحدة ذاتها الخاصة بالوحدة front_of_house (أي أن eat_at_restaurant و front_of_house أشقاء)، ويُمكننا الإشارة إلى front_of_house من eat_at_restaurant. تاليًا

تبدو الوحدة `hosting` معلّمة marked بفضل الكلمة المفتاحية `pub`، إذ يمكننا الوصول إلى الوحدة الأب الخاصة بالوحدة `hosting`، وبالتالي يمكننا الوصول إلى `hosting`، وأخيرًا، الدالة `add_to_waitlist` مُعلّمة بالكلمة المفتاحية `pub`، وبالتالي يعمل بقية المسار ويُعد استدعاء الدالة هذا صالحًا.

نعتمد في المسار النسبي نفس المنطق في المسار المطلق باستثناء الخطوة الأولى، إذ بدلًا من البدء بجذر الوحدة المصرّفة، يبدأ المسار من الوحدة `front_of_house`، التي تُعرّف في الوحدة ذاتها الخاصة بالوحدة `eat_at_restaurant`، وبالتالي يبدأ المسار النسبي من الوحدة التي يعمل عندها تعريف الوحدة `eat_at_restaurant`. تبدو الوحدة `hosting` و `add_to_waitlist` معلّمة بفضل الكلمة المفتاحية `pub`، ولذلك يعمل بقية المسار ويُعد استدعاء الدالة هذا صالحًا.

إذا أردت مشاركة وحدة المكتبة المصرّفة الخاصة بك بحيث تستفيد مشاريع أخرى من الشيفرة البرمجية الخاصة بالوحدة المصرّفة، ستكون واجهتك البرمجية API العامة هي نقطة الوصل بين مستخدمي الوحدة المصرّفة ومحتوياتها وهي التي تحدد كيفية تفاعل المستخدمين مع شيفرتك البرمجية، وهناك الكثير من الأشياء التي يجب وضعها في الحسبان لإدارة التغييرات التي تجريها على الواجهة البرمجية العامة حتى تجعل عملية الاعتماد على وحدتك المصرّفة أسهل بالنسبة للمستخدمين، إلا أن هذه الأشياء خارج نطاق موضوعنا هنا، وإذا كنت مهتمًا بهذا الخصوص عليك رؤية دليل رست الخاصة بالواجهات البرمجية.

الممارسات المثلى للحزم التي تحتوي على وحدات ثنائية مصرفة ووحدات مكتبة مصرفة

ذكرنا أنه يمكن أن تحتوي الحزمة على كلٍّ من جذر وحدة ثنائية مصرّفة src/main.rs إضافةً إلى جذر وحدة مكتبة مصرّفة src/lib.rs وأن يحمل كلتا الوحدتين المصرّفتين اسم الحزمة افتراضيًا، ويحتوي هذا النوع من الحزم عادةً على شيفرة برمجية كافية داخل الوحدة الثنائية المصرّفة، بحيث يمكن إنشاء ملف تنفيذي باستخدامها يستدعي الشيفرة البرمجية الموجودة في وحدة المكتبة المصرّفة، مما يسمح للمشاريع الأخرى بالاستفادة القصوى من المزايا التي تقدمها هذه الحزمة وذلك لأنه من الممكن مشاركة الشيفرة البرمجية الموجودة داخل وحدة المكتبة المصرّفة.

يجب أن تُعرَّف شجرة الوحدة في الملف src/lib.rs، ومن ثمّ يمكننا استخدام أي عنصر عام في الوحدة الثنائية المُصرّفة ببدء المسار باسم الحزمة، وبذلك تصبح الوحدة الثنائية المصرّفة مستخدمًا user لوحدة المكتبة المصرّفة كما تستخدم أية وحدة مصرّفة خارجية وحدة مكتبة مصرّفة عادةً، وذلك باستخدام الواجهة البرمجية العامة فقط. يساعدك ذلك في تصميم واجهة برمجية جيّدة؛ إذ أنك لست كاتب المكتبة فقط في هذه الحالة، بل أنك مستخدمها أيضًا.

سنستعرض الممارسات الشائعة في تنظيم برامج سطر الأوامر التي تحتوي على كل من الوحدة الثنائية المصرّفة ووحدة المكتبة المصرّفة لاحقًا في الفصل 12.

7.3.2 كتابة المسارات النسبية باستخدام super

يُمكننا إنشاء مسار نسبي يبدأ في الوحدة الأب بدلًا من الوحدة الحالية أو جذر وحدة مصرّفة باستخدام الكلمة المفتاحية `super` في بداية المسار، وهذا يُشابه بدء مسار الملفات بالنقطتين `..`، إذ يسمح لنا استخدام `super` بالإشارة إلى عنصر نعلم أنه موجود في الوحدة الأب، مما يجعل إعادة ترتيب شجرة الوحدة أسهل عندما تكون الوحدة مرتبطة بصورةٍ وثيقة مع الوحدة الأب مع احتمال نقل الوحدة الأب إلى مكان آخر ضمن شجرة الوحدة في يوم ما.

ألقِ نظرةً إلى الشيفرة 8 التي تصف موقفًا معينًا، ألا وهو تصحيح الطاهي لطلب غير صحيح وإحضاره بنفسه شخصيًا إلى الزبون. تستدعي الدالة `fix_incorrect_order` المُعرفة في الوحدة `back_of_house` الدالة `deliver_order` المعرّفة في الوحدة الأب بتحديد المسار إلى الدالة `deliver_order` باستخدام الكلمة المفتاحية `super` في البداية:

اسم الملف: src/lib.rs

```
fn deliver_order() {}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::deliver_order();
    }

    fn cook_order() {}
}
```

[الشيفرة 8: استدعاء دالة باستخدام المسار النسبي بدءًا بالكلمة super]

تتواجد الدالة `fix_incorrect_order` في الوحدة `back_of_house`، لذا يمكننا استخدام `super` للإشارة إلى الوحدة الأب الخاصة بالوحدة `back_of_house` التي هي في هذه الحالة الوحدة `crate` الجذر، ومن هناك نبحث عن `deliver_order` ونجده بنجاح. نعتقد هنا أن الوحدة `back_of_house` والدالة `deliver_order` سيبقيان سويًا نظرًا للعلاقة فيما بينهما مهما تغيّر تنظيم شجرة الوحدة، وبالتالي استخدمنا `super` بحيث نختصر على أنفسنا أماكن تحديث الشيفرة البرمجية في المستقبل إذا قرّرنا نقل الشيفرة البرمجية إلى وحدة مختلفة.

7.3.3 جعل الهياكل والتعدادات عامة

يمكننا أيضًا استخدام الكلمة المفتاحية pub لتعيين الهياكل structs والتعدادات enums على أنها عامة، إلا أن هناك بعض التفاصيل الإضافية لاستخدام pub مع الهياكل والتعدادات؛ فإذا استخدمنا pub قبل تعريف الهيكل، فسيتسبب هذا بإنشاء هيكل عام إلا أن حقول هذا الهيكل ستظلّ خاصة، ويمكن جعل كل حقل عامًا أو لا اعتمادًا على أساس وشروط كل حالة بحد ذاتها. نُعرّف في الشيفرة 9 هيكلًا عامًا باسم back_of_house::Breakfast بحقل عام toast وحقل خاص seasonal_fruit، يُحاكي هذا الأمر عملية اختيار الزبون لنوع الخبز الذي يأتي مع الوجبة واختيار الطاهي لنوع الفاكهة الذي يأتي مصحوبًا مع الوجبة بناءً على المتاح في مخزون المطعم؛ ولأن الفاكهة المُتاحة تتغير سريعًا حسب المواسم فالزبون لا يستطيع اختيار الفاكهة أو حتى رؤية النوع الذي سيحصل عليه.

اسم الملف: src/lib.rs

```
mod back_of_house {
    pub struct Breakfast {
        pub toast: String,
        seasonal_fruit: String,
    }

    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("peaches"),
            }
        }
    }
}

pub fn eat_at_restaurant() {
    //  طلب فطور في الصيف مع خبز محمص من النوع راي Rye
    let mut meal = back_of_house::Breakfast::summer("Rye");
    // غيّرنا رأينا بخصوص الخبز الذي نريده
    meal.toast = String::from("Wheat");
    println!("I'd like {} toast please", meal.toast);
```

```
    // لن يُصرَّف السطر التالي إذا أزلنا تعليقه لأنه من غير المسموح رؤية أو تعدي الفاكهة الموسمية التي تأتي مع الوجبة
    // meal.seasonal_fruit = String::from("blueberries");
}
```

[الشيفرة 9: هيكل يحتوي على بعض الحقول العامة والخاصة]

بما أن الحقل `toast` في الهيكل `back_of_house::Breakfast` هو حقل عام، فهذا يعني أننا نستطيع الكتابة إلى والقراءة من الحقل `toast` في `eat_at_restaurant` باستخدام النقطة. لاحظ أنه لا يمكننا استخدام الحقل `seasonal_fruit` في `eat_at_restaurant` وذلك لأن الحقل `seasonal_fruit` هو حقل خاص. حاول إزالة تعليق السطر البرمجي الذي يُعدّل من قيمة الحقل `seasonal_fruit` وستحصل على خطأ.

لاحظ أن الهيكل `back_of_house::Breakfast` بحاجة لتوفير دالة عامة مرتبطة به تُنشئ نسخةً من `Breakast` (سميناها `summer` في هذا المثال)، لأنه يحتوي على حقل خاص، ولن يصبح بالإمكان إنشاء نسخة من الهيكل `Breakfast` في `eat_at_restaurant` إذا لم يحتوي على دالة مشابهة، وذلك لأنه لن يكون بإمكاننا ضبط قيمة للحقل الخاص `seasonal_fruit` في `eat_at_restaurant`.

وفي المقابل، إذا أنشأنا تعدادًا عامًا، ستكون جميع متغيراته variants عامة، ونحن بحاجة لاستخدام الكلمة المفتاحية `pub` مرةً واحدةً فقط قبل الكلمة `enum` كما هو موضح في الشيفرة 10.

اسم الملف: src/lib.rs

```
mod back_of_house {
    pub enum Appetizer {
        Soup,
        Salad,
    }
}

pub fn eat_at_restaurant() {
    let order1 = back_of_house::Appetizer::Soup;
    let order2 = back_of_house::Appetizer::Salad;
}
```

[الشيفرة 10: تحديد المعدد على أنه عام يجعل جميع متغيراته عامة]

يمكننا استخدام المتغايرات Soup و Salad في eat_at_restaurant وذلك لأننا أنشأنا التعداد Appetier على أنه تعداد عام.

ليست التعدادات مفيدة إلا إذا كانت جميع متغايراتها عامة، وسيكون من المزعج تحديد كل متغاير من متغايرات المعدد بكونه متغاير علني باستخدام pub، لذا فإن حالة متغايرات التعداد الافتراضية هي أن تكون عامة. على النقيض، يمكن أن تكون الهياكل مفيدة دون أن يكون أحد حقولها عامًا وبالتالي تتبع الهياكل القاعدة العامة التي تنص على أن جميع العناصر خاصة إلا إذا أُشير إلى العنصر على أنه عام باستخدام pub.

هناك بعض الحالات الأخرى التي تتضمن pub التي لم نناقشها بعد، ألا وهي آخر مزايا نظام الوحدات: الكلمة المفتاحية use، والتي سنغطّيها تاليًا، ومن ثمّ سنناقش كيفية دمج pub و use معًا.

7.4 المسارات paths والنطاق الخاص بها

قد تكون عملية كتابة مسارات استدعاء الدوال في بعض الأحيان عملية غير مريحة ورتيبة، كان علينا في الشيفرة 7 (سابقًا) تحديد front_of_house و hosting في حال اخترنا المسار النسبي relative path أو المسار المطلق absolute path إلى الدالة add_to_waitlistوفي كل مرة أردنا استدعاء الدالة add_to_waitlist أيضًا. لحسن الحظ هناك طريقة لتبسيط العملية السابقة، إذ يمكننا إنشاء اختصار للمسار باستخدام الكلمة المفتاحية use مرةً واحدةً فقط ومن ثمّ استخدام المسار الأقصر المُختصر في كل مكان ضمن النطاق.

نُضيف الوحدة crate::front_of_house::hosting إلى نطاق دالة eat_at_restaurant في الشيفرة 11، وذلك حتى نكتفي بكتابة hosting::add_to_waitlist لاستدعاء الدالة add_to_waitlist في eat_at_restaurant.

اسم الملف: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

[الشيفرة 11: إضافة الوحدة إلى النطاق باستخدام use]

تُعد عملية إضافة use متبوعةً بالمسار في النطاق عمليةً مشابهةً لإنشاء رابط رمزي symbolic link في نظام الملفات، إذ يصبح hosting اسمًا صالحًا بإضافة use crate::front_of_house::hosting في جذر الوحدة المصرّفة وضمن ذلك النطاق وكأن الوحدة hosting معرفةٌ داخل جذر الوحدة المصرّفة. تتبع المسارات الموجودة ضمن النطاق باستخدام use قوانين الخصوصية مثلها مثل أي مسار آخر.

لاحظ أن use يُنشئ اختصارًا للنطاق الذي يحتوي على use فقط. ننقل في الشيفرة 12 الدالة eat_at_restaurant إلى وحدة ابن جديدة تدعى customer وهي ذات نطاق مختلف عن تعليمة use، لذا لن نستطيع تصريف محتوى الدالة:

اسم الملف: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

mod customer {
    pub fn eat_at_restaurant() {
        hosting::add_to_waitlist();
    }
}
```

[الشيفرة 12: تعليمة use تُطبّق فقط في النطاق الواردة فيها]

يوضح خطأ التصريف التالي أن الاختصار غير موجود ضمن الوحدة customer:

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0433]: failed to resolve: use of undeclared crate or module
`hosting`
  --> src/lib.rs:11:9
   |
|         hosting::add_to_waitlist();
   |         ^^^^^^^ use of undeclared crate or module `hosting`
```

```
warning: unused import: `crate::front_of_house::hosting`
 --> src/lib.rs:7:5
  |
| use crate::front_of_house::hosting;
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_imports)]` on by default

For more information about this error, try `rustc --explain E0433`.
warning: `restaurant` (lib) generated 1 warning
error: could not compile `restaurant` due to previous error; 1 warning
emitted
```

لاحظ وجود تحذير بخصوص عدم وجود `use` في النطاق، ولتصحيح هذه المشكلة ننقل التعليمة `use` إلى الوحدة `customer` أيضًا، أو نُشير إلى الاختصار في الوحدة الأب باستخدام `super::hosting` ضمن الوحدة الابن `customer`.

7.4.1 إنشاء مسارات اصطلاحية Idiomatic باستخدام use

لعلّك تساءلت عند قراءتك للشيفرة 11 عن سبب كتابة `use crate::front_of_house::hosting` ومن ثم استدعائنا `hosting::add_to_waitlist` في `eat_at_restaurant` بدلًا من تحديد مسار `use` الموجود في الدالة `add_to_waitlist` لتحقيق النتيجة ذاتها كما هو الأمر في الشيفرة 13.

اسم الملف: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting::add_to_waitlist;

pub fn eat_at_restaurant() {
    add_to_waitlist();
}
```

[الشيفرة 13: إضافة الدالة add_to_waitlist إلى النطاق باستخدام use وهو استخدام غير اصطلاحي unidiomatic]

على الـرغم من أن الشـيفرة 11 والشـيفرة 13 يحققـان الغـرض ذاتـه، إلا أن الشـيفرة 11 هي الطريقـة الاصطلاحية لإضافة دالة إلى النطاق باستخدام use. تعني إضافة وحدة الدالـة الأب إلى النطـاق باسـتخدام use أننا بحاجة تحديد الوحدة الأب عند استدعاء الدالـة، وتحديـد الوحـدة الأب عنـد اسـتدعاء الدالـة يوضّـح أن الدالـة ليست معرّفة محليًا مع المحافظة على تقليـل تكـرار كامـل المسـار. لا توضـح الشـيفرة البرمجيـة الموجـودة في الشيفرة 13 مكان تعريف add_to_waitlist.

على الجـانب الآخـر، عنـد إضـافة الهياكـل والتعـدادات والعناصـر الأخـرى إلى النطـاق باسـتخدام use فـإن الاستخدام الاصطلاحي هو تحديد كامل المسار، وتوضـح الشـيفرة 14 الطريقـة الاصـطلاحية في إضـافة الهيكـل HashMap الموجود في المكتبة القياسية إلى نطاق الوحدة الثنائية المُصرّفة.

اسم الملف: src/main.rs

```
use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();
    map.insert(1, 2);
}
```

[الشيفرة 14: إضافة HashMap إلى النطاق بطريقة اصطلاحية]

لا يوجد هناك أي مبرر قوي بخصوص هـذا الاصـطلاح سـوى أنـه الاصـطلاح المتعـارف عليـه واعتـاد معظم المبرمجين عليه، وبالتالي فإن استخدامه يجعل قراءة شيفرة رست البرمجية والتعديل عليها أسهل.

الاستثناء لهذا الاصطلاح هو حالة إضافة عنصرين إلى النطاق يحملان الاسم ذاته باستخدام تعليمـة use، إذ لا تسمح رست بذلك. توضح الشيفرة 15 كيفية إضافة نوعَي Result إلى النطاق يحملان الاسـم ذاتـه ولكنهمـا ينتميان إلى وحدات أب مختلفة، إضافةً إلى كيفية الإشارة إليهما.

اسم الملف: src/lib.rs

```
use std::fmt;
use std::io;

fn function1() -> fmt::Result {
    // --snip--
    Ok(())
}
```

```
fn function2() -> io::Result<()> {
    // --snip--
    Ok(())
}
```

[الشيفرة 15: إضافة نوعين من الاسم ذاته إلى النطاق ذاته يتطلب استخدام الوحدات الأب]

كما تلاحظ، يميز استخدام الوحدات الأب ما بين النوعين `Result`، وإذا استخدمنا `use std::fmt::Result` و `use std::io::Result` بدلًا من ذلك فسيكون لدينا قيمتين `Result` في نفس النطاق، ولن تعرف رست عندها أي الأنواع التي نقصدها عندما نستخدم `Result`.

7.4.2 إضافة أسماء جديدة باستخدام الكلمة المفتاحية as

هناك حل آخر لمشكلة إضافة نوعين من الاسم ذاته إلى النطاق باستخدام `use`، إذ يمكننا كتابة الكلمة المفتاحية `as` بعد المسار وكتابة اسم محلي بعدها أو اسم مستعار alias للنوع. توضح الشيفرة 16 طريقةً أخرى لكتابة الشيفرة 15 بإعادة تسمية واحد من النوعَين `Result` باستخدام `as`.

اسم الملف: src/lib.rs

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
    // --snip--
    Ok(())
}

fn function2() -> IoResult<()> {
    // --snip--
    Ok(())
}
```

[الشيفرة 16: إعادة تسمية نوع بعد إضافته إلى النطاق باستخدام الكلمة المفتاحية as]

اخترنا في تعليمة `use` الثانية الاسم الجديد `IoResult` للنوع `std::io::Result` وهو اسم لا يتعارض مع الاسم `Result` في `std::fmt` الذي أضفناه إلى النطاق أيضًا. تعدّ كلًا من الشيفرة 15 والشيفرة 16 طريقةً اصطلاحيةً في التغلب على مشكلة تعارض الأسماء، فاختر ما يحلو لك.

7.4.3 إعادة تصدير الأسماء باستخدام pub use

عندما نُضيف اسمًا إلى النطاق باستخدام الكلمة المفتاحية use، سيكون هذا الاسم المتاح في النطاق الجديد خاصًا. لتمكين الشيفرة البرمجية التي تستدعي الشيفرة البرمجية لتُشير إلى ذلك الاسم وكأنه معرّف في نطاق الشيفرة البرمجية تلك، علينا استخدام pub مع use، وتُعرف هذه الطريقة باسم إعادة التصدير re-exporting لأننا نُضيف العنصر إلى النطاق، لكننا نجعله ممكن الإضافة لنطاقات أخرى.

تمثل الشيفرة 17 نسخةً معدلةً عن الشيفرة 11 بتغيير استخدام use في الوحدة الجذر إلى pub use.

اسم الملف: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

[الشيفرة 17: جعل الاسم متاحًا لأي شيفرة برمجية لاستخدامه من نطاق جديد باستخدام pub use]

كان على الشيفرة البرمجية الخارجية -قبل هذا التغيير- استدعاء الدالة add_to_waitlist باستخدام المسار restaurant::front_of_house::hosting::add_to_waitlist()، والآن وبعد إعادة تصدير وحدة hosting باستخدام pub use من الوحدة الجذر يُمكن للشيفرة الخارجية الآن أن تستخدم المسار restaurant::hosting::add_to_waitlist().

إعادة التصدير مفيد عندما يكون الهيكل الداخلي لشيفرتك البرمجية مختلفًا عن الطريقة التي قد يفكّر بها المبرمجون الآخرون باستدعاء شيفرتك البرمجية بحسب استخدامهم لها. على سبيل المثال وبالنظر إلى تشبيه المطعم، يفكر الأشخاص الذين يعملون بالمطعم بخصوص كل من الأشياء التي تحدث في واجهة المطعم وفي خلفية المطعم، إلا أن الزبائن الذي يزورون المطعم لن يفكرون غالبًا بخصوص الأشياء التي تحدث في خلفية المطعم ضمن هذا السياق. يمكننا كتابة شيفرتنا البرمجية بالاستعانة بالكلمتين pub use ضمن هيكل واحد مع كشف expose هيكل مختلف، وفعل ذلك يجعل من مكتبتنا منظّمة للمبرمجين الذي يعملون على المكتبة

وللمبرمجين الذين يستدعون المكتبة على حد سواء، وسننظر لمثال آخر يحتوي على `pub use` وكيف يؤثر على توثيق الوحدة المصرّفة لاحقًا.

7.4.4 استخدام الحزم الخارجية

برمجنا سابقًا لعبة تخمين الأرقام واستخدمنا فيها حزمةً خارجيةً تدعى `rand` للحصول على أرقـام عشـوائية، ولاستخدام `rand` في مشروعنا أضفنا السطر التالي إلى ملف Cargo.toml:

اسم الملف: Cargo.toml

```
rand = "0.8.3"
```

تخبر إضـافة `rand` مثـل اعتماديـة dependency في ملـف Cargo.toml كـارجو بأنـه يجب عليـه تنزيـل الحزمة `rand` وأي اعتمادية من crates.io وجعل الحزمة `rand` متاحة في مشروعنا.

أضفنا السطر `use` مع اسم الوحدة المُصرّفة لجلب التعاريف الموجودة في الحزمة `rand` إلى نطاق حزمتنـا، وأضفنا العناصـر الـتي أردنـا إضـافتها إلى النطـاق. تـذكر أننـا أضـفنا السـمة `Rng` إلى نطاقنـا واسـتدعينا الدالـة `rand::thread_rng`:

```
use rand::Rng;

fn main() {
    let secret_number = rand::thread_rng().gen_range(1..=100);
}
```

كتب الكثير من أعضاء مجتمع رست حزمًا وجعلها متاحة على crates.io، ويجب أن تتبع هذه الخطـوات إذا أردت استخدام أي منها: إضافة الحزم في الملف Cargo.toml مثل قائمة، واستخدام `use` لإضافة العناصـر من وحداتها المُصرّفة إلى النطاق.

لاحظ أن المكتبة القياسية `std` هي في الحقيقة وحدة مُصرَّفة خارجية لحزمتنـا، وليس علينـا تغيـير الملـف Cargo.toml لتضمين `std` لأن المكتبة القياسـية تُضـاف افتراضـيًا إلى مشـروعنا بلغـة رسـت، إلا أننـا بحاجـة إضافة عناصرها إلى نطاق حزمتنا باستخدام `use`. على سبيل المثال، يجب علينا كتابـة السـطر الـبرمجي التـالي لاستخدام `HashMap`:

```
use std::collections::HashMap;
```

وهذا يمثّل مسار مطلق يبدأ بالكلمة `std` وهو اسم وحدة مكتبة مُصرَّفة قياسي.

7.4.5 استخدام المسارات المتداخلة لتنظيم تعليمات use الطويلة

إذا كنا نستخدم عدة عناصر معرفة في الوحدة المصرّفة ذاتها أو الوحدة ذاتها، فكتابة كل واحدة منها على حدى في سطر خاص سيأخذ الكثير من المساحة ضمن ملفنا. على سبيل المثال انظر إلى تعليمات `use` التالية التي استخدمناها في برمجة لعبة التخمين التي تُضيف بعض العناصر من `std` إلى النطاق:

اسم الملف: src/main.rs

```
use std::cmp::Ordering;
use std::io;
```

عوضًا عن ذلك يمكننا استخدام المسارات المتداخلة nested paths لإضافة العناصر ذاتها إلى النطاق ضمن سطر واحد، ونفعل ذلك عن طريق تحديد الجزء المشترك من المسار متبوعًا بنقطتين مزدوجتين ومن ثم أقواس معقوصة حول لائحة من الأجزاء التي تختلف عن الجزء المشترك من المسار كما هو موضح في الشيفرة 18.

اسم الملف: src/main.rs

```
use std::{cmp::Ordering, io};
```

[الشيفرة 18: تحديد مسار متداخل لإضافة عدة عناصر إلى النطاق باستخدام مسار مشترك]

يُقلّل استخدام المسارات المشتركة عدد السطور اللازمة لإضافة بعض العناصر باستخدام تعليمة `use` في البرامج الكبيرة بصورةٍ ملحوظة.

يمكننا استخدام المسار المتداخل ضمن أي مستوى في المسار، وهو أمر مفيد عند دمج تعليمتَي `use` تتشاركان بمسار جزئي. توضح الشيفرة 19 مثلًا تعليمتَي `use`: أحدها تُضيف `std::io` إلى النطاق والأخرى تجلب `std::io::Write` إلى النطاق.

اسم الملف: src/lib.rs

```
use std::io;
use std::io::Write;
```

[الشيفرة 19: تعليمتَي use، تشكّل واحدة منهما مسارًا جزئيًا للآخر]

الجزء المشترك من المسارَين هو `std::io` وهذا هو المسار الكامل الأول، ولدمج المسارَين إلى تعليمة `use` واحدة يمكننا استخدام `self` في المسار المتداخل كما هو موضح في الشيفرة 20.

اسم الملف: src/lib.rs

```
use std::io::{self, Write};
```

[الشيفرة 20: دمج المسارَين في الشيفرة 19 إلى تعليمة use واحدة]

يُضيف السطر البرمجي السابق كلًا من `std::io` و `std::io::Write` إلى النطاق.

7.4.6 عامل تحديد الكل glob

يمكننا استخدام العامل * إذا أردنا إضافة جميع العناصر العلنية الموجودة في مسار ما إلى النطاق:

```
use std::collections::*;
```

تُضيف تعليمة `use` السابقة جميع العناصر العلنية المعرفة في المسار `std::collections` إلى النطاق الحالي. كُن حريصًا عن استخدامك لهذا العامل، لأنه قد يجعل من الصعب معرفة أي الأسماء الموجودة في النطاق وأين الأسماء المستخدمة في البرنامج والمعرفة داخله.

يُستخدم عامل تحديد الكل غالبًا عند تجربة إضافة كل شيء ضمن عملية تجربة إلى الوحدة `tests`، وسنتكلم لاحقًا عن كيفية كتابة هذا النوع من الوحدات. يُستخدم عامل تحديد الكل أيضًا في بعض الأحيان جزءًا من النمط التمهيدي prelude pattern، ألقِ نظرةً على توثيق المكتبة القياسية لمزيد من التفاصيل حول هذا النمط.

7.5 فصل الوحدات إلى ملفات مختلفة

لحدّ اللحظة، عرّفت الشيفرات البرمجية التي استعرضناها في الأمثلة عدة وحدات في ملف واحد، إلا أنك قد تكون بحاجة نقل تعاريف الوحدات إلى ملف منفصل في حال كان عدد الوحدات كبيرًا وذلك بهدف جعل شيفرتك البرمجية سهلة القراءة.

على سبيل المثال، دعنا نبدأ من الشيفرة البرمجية الموجودة في الشيفرة 17 التي تحتوي على عدّة وحدات للمطاعم، وسنستخرج هذه الوحدات إلى ملف مخصص بدلًا من تعريف كل من الوحدات في ملف جذر الوحدة المُصرّفة، وفي هذه الحالة ملف جذر الوحدة المُصرّفة هو src/lib.rs إلا أن هذه العملية تعمل أيضًا مع الوحدات الثنائية المُصرّفة التي يكون ملف ملف جذر الوحدة المُصرّفة الخاص بها هو src/main.rs.

دعنا نستخرج أولًا الوحدة `front_of_house` إلى ملفها الخاص، ونحذف الشيفرة البرمجية داخل الأقواس المعقوصة الخاصة بالوحدة `front_of_house` ونترك فقط تصريح `mod front_of_house;`، بحيث يحتوي src/lib.rs على الشيفرة البرمجية الموضحة في الشيفرة 21، ولاحظ أن الشيفرة لن تُصرَّف حتى نُنشئ ملف src/front_of_house.rs، وهو ما سنفعله في الشيفرة 22.

اسم الملف: src/lib.rs

```
mod front_of_house;

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

[الشيفرة 21: تصريح الوحدة front_of_house التي سيتواجد محتواها في الملف src/front_of_house.rs]

ننقل الشيفرة البرمجية التي كانت موجودة في الأقواس المعقوصة إلى ملف جديد يسمى src/front_of_house.rs كما هو موضح في الشيفرة 22. سيعلم المصرّف مكان هذا الملف لأنه سيصادف التصريح عن الوحدة `front_of_house` في جذر الوحدة المُصرّفة.

اسم الملف: src/front_of_house.rs

```
pub mod hosting {
    pub fn add_to_waitlist() {}
}
```

[الشيفرة 22: التعريف داخل الوحدة front_house ضمن الملف src/front_of_house.rs]

لاحظ أنك تستطيع تحميل ملف ما باستخدام تصريح `mod` مرةً واحدةً فقط ضمن شجرة الوحدة، وحالما يعلم المصرف أن هذا الملف ينتمي إلى المشروع (بالإضافة إلى معرفته لمكان الملف ضمن شجرة الوحدة بفضل المكان الذي كتبت فيه تعليمة `mod`)، ويجب أن تشير الملفات الأخرى إلى الشيفرة البرمجية الخاصة بالملف المُحمّل باستخدام مسار يشير إلى مكان التصريح عنه كما ناقشنا في الفقرات السابقة. بكلمات أخرى، لا تشبه `mod` عملية تضمين "include"، الموجودة في بعض لغات البرمجة الأخرى.

الآن، نستخرج الوحدة `hosting` إلى ملفها الخاص، إلا أن العملية مختلفة هنا بعض الشيء لأن `hosting` هي وحدة ابن من الوحدة `front_of_house` وليس وحدة ابن من الوحدة الجذر، ولذلك سنضع ملف الوحدة `hosting` في مسار جديد يُسمّى بحسب آباء الوحدة في شجرة الوحدة، وفي هذه الحالة سيكون المسار هو src/front_of_house/.

لنقل الوحدة `hosting` نعدّل الملف src/front_of_house.rs بحيث يحتوي على تصريح الوحدة `hosting` فقط:

اسم الملف: src/front_of_house.rs

```
pub mod hosting;
```

نُنشئ بعد ذلك مسار src/front_of_house وملف hosting.rs ليحتوي على التعريف الخاص بالوحدة `hosting`:

اسم الملف: src/front_of_house/hosting.rs

```
pub fn add_to_waitlist() {}
```

إذا وضعنا hosting.rs في المجلد src بدلًا من المجلد الحالي فإن المصرّف سيتوقع رؤية شيفرة hosting.rs في الوحدة `hosting` المصرح عنها في جذر الوحدة المُصرّفة وليس المصرح عنها وحدة ابن للوحدة `front_of_house`. تُملي قواعد المصرف بخصوص مسارات الملفات والوحدات أهمية تنظيمها بالنسبة لشجرة الوحدة الخاصة بالمشروع.

نقلنا الشيفرة البرمجية الخاصة بكل وحدة إلى ملف منفصل وبقيت شجرة الوحدة على حالها، وستعمل استدعاءات الدالة في `eat_at_restaurant` دون أي تعديلات على الرغم من أن التعريفات موجودة في ملفات مختلفة، وتسمح لنا هذه الطريقة بتحريك الوحدات إلى ملفات جديدة بسهولة مع نموّ حجم مشروعك.

لاحظ أن تعليمة `pub use crate::front_of_house::hosting` الموجودة في src/lib.rs لم تتغير، إذ لا تؤثّر `use` على الملفات التي ستُصرَّف مثل جزء من الوحدة المُصرّفة. تعرّف الكلمة المفتاحية `mod` الوحدات، إلا أن رست تنظر إلى الملف الذي يحمل الاسم ذاته الخاص بالوحدة للبحث عن الشيفرة البرمجية الخاصة بتلك الوحدة.

7.5.1 مسارات ملفات مختلفة

غطّينا بحلول هذه النقطة مسارات الملفات الشائعة والاصطلاحية في مصرّف رست، إلا أن رست تدعم أيضًا بعض التنسيقات القديمة لمسارات الملفات، فبالنسبة لوحدة تدعى `front_of_house` مصرح عنها في جذر الوحدة المُصرّفة، ينظر المصرف للشيفرة البرمجية الخاصة بالوحدة في:

- المسار src/front_of_house.rs (وهو مسار تكلمنا عنه سابقًا).
- المسار src/front_of_house/mod.rs (مسار قديم إلا أنه مدعوم).

سينظر المصرف بالنسبة لوحدة جزئية من `front_of_house` تدعى `hosting` إلى اللشيفرة البرمجية الخاصة بالوحدة في:

- المسار src/front_of_house/hosting.rs (مسار ناقشناه سابقًا).

- المسار src/front_of_house/hosting/mod.rs (مسار قديم إلا أنه مدعوم).

إذا استخدمت كلا الأسلوبين للوحدة ذاتها، ستحصل على خطأ من المصرّف، إلا أن استخدام كلا النوعين في ذات المشروع لوحدات مختلفة مسموح، ولكنه قد يتسبب بالخلط لبعض الناس الذين يقرؤون الشيفرة البرمجية.

النقطة السلبية الأساسية للأسلوب الذي يستخدم ملفات بالاسم mod.rs هو انتهاء مشروعك غالبًا باحتوائه على كثير من الملفات تحمل الاسم mod.rs، وهذا قد يثير الإرباك عندما تحاول تعديل الشيفرة البرمجية ضمن محرر ما.

7.6 خاتمة

تسمح لك رست بتجزئة الحزمة إلى عدة وحدات مُصرّفة، وكل وحدة مُصرّفة إلى وحدات لتمكّنك من الإشارة إلى العناصر المعرّفة في وحدة ما ضمن وحدة أخرى. يمكنك تحقيق ما سبق عن طريق تحديد المسار المطلق أو النسبي، ويمكن إضافة هذه المسارات إلى النطاق باستخدام تعليمة use بحيث يمكنك استخدام مسارات أقصر لعدة استخدامات لهذا العنصر في النطاق ذاته. الشيفرة البرمجية الخاصة بالوحدة هي خاصة افتراضيًا إلا أنه من الممكن جعلها عامة باستخدام الكلمة المفتاحية pub.

سنتعرف في الفصل التالي على بعض هياكل البيانات الخاصة بالتجميعات collections في المكتبة القياسية التي يمكنك استخدامها لتنظيم شيفرتك البرمجية بصورةٍ أفضل.

هل تطمح لبيع منتجاتك الرقمية عبر الإنترنت؟

استثمر مهاراتك التقنية وأطلق منتجًا رقميًا
يحقق لك دخلًا عبر بيعه على متجر بيكاليكا

أطلق منتجك الآن

8. التجميعات الشائعة

تحتـوي مكتبـة رسـت القياسـية على عـدد من هياكـل البيانـات data structures المفيـدة الـتي تـدعى بالتجميعات collections، وبينما تمثل معظم أنواع البيانات الأخرى قيمةً واحدةً معينة، تحتوي التجميعات على عدّة قيم مختلفة، وعلى عكس المصفوفات والأصناف tuples، تُخزّن بيانات التجميعات في الكومة heap ممـا يعني أنه ليس من الضـروري أن يكـون حجم البيانـات معلومًـا وقت التصـريف ويمكن أن يـزداد أو ينقص خلال تشغيل البرنامج. يمتلك كل نوع من أنواع التجميعات خصائص وحدود معينة، ويتطلب اختيـار النـوع المناسـب لحالتك مهارةً تطوّرها بمرور الوقت. سنناقش في هذا الفصل ثلاثة أنواع تجميعات شائعة الاسـتخدام في بـرامج رست:

- الشعاع vector، الذي يسمح لك بتخزين عدد متغير من القيم بصورةٍ متتالية.
- السلسلة النصية string، وهي تجميعة من المحارف وقد ذكرناها سابقًا عندما ناقشنا النوع `String` إلا أننا سنناقشها في هذا الفصل بتعمُّق أكبر.
- خارطة تعمية Hash map، وتسمح لك بربط قيمة مـا مـع مفتـاح key، وهي تطـبيق لهيكـل البيانـات الأكثر عمومًا وهو الخارطة.

للاطّلاع على أنواع أخرى من التجميعات المتوفرة في المكتبة القياسية ألقِ نظرةً على الرابط هنا.

سنناقش كيفية إنشاء وتحديث كل من الأشعة والسلاسل النصية وخارطات التعمية، إضافةً إلى ميزات كـل منها.

8.1 تخزين لائحة من القيم باستخدام الأشعة Vectors

سننظر أولًا إلى نوع التجميعة `Vec<T>`، المعروف أيضًا باسم الشعاع vector، إذ تسمح لك الأشعة بتخزين أكثر من قيمة واحدة في هيكل بيانات واحد يضع القيم على نحوٍ متتالي في الذاكرة، ويمكن أن تخزن الأشعة قيمًا من النوع ذاته، وهي مفيدةٌ عندما يكون لديك لائحةٌ من العناصر، مثل سطور نصية ضمن ملف، أو أسعار منتجات في سلة تسوّق.

8.1.1 إنشاء شعاع جديد

لإنشاء شعاع جديد فارغ نستدعي الدالة `Vec::new` كما هو موضح في الشيفرة 1.

```
let v: Vec<i32> = Vec::new();
```

[الشيفرة 1: إنشاء شعاع جديد فارغ لتخزين قيم من النوع i32]

لاحظ أننا كتبنا ما يشير إلى نوع البيانات، لأننا لا نستطيع إدخال أي نوع نريده في الشعاع، ويجب على رست أن تعلم أي نوع من البيانات نريد أن ندخله إلى الشعاع، وهذه النقطة مهمة جدًا. بُنيَت الأشعة باستخدام الأنواع المعمّاة generics وسنغطّي استخدام الأنواع المعمّاة مع أنواعك الخاصة لاحقًا، ويكفي حاليًا أن تعرف أن النوع `Vec<T>` الموجود في المكتبة القياسية يستطيع تخزين أي نوع داخله. يمكننا تحديد النوع الذي يحمله الشعاع عند إنشائه دون استخدام الأقواس المثلثة angle brackets. أخبرنا رست في الشيفرة 1 أن `Vec<T>` في `v` يحمل عناصر من النوع `i32`.

ستُنشئ معظم الأحيان شعاع `Vec<T>` يحتوي على قيم ابتدائية ويستنتج رست نوع البيانات التي تريد أن تخزنها داخل الشعاع من القيم الابتدائية، لذا يُعد استخدام الطريقة السابقة نادرًا. توفر لنا رست الماكرو `vec!` الذي يُنشئ شعاعًا جديدًا يحتوي على القيم التي تمررها له، وتوضح الشيفرة 2 ذلك بإنشاء شعاع من النوع `Vec<i32>` يحتوي على القيم 1 و2 و3، والنوع هو `i32` لأنه النوع الافتراضي للأعداد الصحيحة كما ناقشنا سابقًا في الفصل 3.

```
let v = vec![1, 2, 3];
```

[الشيفرة 2: إنشاء شعاع جديد يحتوي على قيم]

تستطيع رست استنتاج أن نوع الشعاع `v` هو `Vec<i32>`، لأننا أعطينا قيمًا ابتدائية من النوع `i32`، وكتابة النوع مباشرةً ليس ضروريًا هنا. الآن دعنا ننظر إلى كيفية التعديل على شعاع.

8.1.2 تحديث شعاع

يمكننا استخدام التابع `push` لإضافة عناصر إلى شعاع بعد إنشائه كما هو موضح في الشيفرة 3.

```
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

[الشيفرة 3: استخدام التابع push لإضافة قيم إلى شعاع]

إذا أردنا تغيير القيم، علينا أن نجعل الشعاع قابلًا للتعديل -كما هو الحال مع أي متغير اعتيادي- باستخدام الكلمة المفتاحية `mut` كما ناقشنا سابقًا. جميع الأرقام التي وضعناها في الشعاع هي من النوع `i32`، وتستنتج رست ذلك من البيانات، لذلك ليس من الضروري تحديد النوع بكتابة `Vec<i32>`.

8.1.3 قراءة العناصر من الأشعة

هناك طريقتان للإشارة إلى قيمة موجودة في الشعاع: إما باستخدام الدليل index أو باستخدام التابع `get`، نوضح في الأمثلة التالية أنواع البيانات التي تُعيدها الدوال بهدف جعلها واضحةً قدر الإمكان.

توضح الشيفرة 4 كلا الطريقتين في الوصول إلى قيمة ضمن شعاع، وذلك باستخدام دليل العنصر أو التابع `get`.

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
println!("The third element is {}", third);

let third: Option<&i32> = v.get(2);
match third {
    Some(third) => println!("The third element is {}", third),
    None => println!("There is no third element."),
}
```

[الشيفرة 4: استخدام دليل العنصر أو التابع get للحصول على عنصر ضمن الشعاع]

لاحظ بعض التفاصيل المهمة هنا؛ إذ استخدمنا قيمة الدليل "2" للحصول على العنصر الثالث وذلك لأن العناصر في الشعاع تحمل دليل عددي يبدأ من الصفر، كما يعطي استخدام `&` و`[]` مرجعًا إلى العنصر الموجود في الدليل الذي حددناه. عندما نستخدم التابع `get` مع تمرير الدليل مثل وسيط argument نحصل على `Option<&T>` الذي يمكننا استخدامه مع البنية `match`.

توفّر رست طريقتين مختلفتين ليتسنى لك اختيار تصرف برنامجك عندما تحاول استخدام قيمة دليل خارج نطاق العناصر الموجودة في الشعاع، على سبيل المثال دعنا ننظر إلى ما يحدث عندما يكون لدينا شعاع من خمسة عناصر ونحاول الوصول إلى العنصر ذو الدليل 100 باستخدام كلا الطريقتين كما هو موضح في الشيفرة 5.

```
let v = vec![1, 2, 3, 4, 5];

let does_not_exist = &v[100];
let does_not_exist = v.get(100);
```

[الشيفرة 5: محاولة الوصول إلى الدليل 100 في شعاع يحتوي على خمسة عناصر فقط]

ستتسبب الطريقة الأولى `[]` بهلع panic البرنامج عند تشغيل الشيفرة البرمجية السابقة، لأنها تحاول لإشارة إلى عنصر غير موجود، وهذه الطريقة هي الأفضل في حال أردت لبرنامجك أن يتوقف عن العمل في حال محاولتك الوصول إلى عنصر يقع بعد نهاية الشعاع.

عندما نمرر إلى التابع `get` دليلًا يقع خارج المصفوفة فهو يُعيد القيمة `None` دون الهلع، ويجب أن تستخدم هذه الطريقة إذا كانت محاولة الوصول إلى عنصر يقع خارج نطاق الشعاع ممكنة الحدوث تحت الظروف الاعتيادية، ويجب على برنامجك عندها أن يتعامل مع `Some(&element)` أو `None` كما ناقشنا في الفصل السادس. على سبيل المثال، قد يكون الدليل مُدخلًا من قِبل المستخدم وبالتالي إذا أدخل المستخدم رقمًا أكبر من حجم الشعاع عن طريق الخطأ نحصل على القيمة `None` ويمكنك إخبار المستخدم عندها أن الرقم الذي أدخله كبير ويمكن إعلامه أيضًا بحجم الشعاع الحالي وإعادة طلب إدخال القيمة منه، وهذه وسيلة عملية أكثر من جعل البرنامج يتوقف بالكامل بسبب خطأ كتابي بسيط

عندما نحصل على مرجع صالح، يتأكد مدقق الاستعارة borrow checker من أن قوانين الملكية ownership والاستعارة borrowing محققة (ناقشنا هذه القوانين سابقًا في الفصل 4) للتأكد من أن المرجع وأي مراجع أخرى لمحتوى الشعاع هي مراجع صالحة. تذكر القاعدة التي تنص على أنه لا يمكنك الحصول على مرجع قابل للتعديل ومرجع آخر غير قابل للتعديل في النطاق ذاته، وتنطبق هذه القاعدة على الشيفرة 6 عندما نخزن مرجعًا غير قابلٍ للتعديل للعنصر الأول في الشعاع ومن ثم نجرّب إضافة عنصر إلى نهاية الشعاع، ولن يعمل هذا البرنامج إذا أردنا الإشارة إلى ذلك العنصر لاحقًا ضمن الدالة ذاتها:

```
let mut v = vec![1, 2, 3, 4, 5];

let first = &v[0];

v.push(6);
```

```
    println!("The first element is: {}", first);
```

[الشيفرة 6: محاولة إضافة عنصر إلى شعاع مع تخزين مرجع إلى عنصر داخل الشعاع في الوقت ذاته]

سيتسبب تصريف الشيفرة البرمجية السابقة بالخطأ التالي:

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
error[E0502]: cannot borrow `v` as mutable because it is also borrowed
as immutable
 --> src/main.rs:6:5
  |
|     let first = &v[0];
  |                 - immutable borrow occurs here
|
|     v.push(6);
  |     ^^^^^^^^^ mutable borrow occurs here
|
|     println!("The first element is: {}", first);
  |                                         ----- immutable borrow
later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `collections` due to previous error
```

قد تبدو الشيفرة 6 صالحة ويجب أن تعمل. لماذا يتعلق مرجع العنصر الأول بالتغييرات الحاصلة بنهاية الشعاع؟ لنفهم الخطأ يجب أن نفهم كيف تعمل الأشعة، إذ تضع الأشعة القيم بصورةٍ متتالية في الذاكرة وبالتالي يتطلب إضافة عنصر جديد إلى نهاية الشعاع حجز ذاكرة جديدة ونسخ العناصر القديمة في الشعاع إلى مساحة جديدة إذا لم يكن هناك مساحةٌ كافية لوضع جميع العناصر في الشعاع على نحوٍ متتالي، وسيشير مرجع العنصر الأول في تلك الحالة إلى ذاكرة مُحرَّرة deallocated، وتمنع قوانين الاستعارة البرامج من التسبب بهذا النوع من الأخطاء قبل حدوثها.

> للمزيد من التفاصيل حول النوع Vec<T> ألقِ نظرةً على إلى مثال عن تنفيذ Vec في Rustonomicon.

8.1.4 الوصول إلى قيم شعاع متعاقبة

للوصول إلى عناصر الشعاع بصورةٍ متعاقبة، نمرّ بجميع العناصر الموجودة دون الحاجة لاستخدام دليل كل عنصر في كل مرة، وتوضح الشيفرة 7 كيفية استخدام الحلقة `for` للحصول على مراجع غير قابلة للتعديل لعناصر الشعاع من النوع `i32` وطباعتها.

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}
```

[الشيفرة 7: طباعة كل عنصر في شعاع عن طريق المرور على العناصر باستخدام حلقة for]

يمكننا أيضًا المرور على مراجع قابلة للتعديل mutable لكل عنصر في شعاع قابل للتعديل، وذلك بهدف إجراء تغييرات على جميع العناصر. تجمع الحلقة `for` في الشيفرة 8 القيمة 50 إلى كل عنصر في الشعاع.

```
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

[الشيفرة 8: المرور على مراجع قابلة للتعديل لعناصر في شعاع]

`*` dereference لتغيير قيمة العنصر الذي يشير المرجع القابل للتعديل إليه نستخدم عامل التحصيل للحصول على القيمة الموجودة في `i` قبل استخدام العامل `+=`، وسنناقش عامل التحصيل بتوسع أكبر لاحقًا.

المرور على عناصر الشعاع عملية آمنة، سواءً كان ذلك الشعاع قابلًا للتعديل أو غير قابل للتعديل وذلك بسبب قوانين مدقق الاستعارة، فإذا حاولنا إدخال أو إزالة العناصر ضمن الحلقة `for` في الشيفرة 7 أو الشيفرة 8، فسنحصل على خطأ تصريفي مشابه للخطأ الذي حصلنا عليه عند تصريفنا للشيفرة 6، إذ يمنع المرجع الذي يشير إلى الشعاع ضمن الحلقة `for` أي تعديلات متزامنة لكامل الشعاع.

8.1.5 استخدام تعداد لتخزين عدة أنواع

يمكن للأشعة أن تخزن قيمًا من النوع ذاته فقط، وقد يشكّل هذا عقبةً صغيرةً، إذ أن هناك الكثير من الاستخدامات التي نريد فيها تخزين لائحة من عناصر من أنواع مختلفة، ولحسن الحظ المتغايرات variants الخاصة بالتعداد enum معرفة تحت نوع التعداد ذاته وبالتالي يمكننا استخدام نوع واحد لتمثيل عناصر من أنواع مختلفة بتعريف واستخدام تعداد.

على سبيل المثال، لنفرض أننا بحاجة للحصول على قيم من صف ضمن جدول، ويحتوي هذا الصف قيمًا من نوع الأعداد الصحيحة وقيم أعداد عشرية floating point وقيم سلاسل نصية strings، حينها يمكننا تعريف تعداد يحتوي على متغايرات يمكنها تخزين أنواع القيم المختلفة وسيُنظر إلى جميع متغايرات التعداد إلى أنها من النوع ذاته الخاص بالتعادد، يمكننا بعد ذلك إنشاء شعاع يخزّن ذلك التعداد، وبالتالي سيخزّن أنواعًا مختلفة، وقد وضحنا هذه العملية في الشيفرة 9.

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

[الشيفرة 9: تعريف تعداد enum لتخزين قيم من أنواع مختلفة في شعاع واحد]

تحتاج رست إلى معرفة الأنواع التي ستُخزَّن في الشعاع وقت التصريف حتى تعلم بالضبط المساحة التي ستحتاجها لتخزين كل عنصر في الكومة، ويجب علينا أن نكون واضحين بخصوص الأنواع التي ستُخزَّن في الشعاع. في الحقيقة إذا سمحت رست للشعاع بأن يحمل أي نوع فهناك احتمال أن يتسبب نوع أو أكثر بالأخطاء عند إجراء العمليات على عناصر الشعاع، واستخدام التعداد مع التعبير `match` يعني أن رست ستعالج كل حالة بوقت التصريف كما ناقشنا سابقًا في الفصل السادس.

لن تعمل طريقة التعداد هذه إذا لم تُعرّف جميع أنواع البيانات التي سيحتاجها البرنامج عند وقت التشغيل، ويمكنك استخدام كائن السمة trait object عندها، والذي سنتكلم عنه لاحقًا.

الآن، وبعد أن ناقشنا أكثر الطرق شيوعًا في استخدام الأشعة، ألقِ نظرةً على توثيق الواجهة البرمجية للمزيد من التوابع المفيدة المعرفة في النوع `Vec<T>` في المكتبة القياسية، على سبيل المثال، بالإضافة إلى تابع `push`، هناك تابع `pop` لحذف وإعادة العنصر الأخير ضمن الشعاع.

8.1.6 التخلص من الشعاع يعني التخلص من عناصره

يُحرَّر الشعاع من الذاكرة مثل أيّ هيكل `struct` اعتيادي عند خروجه من النطاق كما هو موضح في الشيفرة 10.

```
    {
        let v = vec![1, 2, 3, 4];

        // استخدام v
    } // تخرج v من النطاق وتُحرَّر من الذاكرة بحلول هذه النقطة
```

[الشيفرة 10: توضيح النقطة التي يُحرَّر فيها الشعاع ونفقد عناصره]

عندما تُحرَّر الذاكرة الخاصة بالشعاع، هذا يعني أننا نفقد عناصر أيضًا، أي أننا لن نستطيع الوصول إلى الأعداد الصحيحة في الشيفرة السابقة بعد خروج الشعاع من النطاق. يتأكد مدقق الاستعارة من أن جميع المراجع التي تشير إلى محتوى الشعاع -إن وُجدت- تُستخدَم فقط عندما يكون الشعاع بنفسه صالحًا.

دعنا ننتقل إلى نوع التجميعة التالي: ألا وهو السلسلة النصية `String`.

8.2 تخزين النصوص بترميز UTF-8 داخل السلاسل النصية

أتينا على ذكر السلاسل النصية سابقًا إلا أننا لم ننظر إليها بالتفصيل بعد، إذ يجد متعلمو لغة رست فهم السلاسل النصية صعبًا لثلاثة أسباب رئيسة: ميل رست لاستباق الخطأ قبل حدوثه وعرض رسالة خطأ تدلّ عليه، كما يبدو هيكل بيانات السلاسل النصية أكثر صعوبةً وتعقيدًا مما تبدو عليه، وأخيرًا ترميز UTF-8. قد تجتمع جميع الأسباب الثلاث السابقة وتجعل من الصعب عليك فهم السلاسل النصية إن قدمت من لغات برمجة مختلفة.

نناقش السلاسل النصية هنا في سياق التجميعات collections، وذلك لأن السلاسل النصية هي تطبيق لتجميعة من البايتات إضافةً إلى بعض التوابع المفيدة الأخرى، والتي تبرز أهميتها عندما تمثّل هذه البايتات نصًا ما. سنتحدث في هذا الفصل عن العمليات على السلاسل النصية `String` الموجودة في كل نوع تجميعة، مثل إنشاء سلسلة نصية والتعديل عليها وقراءة محتوياتها، كما سنناقش أيضًا الطرق التي تختلف فيها السلاسل النصية عن التجميعات الأخرى وبالأخص استخدام دليل السلسلة النصية للوصول إلى محتوياتها، فهو معقد نظرًا لاختلافه مع ما يفسّره البشر لبيانات سلسلة نصية وما تفسره الحواسيب.

8.2.1 ما هي السلسلة النصية؟

دعنا نبدأ أولًا بتعريف ما الذي نقصده عندما نقول سلسلة نصية؛ إذ تمتلك رست نوع سلسلة نصية واحد في أساس اللغة وهو شريحة السلسلة النصية string slice `str`، الذي نراه عادةً بشكله المختصر `&str`. تحدثنا سابقًا الفصل 4 عن شرائح السلاسل النصية؛ وهي مراجع إلى بعض بيانات السلاسل النصية المرمزة بترميز UTF-8 والمخزنة في مكان آخر. على سبيل المثال، تُخزَّن السلاسل النصية المجرّدة string literals في ملف البرنامج الثنائي وبالتالي فهي شرائح سلسلة نصية.

النوع `String` الموجود في مكتبة رست القياسية بدلًا من وجوده في أساس اللغة، هو نوع سلسلة نصية قابل للتعديل mutable وللنمو growable والامتلاك owned ومُرمَّز بترميز UTF-8. عندما يقول مبرمجو لغة رست "سلسلة نصية" في رست فهم يقصدون إما النوع `String` أو أنواع شريحة السلسلة النصية `&str` ولا يقتصر الأمر على واحد من النوعَين. على الرغم من أن هذا الفصل يتكلم خاصةً على النوع `String` إلا أن كلا النوعين مُستخدمان جدًا في مكتبة رست القياسية، وكلٌ من النوع `String` وشريحة السلسلة النصية مُرمّزَان بترميز UTF-8.

8.2.2 إنشاء سلسلة نصية جديدة

الكثير من العمليات المتاحة في النوع `Vec<T>` هي عمليات متاحة في النوع `String` أيضًا، وذلك لأن النوع `String` هو تطبيق لمُغلَّف wrapper حول شعاع من البايتات، إضافةً إلى بعض الضمانات والقيود والإمكانات الأخرى. الدالة `new` هي مثال على دالة تعمل بنفس الطريقة على `Vec<T>` وعلى `String` سويًا لإنشاء نسخة instance كما هو موضح في الشيفرة 11.

```
let mut s = String::new();
```

[الشيفرة 11: إنشاء نسخة جديدة وفارغة من النوع String]

يُنشئ السطر السابق سلسلةً نصيةً جديدةً وفارغة بالاسم `s`، وبالتالي يمكننا الآن إضافة البيانات إليها، وسنحتاج غالبًا في البداية إلى بيانات تهيئة لتخزينها في السلسلة النصية، ونستخدم لهذا الغرض التابع `to_string`؛ وهو تابع متاح في أي نوع يطبّق السمة `Display` وهو ما تفعله السلسلة النصية المجردة، توضح الشيفرة 12 مثالين على ذلك.

```
    let data = "initial contents";

    let s = data.to_string();

    //تعمل هذه الدالة على السلاسل النصية المجردة مباشرةً
    let s = "initial contents".to_string();
```

[الشيفرة 12: استخدام التابع to_string لإنشاء النوع String من سلسلة نصية مجردة]

تُنشئ الشيفرة البرمجية السابقة سلسلة نصية تحتوي على بيانات تهيئة `initial contents`.

يمكننا أيضًا استخدام الدالة `String::from` لإنشاء النوع `String` من سلسلة نصية مجردة، والشيفرة 13 مكافئة للشيفرة 12 التي تستخدم التابع `to_string`.

```
let s = String::from("initial contents");
```

[الشيفرة 13: استخدام الدالة String::from لإنشاء النوع String من سلسلة نصية مجردة]

بما أن السلاسل النصية تُستخدم للعديد من الأشياء، يمكننا استخدام عدة واجهات برمجية عامة لها، إذ تزودنا هذه الواجهات بكثيرٍ من الخيارات، وقد يبدو بعضها مكرّرًا إلا أن جميعها تؤدي غرضًا ما. في هذه الحالة، يؤدي كلًا من `String::from` و `to_string` الغرض ذاته، واستخدام أي منهما يعود إلى أسلوبك في كتابة الشيفرات البرمجية.

تذكر أن السلاسل النصية تستخدم ترميز UTF-8 وهذا يعني أنه يمكننا تضمين أي بيانات بهذا الترميز، ألقِ نظرةً على الشيفرة 14.

```
let hello = String::from("السلام عليكم");
let hello = String::from("Dobrý den");
let hello = String::from("Hello");
let hello = String::from("שלום");
let hello = String::from("नमस्ते");
let hello = String::from("こんにちは");
let hello = String::from("안녕하세요");
let hello = String::from("你好");
let hello = String::from("Olá");
let hello = String::from("Здравствуйте");
let hello = String::from("Hola");
```

[الشيفرة 14: تخزين رسالة تحية بلغات مختلفة في سلاسل نصية]

تحتوي جميع السلاسل النصية السابقة قيمًا صالحة من النوع `String`.

8.2.3 تحديث سلسلة نصية

يمكن أن يكبر حجم النوع `String` وأن تتغير محتوياته بصورةٍ مشابهة للنوع `Vec<T>` عند إضافة مزيدٍ من البيانات إليه، ويمكنك إضافةً إلى ذلك استخدام العامل `+` أو الماكرو `format!` لضمّ concatenate عدّة قيم من النوع `String`.

ا. إضافة قيم إلى السلسلة النصية باستخدام push_str و push

يمكننا توسعة حجم النوع `String` باستخدام التابع `push_str` لإضافة شريحة سلسلة نصية كما هو موضح في الشيفرة 15.

```
let mut s = String::from("foo");
s.push_str("bar");
```

[الشيفرة 15: إضافة شريحة سلسلة نصية إلى النوع String باستخدام التابع push_str]

ستحتوي السلسلة النصية `s` -بعد السطرين البرمجيين السابقين- على `foobar`، إذ يأخذ التابع `push_str` شريحة سلسلة نصية لأننا لسنا بحاجة لأخذ ملكية المعامل، على سبيل المثال نريد في الشيفرة 16 أن نكون قادرين على استخدام `s2` بعد إضافة محتوياته إلى `s1`.

```
    let mut s1 = String::from("foo");
    let s2 = "bar";
    s1.push_str(s2);
    println!("s2 is {}", s2);
```

[الشيفرة 16: استخدام شريحة سلسلة نصية بعد إضافة محتوياتها إلى النوع String]

إذا أخذ التابع `push_str` ملكية `s2`، فلن نكون قادرين على طباعة قيمتها في السطر الأخير، إلا أن الشيفرة البرمجية هنا تعمل كما هو متوقع منها.

يأخذ التابع `push` محرفًا وحيدًا مثل معامل ويُضيفه إلى النوع `String`، ونُضيف في الشيفرة 17 الحرف "I" إلى النوع `String` باستخدام التابع `push`.

```
    let mut s = String::from("lo");
    s.push('l');
```

[الشيفرة 17: إضافة محرف واحد إلى قيمة من النوع String باستخدام push]

ستحتوي السلسلة النصية `s` نتيجةً لما سبق على `lol`.

ب. ضم السلاسل النصية باستخدام العامل + أو الماكرو format!

ستحتاج غالبًا إلى ضم سلسلتين نصيتين، وإحدى الطرق لتحقيق ذلك هو باستخدام العامل `+` كما هو موضح في الشيفرة 18.

```
fn main() {
    let s1 = String::from("Hello, ");
    let s2 = String::from("world!");
    let s3 = s1 + &s2; // لاحظ أن s1 نُقلَت إلى هنا ولا يمكن استخدامها بعد الآن
}
```

[الشيفرة 18: استخدام العامل + لضم قيمتين من النوع String إلى قيمة أخرى جديدة من النوع String]

ستحتوي السلسلة النصية s3 بعد تنفيذ الشيفرة السابقة على "Hello, world!"، والسبب في عدم كون s1 صالحة بعد عملية الجمع إلى استخدامنا مرجع إلى s2 يعود إلى بصمة التابع method signature الذي استدعيناه عندما استخدمنا العامل +، إذ يستخدم هذا العامل التابع add وتبدو بصمته كما يلي:

```
fn add(self, s: &str) -> String {
```

ستجد التابع add معرفًا في المكتبة القياسية باستخدام الأنواع المعمّاة generics والأنواع المترابطة associated types، إلا أننا استبدلنا هذه الأنواع بأنواع فعلية وهو ما يحدث عند استدعاء هذا التابع بقيمٍ من النوع String (سنناقش الأنواع المعمّاة لاحقًا)، تعطينا بصمة التابع أدلة يجب علينا فهمها لفهم العامل +.

أولًا، لدى s2 الرمز & أي أننا نُضيف مرجعًا للسلسلة النصية الثانية إلى السلسلة النصية الأولى، وهذا بسبب المعامل s في الدالة، إذ يمكننا جمع &str إلى النوع String فقط وليس بإمكاننا إضافة قيمتين من النوع String سويًا، ولكن تمهّل، نوع &s2 هو &String وليس &str كما هو موضح في المعامل الثاني للتابع add. إذًا، لمَ تُصرَّف الشيفرة 18 بنجاح؟

السبب في كوننا قادرين على استخدام &s2 في استدعاء add هو أن المصرف هنا قادرٌ على تحويل الوسيط &String قسريًا إلى &str، وعند استدعاء للتابع add، تستخدم رست **التحصيل القسري deref corecion** الذي يحوّل &s2 إلى &s2[..] (سنناقش التحصيل القسري بمزيدٍ من التفاصيل لاحقًا)، ولأن add لا يأخذ ملكية المعامل s السلسلة s2، ستظل قيمةً صالحةً من النوع String بعد هذه العملية.

ثانيًا، يمكننا في بصمة التابع رؤية أن add يأخذ ملكية self، لأن self لا تحتوي على الرمز &، وهذا يعني أن السلسلة s1 في الشيفرة 18 ستُنقل إلى استدعاء add ولن تصبح قيمةً صالحةً بعد ذلك، لذلك يبدو السطر البرمجي let s3 = s1 + &s2; بأنه ينسخ كلا السلسلتين النصيتين ويُنشئ سلسلةً جديدةً، إلا أنه في الحقيقة يأخذ ملكية s1 ويُسند نسخةً من محتويات s2 إلى نهايتها ومن ثم يُعيد ملكية النتيجة؛ أي بكلمات أخرى يبدو أن السطر البرمجي يُنشئ كثيرًا من النُسخ إلا أنه في حقيقة الأمر لا يفعل ذلك، وهذا التطبيق فعال أكثر من النسخ.

يصبح سلوك العامل + غير عملي في حال أردنا ضمّ عدة سلاسل نصية في نفس الوقت:

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = s1 + "-" + &s2 + "-" + &s3;
```

بحلول هذه النقطة، ستكون قيمة السلسلة s هي "tic-tac-toe"، إلا أن الأمر صعب المعرفة فورًا باستخدام محارف + و ". يمكننا استخدام الماكرو format! لعمليات ضم السلاسل النصية الأكثر تعقيدًا:

```
    let s1 = String::from("tic");
    let s2 = String::from("tac");
    let s3 = String::from("toe");

    let s = format!("{}-{}-{}", s1, s2, s3);
```

نحصل على القيمة "tic-tac-toe" في السلسلة `s` أيضًا بتنفيذ الشيفرة السابقة، إذ يعمل الماكرو `format!` على نحوٍ مماثل لعمل الماكرو `println!` إلا أنه يُعيد قيمةً من النوع `String` بدلاً من طباعة الخرج على الشاشة. نلاحظ أن الشيفرة البرمجية التي تستخدم الماكرو `format!` أسهل قراءةً من سابقتها. تستخدم الشيفرة المولّدة عن طريق استخدام الماكرو `format!` المراجع، لذلك لن يتسبب استدعائها بأخذ ملكية أي من معاملاتها.

8.2.4 الحصول على محتويات السلسلة النصية باستخدام الدليل

يمكننا الوصول إلى محارف السلسلة النصية في العديد من لغات البرمجة باستخدام دليلها index، وهي عمليةٌ صالحةٌ وشائعة في هذه اللغات، إلا أنك ستحصل على خطأ في رست إذا حاولت الوصول إلى أجزاء من النوع `String` باستخدام نفس الأسلوب. ألقِ نظرةً على الشيفرة البرمجية غير الصالحة في الشيفرة 19.

```
    let s1 = String::from("hello");
    let h = s1[0];
```

[الشيفرة 19: محاولة الوصول إلى أجزاء من السلسلة النصية باستخدام دليلها]

ستتسبب الشيفرة البرمجية السابقة بظهور الخطأ التالي:

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
error[E0277]: the type `String` cannot be indexed by `{integer}`
 --> src/main.rs:3:13
  |
|     let h = s1[0];
  |             ^^^^^ `String` cannot be indexed by `{integer}`
  |
  = help: the trait `Index<{integer}>` is not implemented for `String`
  = help: the following other types implement trait `Index<Idx>`:
            <String as Index<RangeFrom<usize>>>
            <String as Index<RangeFull>>
            <String as Index<RangeInclusive<usize>>>
```

```
            <String as Index<RangeTo<usize>>>
            <String as Index<RangeToInclusive<usize>>>
            <String as Index<std::ops::Range<usize>>>
            <str as Index<I>>

For more information about this error, try `rustc --explain E0277`.
error: could not compile `collections` due to previous error
```

تخبرنا رسالة الخطأ بالآتي: لا تدعم السلاسل النصية في رست الفهرسة indexing، ولكن لماذا؟ للإجابة على هذا السؤال دعنا نناقش كيف تخزّن رست السلاسل النصية في الذاكرة.

ا. التمثيل الداخلي

النوع `String` في الحقيقة هو مغلّف حول النوع `Vec<u8>`، دعنا نلقي نظرةً على السلسلة النصية التالية المرمزة بترميز UTF-8 من الشيفرة 14:

```
let hello = String::from("Hola");
```

طول السلسلة النصية `len` في هذه الحالة هو 4، ما يعني أن الشعاع الذي يخزن السلسلة النصية "Hola" هو بطول 4 بايتات، إذ يأخذ كل حرف من هذه الأحرف 1 بايت عند ترميزه بترميز UTF-8، إلا أن السطر التالي قد يفاجئك (لاحظ أن السلسلة النصية التالية تبدأ بالحرف السيريلي زي Cyrillic letter Ze وليس العدد العربي 3).

```
let hello = String::from("Здравствуйте");
```

إذا سألناك عن طول السلسلة النصية ستجيب غالبًا 12، إلا أن رست ستجيبك بالقيمة 24 وهو رقم البايتات المطلوبة لترميز السلسلة النصية "Здравствуйте" بترميز UTF-8 وذلك لأن كل محرف يونيكود unicode يأخذ 2 بايت للتخزين، ولذلك استخدام دليل إلى بايت السلسلة النصية لن يعمل دومًا، ولتوضيح ذلك ألقِ نظرةً على شيفرة رست التالية غير الصالحة:

```
let hello = "Здравствуйте";
let answer = &hello[0];
```

أنت تعلم مسبقًا أن قيمة `answer` لن تكون الحرف الأول `3`، إذ تكون قيمة البايت الأول من الحرف `3` عند ترميز السلسلة النصية بترميز UTF-8 هي `208` والثاني قيمته 151 وبالتالي ستكون قيمة `answer` هي `208` إلا أن القيمة 208 ليست بقيمة صالحة لمحرف، وإعادة القيمة 208 ليست التصرف الذي يترقبه المستخدم غالبًا عند سؤاله عن المحرف الأول من السلسلة النصية، إلا أنها القيمة الوحيدة الموجودة في الدليل 0. لا يريد المستخدمون عادةً الحصول على قيمة البايت حتى لو احتوت السلسلة النصية على أحرف لاتينية، فإذا كانت `&"hello"[0]` شيفرة برمجية صالحة، فسيعيد ذلك قيمة البايت الممثلة بالقيمة 104 وليس h.

يكمن الحل هنا بتفادي إعادة قيمة غير متوقعة تتسبب بالأخطاء وقد لا نستطيع اكتشافها مباشـرةً، ولـذلك لا تسمح لنا رست بتصريف هذه الشيفرة البرمجية بهدف منع سوء التفاهم المسـتقبلي الـذي قـد يحصـل خلال عملية التطوير.

ب. البايتات والقيم العددية ومجموعات حروف اللغة

نقطة أخرى يجب ذكرها عن ترميز UTF-8، ألا وهو وجود ثلاث طرق مختلفة للنظر إلى السلاسل النصـية من منظور لغة رست: مثل بايتات أو قيم عددية scalar values أو مجموعات قيم عدديـة grapheme clusters (التمثيل الأقرب لما نسميه نحن البشر بالأحرف).

إذا نظرنا إلى الكلمة الهندية "नमस्ते" المكتوبة بالطريقة الديفاناغاريـة Devanagari، فهي مُخزّنـة على أنهـا شعاع من نوع u8 تبدو قيمه على الشكل التالي:

```
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164,
164,
165, 135]
```

الشعاع حجمه 18 بايت وهو ما يستخدمه الحاسوب لتخزين هـذه البيانـات، وإذا أردنـا النظـر إلى القيم على أنها قيم عددية، التي تمثّل نوع char في رست فسنحصل على البايتات كما يلي:

```
['न', 'म', 'स', '्', 'त', 'े']
```

هناك ست قيم من النوع char هنا إلا أن المحرف الرابع والسادس لا يمثلان أحرفًا وإنما علامات تشكيل لا تعني أي شيء بمفردهما. أخيرًا، إذا أردنا النظر إلى السلسلة النصية بكونهـا مجموعـات حـروف لغـة، فسنحصـل على تمثيل لما قد ندعوه بأحرف باللغة الهندية:

```
["न", "म", "स्", "ते"]
```

تقدم لغة رست طرقًا مختلفة لتمثيل بيانات السلسلة النصية وتخزينها بالحاسوب، بحيث يختار كل برنـامج التمثيل الذي يناسبه بغض النظر عن اللغة التي كُتب فيها النص.

السبب الأخير لكون رست لا تسمح بالوصول إلى النـوع String باسـتخدام الـدليل للحصـول على محـرف معين هـو أن عمليـات الفهرسـة تأخـذ تعقيـدًا زمنيًـا time complexity ثابتًـا -مقـداره O(1)- إلا أنـه ليس من الممكن ضمان ذلك الأداء مع النوع String لأن رست ستكون بحاجة للنظر إلى محتويات السلسلة النصية من البداية إلى الدليل المُحدّد لتحديد عدد المحارف الصالحة الموجودة.

8.2.5 شرائح السلاسل النصية

لا يُعد استخدام الأدلة مع السلاسل النصية فكرةً جيدةً كما أوضحنا سابقًا، إذ أن القيمة المُعادة من عملية الفهرسة ليست واضحة، فهل ستكون قيمة بايت أم محرف أم مجموعة حروف لغة أم شريحة سلسلة نصية. ستسألك رست أن تكون دقيقًا إذا أردت استخدام الأدلة للحصول على شريحة سلسلة نصية.

بدلًا من استخدام الأقواس `[]` مع رقم وحيد، يمكنك استخدام الأقواس `[]` لتحديد نطاق معين يحتوي على الشريحة النصية بعدد محدد من البايتات:

```
let hello = "Здравствуйте";

let s = &hello[0..4];
```

ستكون `s` هنا هي `&str` تحتوي على أول 4 بايتات من السلسلة. ذكرنا سابقًا أن كل حرف من هذه الأحرف هو بحجم 2 بايت، وهذا يعني أن `s` ستكون "Зд".

إذا حاولنا تقسيم جزء واحد من بايتات المحرف مثل `&hello[0..1]`، ستصاب رست بالهلع أثناء التشغيل بنفس الطريقة التي تحدث عند الوصول إلى دليل غير صالح في شعاعٍ ما، كما هو موضح في الخطأ التالي:

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/collections`
thread 'main' panicked at 'byte index 1 is not a char boundary; it is
inside '3' (bytes 0..2) of `Здравствуйте`',
library/core/src/str/mod.rs:127:5
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

ينبغي استخدام المجالات لإنشاء شرائح سلسلة نصية بحذر، لأن ذلك سيؤدي إلى تعطُّل برنامجك.

8.2.6 توابع التكرار على السلاسل النصية

أفضل طريقة للعمل على قطع pieces من السلاسل النصية هي أن تكون واضحًا فيما إذا كنت تريد أحرفًا أم بايتات، فمن أجل قيم عددية مفردة مُرمزة بالترميز الموحد يونيكود Unicode، استخدم التابع `chars`، إذ أن استدعاء هذا التابع على السلسلة "Зд" سيفصل المحرفين ويعيد قيمتان من النوع `char`، ويمكنك تكرار النتيجة للوصول إلى كل عنصر لوحده:

```
#![allow(unused)]
fn main() {
for c in "Зд".chars() {
    println!("{}", c);
}
}
```

ستطبع الشيفرة البرمجية ما يلي:

```
З
д
```

يمكننا استخدام التابع `bytes` بديلًا عمّا سبق وهو تابع يُعيد كل بايتًا كاملًا، إلا أنه قد يكون غير مناسبًا لاستخدامك:

```
for b in "Зд".bytes() {
    println!("{}", b);
}
```

تطبع الشيفرة البرمجية السابقة الخرج التالي:

```
208
151
208
180
```

تأكد من تذكرك أن القيمة العددية الصالحة ليونيكود قد تتألف من أكثر من بايت واحد.

الحصول على مجموعة حروف لغة من السلاسل النصية كما حدث في مثال الأحرف الديفاناغارية غير موجود في المكتبة القياسية لأنه معقّد، إلا أن هناك العديد من الصناديق crates.io crates التي تساعدك للحصول على النتيجة المرجوة.

8.2.7 السلاسل النصية ليست بسيطة

لنلخّص ما سبق، السلاسل النصية معقدة، وتتخذ لغات البرمجة المختلفة خيارات مختلفة لتقديم وتبسيط هذا التعقيد للمبرمج، واختارت رست هنا التقيد بالسلوك الموجود للتعامل مع بيانات من النوع `String` سلوكًا افتراضيًا لكل برامجها، ما يعني أنه على المبرمج التفكير مليًا عند التعامل مع بيانات بترميز UTF-8، وسلبيات هذا السلوك هو كشف تعقيد السلاسل النصية بوضوح أكثر من لغات البرمجة الأخرى، إلا أن هذا السلوك

يُعفيك من الحاجة إلى التعامل مع الأخطاء المتعلقة بالمحارف التي لا تنتمي إلى آسكي ASCII لاحقًا ضمن دورة حياة التطوير.

الخبر الجيد هنا هو أن المكتبة القياسية تقدم العديد من المزايا المبنية على كل من النـوعين `String` و `&s` `tr` لمساعدتنا في التعامل مع الحالات المعقدة بصورةٍ صحيحة. ألقِ نظرةً على التوثيق في حال أردت اسـتخدام توابع مفيـدة مثـل `contains` للبحث في سلسـلة نصـية و `replace` لاسـتبدال أجـزاء من السلسـلة النصـية بسلسلة نصية أخرى.

دعنا ننتقل إلى شيء أقل تعقيدًا، ألا وهو الخرائط المُعمّاة Hash maps.

8.3 تخزين مفاتيح وقيم مرتبطة بها عبر الخارطة المعماة Hash Map

نستعرض في هذا الفصل آخر التجميعـات الشـائعة في رسـت ألا وهي الخريطـة المعمّـاة hash map، إذ يخزّن النوع `HashMap<K, V>` ربطًا ما بين القيم من النوع `K` التي تمثل المفاتيح والقيم من النوع `V` التي تمثل القيم باستخدام دالة التعمية hashing function، التي تحدد أين يجب وضع هذه المفاتيح والقيم في الـذاكرة. تدعم كثيرًا من لغات البرمجة هذا النوع من هيكل البيانات إلا أنها غالبًـا مـا تسـتخدم اسـمًا مختلفًـا مثـل النـوع hash، أو خارطــة map، أو كــائن object، أو جــدول hash، أو قــاموس dictionary، أو مصــفوفة مترابطــة associative array والقائمة تطول.

الخرائط المعمّاة مفيدة عندما تريد البحث عن بيانات دون اسـتخدام دليـل لهـا كمـا هـو الحـال في الأشـعة vectors وإنما باستخدام مفتاح key قد يكون من أي نوع بيانات. على سـبيل المثـال، يمكنـك تتبـع نتيجـة كـل فريق في لعبة ما باستخدام الخريطة المعمّاة باستخدام اسم الفريق مفتاحًا للقيمة ونتيجـة كـل فريـق مثـل قيم، ويمكنك بالتالي الحصول على نتيجة الفريق باستخدام اسمه.

سنستعرض مبادئ الواجهة البرمجية الخاصـة بالخرائـط المعمّـاة في هـذا الفصـل، إلا أن هنـاك المزيـد من الأشياء المثيرة للاهتمام الموجودة ضمن الدوال المُعرّفة في `HashMap<K, V>` ضمن المكتبة القياسية، ولكن عليك التحقق من توثيق المكتبة القياسية إذا أردت المزيد من التفاصيل.

8.3.1 إنشاء خارطة معماة جديدة

إحدى الطرق لإنشاء خارطة معمّاة فارغة هي باستخدام `new` وإضافة العناصر باستخدام `insert`. نتـابع في الشيفرة 20 نتيجة فريقّين اسمهما أزرق Blue وأصفر Yellow، إذ يبدأ الفريق الأزرق بعشر نقاط والفريق الأصفر بخمسين نقطة.

```
use std::collections::HashMap;

let mut scores = HashMap::new();
```

```
    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);
```

[الشيفرة 20: إنشاء خارطة معماة جديدة وإضافة بعض المفاتيح والقيم إليها]

لاحظ أننا بحاجة كتابة `use` لتضمين `HashMap` من الجزء الذي يحتوي على التجميعات من المكتبة القياسية، وهذه هي التجميعة الأقل استخدامًا من التجميعات الثلاث الشائعة التي تكلمنا عنها، لذا فهي غير مضمّنة تلقائيًا في مقدمة البرنامج، إضافةً لما سبق، تملك الخرائط المعمّاة دعمًا أقل في المكتبة القياسية من التجميعات الأخرى وليس هناك ماكرو موجود مسبقًا لإنشاء خارطة معمّاة على سبيل المثال.

تخزّن الخرائط المعماة البيانات في الكومة heap كما هو الحال في الأشعة، إذ تحتوي الخريطة المُعمّاة السابقة على مفاتيح من النوع `String` وقيم من النوع `i32`، وكما هو الحال في الأشعة أيضًا فإن الخرائط المعماة متجانسة، بمعنى أن جميع المفاتيح يجب أن تكون من نفس النوع وكذلك الحال بالنسبة للقيم.

8.3.2 الوصول إلى القيم في الخارطة المعماة

يمكننا الحصول على قيمة في الخارطة المعماة باستخدام مفتاحها وتابع `get` كما هو موضح في الشيفرة 21.

```
fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    let team_name = String::from("Blue");
    let score = scores.get(&team_name).copied().unwrap_or(0);
}
```

[الشيفرة 21: الوصول إلى نتيجة الفريق الأزرق المخزنة في الخارطة المعمّاة]

سيأخذ `score` القيمة المرتبطة مع الفريق الأزرق وستكون النتيجة 10. يُعيد التابع `get` قيمةً من النوع `Option<&V>` وبالتالي إذا لم يكن هناك أي قيمة للمفتاح المحدد في الخارطة المعمّاة، فسيعيد التابع `get` القيمة `None`. يتعامل هذا البرنامج مع `Option` باستدعاء `unwrap_or` لضبط `score` إلى صفر إذا كان `score` لا يحتوي على قيمة للمفتاح المحدّد.

يمكننا المرور على كل زوج مفتاح/قيمة في الخريطة المعمّاة بطريقة مشابهة لما نفعل في الأشعة عن طريق حلقة `for`:

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

ستطبع الشيفرة البرمجية السابقة كل زوج بترتيب عشوائي:

```
Yellow: 50
Blue: 10
```

8.3.3 الخرائط المعماة والملكية

تُنسخ القيم التي تطبّق السمة `Copy`، مثل `i32` إلى الخارطة المعماة؛ بينما تُنقل القيم للأنواع المملوكة، مثل `String` وتصبح الخارطة المعماة مالكةً لهذه القيم كما هو موضح في الشيفرة 22.

```
use std::collections::HashMap;

let field_name = String::from("Favorite color");
let field_value = String::from("Blue");

let mut map = HashMap::new();
map.insert(field_name, field_value);
// يصبح كل من field_name و field_value غير صالح بحلول هذه النقطة
// حاول استخدامهما وانظر إلى خطأ المصرف الذي ستحصل عليه
```

[الشيفرة 22: توضيح أن المفاتيح والقيم تصبح مملوكة من قبل الخارطة المعماة فور إدخالها إليها]

لا يمكننا استخدام القيمتين `field_name` و `field_value` بعد نقلهما إلى الخارطة المعماة باستدعاء التابع `insert`.

لن تُنقل القيم إلى الخارطة المعماة في حال أضفنا مراجعًا القيم، إلا أن القيم التي تُشير إليها هذه المراجع يجب أن تكون صالحة طالما الخارطة المعماة صالحة على أقل تقدير، وسنتحدث مفصلًا عن هذه المشاكل لاحقًا.

8.3.4 تحديث قيم خارطة معماة

على الرغم من كون أزواج القيمة والمفتاح قابلة للزيادة إلا أنه يجب أن يقابل كل مفتاح فريد قيمة واحدة فقط (العكس ليس صحيحًا، على سبيل المثال قد تكون نتيجة كل من الفريق الأزرق والأصفر 10 في الخارطة المعماة `scores` في الوقت ذاته).

عليك أن تحدد التصرف الذي ستفعله عند تعديل القيم الموجودة في الخارطة المعمّاة والوصول إلى مفتاح له قيمة مسبقة، إذ يمكنك في هذه الحالة استبدال القيمة القديمة بالقيمة الجديدة وإهمال القيمة القديمة كليًا، أو تستطيع جمع القيمة القديمة مع القيمة الجديدة. دعنا ننظر إلى كيفية تحقيق كلا الطريقتين.

ا. الكتابة على القيمة

إذا أدخلنا مفتاحًا وقيمةً إلى خارطة معمّاة، ثمّ أدخلنا المفتاح ذاته مع قيمة مختلفة، ستُستبدل القيم المرتبطة بذلك المفتاح. على الرغم من أن الشيفرة 23 تستدعي التابع `insert` مرتين، إلا أن الخارطة المعماة الناتجة ستحتوي على زوج مفتاح/قيمة واحد لأننا أدخلنا قيمةً لمفتاح الفريق الأزرق في المرتين.

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Blue"), 25);

println!("{:?}", scores);
```

[الشيفرة 23: استبدال قيمة مخزّنة في خارطة معمّاة باستخدام مفتاحها]

ستطبع الشيفرة البرمجية السابقة `{Blue": 25"}`، إذ كُتِبَ على القيمة 10 السابقة القيمة 25.

ب. إضافة مفتاح وقيمة فقط في حال عدم وجود المفتاح

من الشائع أن نكون بحاجة للتحقق من مفتاح فيما إذا كان موجود مسبقًا أم لا في الخارطة المعمّاة مع قيمة ما، ومن ثم إجراء التالي: نبقي القيمة الموجود كما هي إذا كان المفتاح موجودًا في الخارطة المعماة، وإلا فنضيف المفتاح مع القيمة إذا لم نجد المفتاح.

لدى الخرائط المعماة واجهة برمجية خاصة بتلك العملية وهي التابع `entry` الذي يأخذ المفتاح التي تريد أن تتحقق منه مثل معامل، ويُعيد معددًا اسمه `Entry` يمثّل القيمة الموجودة أو غير الموجودة. دعنا نقول بأننا نريد أن نتحقق إذا كان مفتاح الفريق الأصفر يحتوي على قيمة؛ وإذا لم تكن هناك قيمة موجودة، نريد عندها إضافة القيمة 50 وينطبق الأمر ذاته على الفريق الأزرق. تبدو الشيفرة البرمجية باستخدام الواجهة البرمجية الخاصة بالتابع `entry` كما هو موضح في الشيفرة 24.

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);

scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);

println!("{:?}", scores);
```

[الشيفرة 24: استخدام التابع entry لإدخال قيمة فقط في حال لا يحتوي المفتاح على قيمة مرتبطة به]

التابع `or_insert` في `Entry` معرّف ليُعيد مرجعًا قابلًا للتعديل يشير إلى قيمة المفتاح `Entry` إذا كان المفتاح المحدد موجودًا، وإلا سيدخل قيمة المعامل على أنها قيمة جديدة للمفتاح وسيُعيد مرجعًا قابلًا للتعديل إلى القيمة الجديدة، وهذه الطريقة أفضل من كتابة المنطق ذلك بأنفسنا، كم أنها تعمل بصورةٍ أفضل مع مدقق الاستعارة borrow checker.

بتشغيل الشيفرة 24، نحصل على الخرج `{"Yellow": 50, "Blue": 10}`، إذ سيتسبب الاستدعاء الأول للتابع `entry` بإضافة مفتاح الفريق الأصفر بالقيمة 50 لأن الفريق الأصفر لا يحتوي على أي قيمة بعد، بينما لن يُغيّر الاستدعاء الثاني للتابع `entry` الخريطة المعمّاة لأن مفتاح الفريق الأزرق موجود مسبقًا بقيمة 10.

ج. تحديث قيمة بحسب قيمتها السابقة

حالة أخرى لاستخدام الخرائط المعماة هي بالبحث عن قيمة المفتاح ومن ثم التعديل عليها بناءً على قيمتها القديمة، على سبيل المثال توضح الشيفرة 25 برنامجًا يعدّ عدد مرات ظهور كل كلمة في النص، إذ نستخدم هنا خارطة معماة تحتوي على الكلمات مثل مفاتيح بحيث نزيد قيمة كل كلمة حتى نراقب عدد ظهور كلمة ما، وإذا رأينا الكلمة للمرة الأولى فإننا نضيفها إلى الخارطة بالقيمة 0.

```
use std::collections::HashMap;

```

```
let text = "hello world wonderful world";

let mut map = HashMap::new();

for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}

println!("{:?}", map);
```

[الشيفرة 25: عدّ عدد مرات ظهور الكلمات باستخدام خارطة معماة تخزّن الكلمة وعدد مرات ظهورها]

ستطبع الشيفرة البرمجية `{"world": 2, "hello": 1, "wonderful": 1}`.

قد تجد أزواج قيمة/مفتاح متماثلة بترتيب مختلف عن ذلك، لذلك نذكر هنا أننا تكلمنا في الفقرات السابقة وقلنا أن المرور على قيم الخارطة المعماة يكون بترتيب عشوائي.

يُعيد التابع `split_whitespace` مؤشرًا يشير إلى الشرائح الجزئية ضمن `text` والمفصول ما بينها بالمسافة، بينما يعيد التابع `or_insert` مرجعًا قابلًا للتعديل (`&mut V`) إلى قيمة المفتاح المُحدّد، ونخزّن هنا المرجع القابل للتعديل في المتغير `count`، لذا ومن أجل الإسناد إلى تلك القيمة علينا أولًا أن نُحصّل المتغير `count` باستخدام رمز النجمة `*`. تخرج المراجع القابلة للتعديل من النطاق في نهاية الحلقة `for`، لذا جميع هذه التغييرات آمنة ومسموحة استنادًا لقوانين الاستعارة.

8.3.5 دوال التعمية

تستخدم الخارطة المعماة `HashMap` افتراضيًا دالة تعمية hashing function تدعى SipHash، وهي دالة توفر حمايةً لخرائط التعمية ضد هجمات الحرمان من الخدمة Denial of Service- أو اختصارًا DoS- إلا أنها ليست أسرع خوارزميات التعمية المتاحة ولكنها تقدم حمايةً أفضل، والتراجع في السرعة مقابل ذلك يستحقّ المساومة. إذا راجعت شيفرتك البرمجية ورأيت أن دالة التعمية الاعتيادية بطيئة جدًا مقارنةً بحاجتك، فيمكنك استبدالها بدالة أخرى بتحديد مُعَمّي hasher؛ والمعمّي هو النوع الذي يطبّق السمة `BuildHasher` وسنناقش السمات بالتفصيل لاحقًا. ليس من الضروري كتابة المعمّي الخاص بك بنفسك، إذ يحتوي crates.io على مكتبات شاركها مستخدمو رست آخرون لتقديم تطبيق معمّي معيّن باستخدام خوارزميات التعمية الشهيرة.

8.4 خاتمة

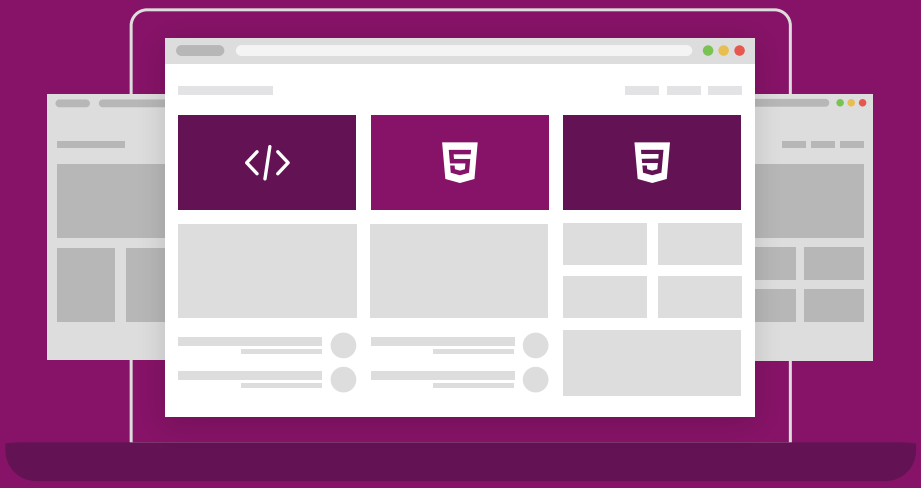
ستقدّم كل من الأشعة والسلاسل النصية وخرائط التعمية مزايا هامة في البرامج التي تخّـزن فيهـا البيانـات وتحاول الوصول إليها وتعديلها، إليك بعضًا من التمارين التي يمكنك محاولة حلّها الآن باستخدام ما تعلمناه لحد اللحظة:

- استخدم شعاعًا لحساب القيمة الوسطية median من لائحة من الأرقـام الصـحيحة (القيمـة الموجـودة في وسط اللائحة بعد ترتيبها) والمنوال mode (أكثر قيمـة مكـررة ضـمن اللائحـة، وسـيفيدنا اسـتخدام الخارطة المعماة هنا).
- تحويل السلاسل النصية إلى شيفرة pig latin، إذ يُنقـل أول حـرف غـير صـوتي من كـل كلمـة وصـولًا لنهايتها إلى بداية الكلمة ويُضاف "ay" بدلًا عنها، على سبيل المثال تصبح "first" بالشكل "irst-fay"، وإذا كان الحرف صوتيًا فتُضاف "hay" إلى نهايـة الكلمـة، على سـبيل المثـال تصـبح الكلمـة "apple" بالشكل "apple-hay" (تذكّر جميع التفاصيل الخاصة بالترميز UTF-8 التي ناقشناها سابقًا).
- أنشئ واجهة نصية باستخدام الخرائط المعماة والأشعة، بحيث تسمح للمستخدم بإضـافة اسـم موظـف إلى قطاع ما في شركة، على سبيل المثال "Add Sally to Engineering" (أي أضف سالي إلى قطاع الهندسة) أو "Add Amir to Sales" (أي أضف أمير إلى قطاع المبيعات)، ومن ثم اسمح للمستخدم بالحصول على لائحـة من الأشـخاص الـذي ينتمـون إلى قسـم مـا أو جميـع الأشـخاص الموجـودين في الشركة مرتبين بحسب قسمهم وترتيبهم الأبجدي.

يصف توثيق الواجهة البرمجية الخاص بالمكتبة القياسية التوابع المفيدة لكل من الأشعة والسلاسل النصية والخرائط المعماة لتمارين مشابهة لما سبق.

زاد تعقيد برامجنا بصورةٍ ملحوظة لحد اللحظة، لذا هذه هي اللحظة المناسبة لمناقشة التعامل مع الأخطـاء، وهذا ما سنفعله تاليًا.

دورة تطوير واجهات المستخدم



ابدأ مسارك المهني كمطور واجهات المواقع والمتاجر الإلكترونية
فور انتهائك من الدورة

التحق بالدورة الآن

9. الأخطاء والتعامل معها

لا مهرب من الأخطاء في دورة تطوير البرمجيات، لذا توفّر رست عددًا من المزايا للتعامل مع الحالات التي يحدث فيها شيء خاطئ، وتطلب رست منك في العديد من الحالات معرفتك باحتمالية حدوث الخطأ واتخاذ فعل ما قبل أن تُصرَّف compile الشيفرة البرمجية، ويجعل ذلك من برنامجك أكثر قوة بالتأكد من أنك ستكتشف الخطأ وستتعامل معه على نحوٍ مناسب قبل إطلاق شيفرتك البرمجية إلى مرحلة الإنتاج.

تصنِّف رست الأخطاء ضمن مجموعتين:

- **الأخطاء القابلة للحل recoverable errors**
- **الأخطاء غير القابلة للحل unrecoverable errors**

بالنسبة للأخطاء القابلة للحل، فهي أخطاء الهدف منها إعلام المستخدم بالمشكلة وإعادة محاولة العملية ذاتها مثل خطأ "لم يُعثَر على الملف file not found"، بينما تدل الأخطاء غير القابلة للحل دائمًا على أخطاء في الشيفرة البرمجية مثل محاولة الوصول إلى موقع يقع خارج نهاية مصفوفة وهذا يعني أننا نريد إيقاف تنفيذ البرنامج مباشرةً.

لا تميّز معظم لغات البرمجة بين النوعين السابقين وتتعامل معهما بنفس الطريقة باستخدام الاستثناءات exceptions، إلا أن رست لا تحتوي على الاستثناءات بل تحتوي على النوع `Result<T, E>` للأخطاء القابلة للحل والماكرو `panic!` الذي يوقف تنفيذ البرنامج عندما يصادف خطئًا غير قابل للحل، وسنغطّي في هذا الفصل كلًا من استدعاء الماكرو `panic!` والحصول على قيم النوع `Result<T, E>`.

9.1 الأخطاء غير القابلة للحل باستخدام الماكرو !panic

قد تحدث بعض الأخطاء من حين إلى الآخر في شيفرتك البرمجية، ولا يوجد أي شيء تستطيع فعلـه لتمنـع ظهورها، وفي هذه الحالة توفر لك رست الماكرو `panic!`.

هناك طريقتان لبدء حالة هلع panic، هما:

- فعل شيء يتسبب بهلع الشيفرة البرمجية، مثل محاولة الوصول إلى مكان خارج نطاق مصفوفة.
- أو استدعاء الماكرو `panic!` مباشرةً.

تتسبب الحالتين السابقتين بحالة هلع لبرنامجنا وتطبع حـالات الهلـع هـذه افتراضـيًا رسـالة فشـل ومن ثم تفرّغ محتويات المكدس stack وتغادر البرنامج. يمكنك عرض محتويات اسـتدعاء المكـدس عنـد حـدوث حالـة الهلع في لغة رست باستخدام متغير بيئة environment variable وذلك حتى تصبح مهمة تتبع مصـدر حالـة الهلع أسهل.

كيفية الاستجابة إلى حالة هلع

يبدأ البرنامج باستعادة الحالة الأولية unwinding افتراضيًا عنـد حـدوث حالـة هلـع، وهـذا يعـني أن رسـت تسترجع القيم الموجودة في المكدس وتفرغها، وتتضمن هذه العملية الكثير من العمل، لذلك تسـمح لـك رسـت باختيار الحل الثاني ألا وهو الخروج من البرنامج مباشرةً مما يُنهي تنفيذ البرنامج دون تفريغ المكدس.

عندها، تقع مسؤولية تحرير البيانات المستخدمة من البرنامج على عاتق نظـام التشـغيل. إذا أردت الحصـول على ملف تنفيذي في مشروعك بحجم صغير قدر الإمكان فعليك عندها التحويل من اسـتعادة الحالـة الأوليـة إلى الخروج من البرنامج فور حدوث حالة هلع بكتابة `panic = 'abort'` في قسم `[profile]` المناسب في ملف Cargo.toml. على سبيل المثال، إذا أردت الخروج من البرنـامج فـور حـدوث حالـة هلـع في نمـط الإطلاق release mode، فعليك بإضافة التالي:

```
[profile.release]
panic = 'abort'
```

دعنا نجرّب استدعاء الماكرو `panic!` في برنامج بسيط:

اسم الملف: src/main.rs

```
fn main() {
    panic!("crash and burn");
}
```

عند تشغيل البرنامج ستحصل على خرج مشابه لما يلي:

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.25s
     Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

يتسبب استدعاء الماكرو `panic!` برسالة الخطأ السابقة والموضحة في السطرين الأخيرين. يوضح السطر الأول رسالة الهلع ومكان حدوثه في شيفرتنا البرمجية، إذ يدل "src/main.rs:2:5" على أن حالة الهلع حدثت في السطر الثاني في المحرف الخامس ضمن الملف src/main.rs، ويكون السطر المشار إليه هو سطر ضمن شيفرتنا البرمجية التي كتبناها، وإذا ذهبنا إلى المكان المُحدّد فسنجد استدعاء الماكرو `panic!`، وقد يكون استدعاء الماكرو في حالات أخرى ضمن شيفرة برمجية أخرى تستدعيها شيفرتنا البرمجية وحينها سيكون اسم الملف ورقم السطر في رسالة الخطأ عائدين لشيفرة برمجية خاصة مكتوبة من قبل شخص آخر غيرنا وليس السطر الخاص بشيفرتنا البرمجية الذي أدى لاستدعاء `panic!`. يمكننا تتبع مسار backtrace الدالة التي استدعت `panic!` لمعرفة الجزء الذي تسبب بالمشكلة ضمن شيفرتنا البرمجية، وسنناقش تتبع مسار الخطأ بالتفصيل تاليًا.

9.1.1 تتبع مسار panic!

دعنا ننظر إلى مثال آخر لرؤية ما الذي يحدث عندما يُستدعى الماكرو `panic!` من مكتبة بسبب خطأ في شيفرتنا البرمجية، وذلك بدلًا من استدعائه من ضمن شيفرتنا البرمجية مباشرةً. توضح الشيفرة 1 محاولة الوصول إلى دليل في شعاع خارج نطاق الأدلة الصالحة.

اسم الملف: src/main.rs

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

[الشيفرة 1: محاولة الوصول إلى عنصر يقع خارج نهاية شعاع مما سيتسبب باستدعاء الماكرو panic!]

نحاول هنا الوصول إلى العنصر المئة في الشعاع (وهو العنصر ذو الدليل 99 لأن عدّ الأدلة يبدأ من الصفر)، إلا أن الشعاع يحتوي على ثلاثة عناصر فقط، وفي هذه الحالة تهلع رست؛ إذ من المفترض أن استخدام `[]`

سيعيد قيمة عنصر إلا أن تمرير دليل غير صالح يتسبب بهلع رست لأنها لا تعلم القيمة التي يجب أن تُعيدها بصورةٍ صحيحة.

تتسبب هذه المحاولة في لغة سي C بسلوك غير معرف undefined behaviour، إذ من الممكن أن تحصل على قيمة عشوائية في مكان الذاكرة تلك على الرغم من أن حيز الذاكرة ذلك لا ينتمي إلى هيكل البيانات ويُدعى هذا الأمر بتجاوز المخزن المؤقت buffer overread، ويمكن أن يتسبب بخطورات أمنية إذا استطاع المهاجم التلاعب بالدليل بطريقة تمكّنه من قراءة معلومات لا يُفترض له أن يقرأها بحيث تكون مخزّنة بعد هيكل البيانات.

لحماية برنامجك من هذا النوع من الثغرات، توقف رست تنفيذ البرنامج وترفض المتابعة إذا حاولت قراءة عنصر موجود في دليل خارج النطاق، دعنا نجرّب ذلك ونرى ما الذي يحدث:

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the
index is 99', src/main.rs:4:5
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

تُشير رسالة الخطأ إلى السطر 4 ضمن ملف "main.rs" وذلك هو السطر الذي نحاول عنده الوصول إلى الدليل 99، وتُخبرنا الملاحظة التالية أنه يمكننا ضبط متغير البيئة RUST_BACKTRACE للحصول على مسار تتبع الخطأ ومعرفة سبب حدوثه، إذ يمثّل مسار تتبع الخطأ لائحةً من جميع الدوال التي استُدعيت إلى نقطة حدوث حالة الهلع، ويعمل في رست على نحوٍ مماثل للغات البرمجة الأخرى كما يلي: المفتاح في قراءة مسار تتبع الخطأ هو البدء من البداية إلى نقطة وصولك للملفات التي كتبتها إذ أن الملفات التي كتبتها ستكون نقطة ظهور المشكلة، والسطور التي تقع قبل تلك النقطة هي السطور التي استدعتها شيفرتك البرمجية والسطور التي تلي تلك النقطة هي السطور التي استدعَت شيفرتك البرمجية، وقد تتضمن كل من هذه السطور شيفرة برمجية خاصة برست أو شيفرة برمجية خاصة بالمكتبة القياسية أو وحدات مُصرّفة crates تستخدمها.

دعنا نجرب الحصول على مسار تتبع الخطأ بضبط متغير البيئة RUST_BACKTRACE إلى أي قيمة عدا 0، وسيكون خرج الشيفرة 2 التالي مشابهًا لما ستحصل عليه عندها.

```
$ RUST_BACKTRACE=1 cargo run
thread 'main' panicked at 'index out of bounds: the len is 3 but the
index is 99', src/main.rs:4:5
stack backtrace:
```

```
rust_begin_unwind
             at
/rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/std/src/panick
ing.rs:584:5
core::panicking::panic_fmt
             at
/rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/panic
king.rs:142:14
core::panicking::panic_bounds_check
             at
/rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/panic
king.rs:84:5
<usize as core::slice::index::SliceIndex<[T]>>::index
             at
/rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/slice
/index.rs:242:10
core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
             at
/rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/slice
/index.rs:18:9
<alloc::vec::Vec<T,A> as core::ops::index::Index<I>>::index
             at
/rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/alloc/src/
vec/mod.rs:2591:9
panic::main
             at ./src/main.rs:4:5
core::ops::function::FnOnce::call_once
             at
/rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/ops/
function.rs:248:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a
verbose backtrace.
```

[الشيفرة 2: مسار تتبع الخطأ المولّد من استدعاء الماكرو panic! والذي يُعرض عند ضبط متغير البيئة RUST_BACKTRACE]

هناك الكثير من المعلومات في الخرج، وقد يكون الخرج الذي تراه أمامك مختلفًا عمّا ستحصل عليه بحسب نظام تشغيلك وإصدار رست. علينا تمكين رموز تنقيح الأخطاء debug symbols للحصول على مسار تتبع الأخطاء بالتفاصيل هذه، إذ تكون رموز تنقيح الأخطاء مفعّلة افتراضيًا باستخدام `cargo build` أو `cargo run` دون استخدام الراية `--release` كما هو الحال هنا.

يدل السطر 6 في الشيفرة 2 إلى أن مسار تتبع الأخطاء يشير إلى السطر المسبب للمشكلة في مشروعنا ألا وهو السطر 4 في الملف src/main.rs، وإن لم نرد لبرنامجنا أن يهلع فعلينا البدء بالنظر إلى ذلك المكان المحدد في السطر الأول الذي يذكر الملف الذي كتبناه وهو الشيفرة 1 الذي يحتوي على شيفرة برمجية تتسبب بالهلع عمدًا، وتكمن طريقة حل حالة الهلع هذه في عدم محاولة الوصول إلى عنصر يقع خارج نطاق أدلة الشعاع. عليك أن تكتشف العمل الذي يتسبب بحالة الهلع في برنامجك في المستقبل وذلك بالنظر إلى القيم التي تتسبب بحالة الهلع ومن ثم النظر إلى الشيفرة البرمجية التي تسببت بها وتعديلها.

سنعود لاحقًا إلى الماكرو `panic!` وإلى الحالات الواجب عدم استخدامها للتعامل مع الأخطاء لاحقًا، إلا أننا سننتقل حاليًا إلى كيفية الحل من الأخطاء باستخدام `Result`.

9.2 الأخطاء القابلة للحل باستخدام Result

ليست معظم الأخطاء خطيرة وتتطلب إيقاف كامل البرنامج عند حدوثها، ففي بعض الأحيان يدل فشل عمل دالة ما على سبب ما يتطلب انتباهك واستجابتك له، فعلى سبيل المثال إذا فشلت عملية فتح ملف ما فهذا يعني غالبًا أن الملف الذي حددته غير موجود ولعلّك تفضّل إنشاء الملف وإعادة المحاولة بدلًا من إنهاء البرنامج كاملًا.

تذكر أننا ذكرنا سابقًا في الفصل 2 أن المعدد enum الذي يُدعى `Result` معرّف وداخله متغايرَان variants، هما `Ok` و `Err` كما يلي:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

يمثّل كل من `T` و `E` معاملات نوع معمّم generic، وسنناقش الأنواع المعممة بتفصيل أكثر لاحقًا، ويكفي الآن معرفة أن `T` يمثّل نوع القيمة التي ستُعاد في حالة النجاح مع المتغاير `Ok`، بينما يمثل `E` نوع الخطأ الذي سيُعاد في حالة الفشل مع المتغاير `Err`، ولأن `Result` تحتوي على معاملات النوع المعمم فيمكننا استخدام النوع `Result` والدوال المعرفة ضمنها في العديد من الحالات، إذ تختلف كل من قيمة النجاح وقيمة الخطأ التي نريد أن نُعيدها.

دعنا نستدعي دالةً تُعيد القيمة `Result` لأن الدالة قد تفشل. نحاول في الشيفرة 3 فتح ملف.

اسم الملف: src/main.rs

```
use std::fs::File;

```

```
fn main() {
    let greeting_file_result = File::open("hello.txt");
}
```

[الشيفرة 3: فتح ملف]

النـــوع المُعـــاد من `File::open` هـــو `Result<T, E>`. يُملأ المعامـــل المعمّم `T` من خلال تنفيـــذ `File::open` مع نوع قيمة النجاح ألا وهي `std::fs::File` والتي تمثّل مقبض الملف file handle، بينما يُستخدم النوع `E` لتخزين قيمة الخطأ `std::io::Error`. يعني النوع المُعـاد هـذا أن اسـتدعاء `File::open` قد ينجح ويُعيد مقبض ملف يمكن الكتابة إليه أو القراءة منه، إلا أن استدعاء الدالـة قـد يفشـل في حـال لم يكن الملف موجودًا على سبيل المثال، أو عند عدم توافر الصـلاحيات المناسـبة للوصـول إليـه؛ وبالتـالي، ينبغي على الدالة `File::open` أن تمتلك القدرة على إخبارنا فيما إذا نجحت العملية ومنحنا مقبض الملف، أو إذا فشـلت وتوفّر معلومات مناسبة عن الخطأ بنفس الوقت، وهذه هي المعلومات الموجودة فعلًا في المعدّد `Result`.

ستكون قيمة المتغير `greeting_file_result` في حـال نجـاح `File::open` نسـخةً من `Ok` تحتـوي على مقبض الملف، وإلا فستكون قيمة المتغير `greeting_file_result` في حال الفشـل نسـخةً من `Err` تحتوي على المزيد من المعلومات حول نوع الخطأ الذي حدث.

نحتاج الإضـافة إلى الشـيفرة 3 لتتخّـذ بعض الإجـراءات المختلفـة بحسـب قيمـة `File::open` المُعـادة، وتوضح الشيفرة 4 طريقة من الطرق للتعامل مع `Result` باسـتخدام أداة بسـيطة ألا وهي تعبـير `match` الـذي ناقشناه سابقًا.

اسم الملف: src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {:?}", error),
    };
}
```

[الشيفرة 4: استخدام تعبير match للتعامل مع متغايرات Result]

لاحظ أن المعدد `Result` ومتغايراته -كما هو الحال مع المعدد `Option`- أُضيف إلى النطاق في بداية الشيفرة البرمجية، لذا ليس علينا تحديد `Result::` قبل المتغايرين `Ok` و `Err` في أذرع `match`.

تُعيد الشيفرة البرمجية قيمة `file` الداخلية من المتغاير `Ok` عندما تكون النتيجة `Ok`، ومن ثم نُسند قيمة مقبض الملف إلى المتغير `greeting_file`، ومن ثم يمكننا استخدام مقبض الملف للقراءة منه أو الكتابة إليه بعد التعبير `match`.

تتعامل الذراع الأخرى من `match` مع الحالات التي نحصل فيها على قيمة `Err` من `File::open`. استدعينا في هذا المثال الماكرو `panic!`، إذ سنحصل على الخرج التالي من الماكرو إذا لم يكن هناك أي ملف باسم "hello.txt" في المسار الحالي عند تشغيل الشيفرة البرمجية:

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
    Finished dev [unoptimized + debuginfo] target(s) in 0.73s
     Running `target/debug/error-handling`
thread 'main' panicked at 'Problem opening the file: Os { code: 2,
kind: NotFound, message: "No such file or directory" }',
src/main.rs:8:23
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

يخبرنا الخرج عن الخطأ بالتحديد كما اعتدنا.

9.2.1 مطابقة عدة أخطاء

ستهلع الشيفرة 4 (أي ستتسبب باستدعاء الماكرو `panic!`) عند فشل `File::open` لأي سبب من الأسباب، إلا أنه من الممكن أن نتخذ إجراءات مختلفة لكل سبب من الأسباب: على سبيل المثال نريد إنشاء ملف وإعادة مقبضه إذا فشلت `File::open` بسبب عدم وجود الملف؛ وإلا فنريد الشيفرة أن تهلع باستخدام `panic!` إذا كان السبب مختلفًا -مثل عدم امتلاكنا للأذونات المناسبة- بالطريقة ذاتها في الشيفرة 4، ولتحقيق ذلك نُضيف تعبير `match` داخلي كما هو موضح في الشيفرة 5.

اسم الملف: src/main.rs

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file_result = File::open("hello.txt");
```

```
    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file: {:?}",
e),
            },
            other_error => {
                panic!("Problem opening the file: {:?}", other_error);
            }
        },
    };
}
```

[الشيفرة 5: التعامل بصورةٍ مختلفة مع أخطاء مختلفة]

نوع القيمة التي تعيدها `File::open` داخل متغاير `Err` هو `io::Error` وهو هيكل struct موجود في المكتبة القياسية، ويحتوي هذا الهيكل على التابع `kind` الذي يمكننا استدعاءه للحصول على القيمة `io::ErrorKind`. يحتوي المعدّد `io::ErrorKind` الموجود في المكتبة القياسية على متغايرات تمثل الأنواع المختلفة من الأخطاء التي قد تنتج من عملية `io`، والمتغاير الذي نريد استخدامه هنا هو `ErrorKind::NotFound` الذي يشير إلى الملف الذي نحاول فتحه إلا أنه غير موجود بعد، لذا نُطابقه مع `greeting_file_result` إلا أنه يوجد تعبير `match` داخلي خاص بالتابع `error.kind()`.

الشرط الذي نريد أن نتحقق منه في تعبير `match` الداخلي هو فيما إذا كانت القيمة المُعادة من `error.kind()` هي المتغاير `NotFound` من المعدد `ErrorKind`، فإذا كان هذا الحال فعلًا فسنحاول إنشاء ملف باستخدام `File::create`، وإذا فشل `File::create` أيضًا، فنحن بحاجة ذراع آخر في تعبير `match` الداخلي، وعندما لا يكون من الممكن إنشاء الملف تُطبع رسالة خطأ مختلفة. تبقى ذراع `match` الخارجية الثانية كما هي حتى يهلع البرنامج عند حدوث أي خطأ ما عدا خطأ عدم العثور على الملف.

9.2.2 بدائل لاستخدام match مع Result<T, E>

استخدمنا كثيرًا من تعابير `match`، فهي مفيدةٌ جدًا إلا أنها بدائية، وسنتحدث لاحقًا عن المغلفات closures التي تُستخدم مع الكثير من التوابع المعرّفة في `Result<T, E>`، وقد تكون هذه التوابع أكثر اختصارًا من `match` عند التعامل مع قيم `Result<T, E>` في شيفرتك البرمجية. على سبيل المثال، إليك

طريقة أخرى لكتابة المنطق ذاته الموضح في الشيفرة 5، ولكن سنستخدم في هذه المرة المغلفات وتابع unwrap_or_else:

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap_or_else(|error|
{
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("Problem creating the file: {:?}", error);
            })
        } else {
            panic!("Problem opening the file: {:?}", error);
        }
    });
}
```

على الرغم من أن هذه الشيفرة البرمجية تبدي السلوك ذاته الخاص بالشيفرة 5، إلا أنها لا تحتوي على أي تعبير match وهي أوضح للقراءة. ألقِ نظرةً على التابع unwrap_or_else وكيفية عمله في توثيق المكتبة القياسية إذا أردت وعُد مرةً ثانية لهذا المثال. تغنينا العديد من التوابع الأخرى عن الحاجة لاستخدام تعابير match متداخلة عند التعامل مع الأخطاء.

9.2.3 اختصارات للهلع عند حصول الأخطاء باستخدام unwrap و expect

يفي استخدام match بالغرض، إلا أن استخدامه يتطلب كتابة مطوّلة ولا يدلّ على الغرض منه بوضوح. بدلًا من ذلك يحتوي النوع Result<T, E> العديد من التوابع المساعدة المعرفة بداخله لإنجاز مهام متعددة ومحددة، إذ يمثّل التابع unwrap مثلًا تابعًا مختصرًا يؤدي مهمّة التعبير match الذي كتبناه في الشيفرة 4، فإذا كانت قيمة Result هي المتغاير Ok، فسيعيد القيمة الموجودة في Ok وإلا إذا احتوى على المتغاير Err، فسيستدعي الماكرو panic!. إليك مثالًا عمليًا على unwrap:

اسم الملف: src/main.rs

```
use std::fs::File;

fn main() {
```

```
    let greeting_file = File::open("hello.txt").unwrap();
}
```

إذا نفذت الشيفرة البرمجية السابقة دون وجود الملف hello.txt، سنحصل على رسالة خطأ من استدعاء panic! بسبب التابع unwrap:

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err`
value: Os {
code: 2, kind: NotFound, message: "No such file or directory" }',
src/main.rs:4:49
```

يسمح لنا التابع expect باختيار رسالة خطأ panic! بصورةٍ مشابهة، كما يمكن أن ينقل استخدام expect بدلًا من unwrap وتقديم رسالة خطأ معبّرة قصدك جيدًا، مما يساعدك في تعقب مصدر الهلع بصورةٍ أفضل. يمكننا استخدام expect على الشكل التالي:

اسم الملف: src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")
        .expect("hello.txt should be included in this project");
}
```

نستخدم expect كما نستخدم unwrap، إما لإعادة مقبض الملف أو لاستدعاء الماكرو panic!. تمثل رسالة الخطأ التي سترسل باستخدام expect لاستدعاء panic! معاملًا يُمرّر إلى expect بدلًا من رسالة panic! الافتراضية التي يستخدمها التابع unwrap، إليك ما ستبدو عليه الرسالة:

```
thread 'main' panicked at 'hello.txt should be included in this
project: Os {
code: 2, kind: NotFound, message: "No such file or directory" }',
src/main.rs:5:10
```

يختار معظم مبرمجي لغة رست عند كتابة شيفرة برمجية مُخرجة جيدًا التابع expect بدلًا من unwrap لمنح السياق المناسب عن سبب نجاح العملية دومًا، ويمكنك بهذه الطريقة الحصول على معلومات أكثر لتستخدمها في تنقيح الأخطاء في حال كانت افتراضاتك خاطئة.

9.2.4 نشر الأخطاء

يُمكنك إعادة الخطأ الناتج عن استدعاء دالةٍ ما شيئًا قد يفشل إلى الشيفرة البرمجية المُستدعية له للتعامل مع الخطأ بدلًا من التعامل مع الخطأ داخل الدالة نفسها، وهذا يُعرف بنشر propagating الخطأ ويُعطي تحكمًا أكبر بالشيفرة البرمجية التي استدعت هذا الخطأ، إذ يمكننا توفير المزيد من المعلومات أو المنطق الذي يتعامل مع الخطأ بصورةٍ أفضل عما هو موجود في سياق شيفرتك البرمجية.

على سبيل المثال، ألقِ نظرةً على الشيفرة 6 التي تقرأ اسم مستخدم من ملف، وتُعيد الدالة خطأ عدم وجود الملف أو عدم القدرة على قراءته إلى الشيفرة البرمجية التي استدعت الدالة.

اسم الملف: src/main.rs

```
#![allow(unused)]
fn main() {
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
}
```

[الشيفرة 6: دالة تعيد الأخطاء إلى الشيفرة البرمجية التي استدعتها باستخدام تعبير match]

يُمكن كتابة هذه الدالة بطريقة أقصر إلا أننا سنبدأ بكتابة معظمها يدويًا حتى نفهم التعامل مع الأخطاء أكثر، ثم سننظر إلى الطريقة الأقصر. دعنا ننظر إلى النوع المُعاد من الدالة أولًا ألا وهو

`Result<String, io::Error>` وهذا يعني أن الدالة تُعيد قيمـةً من النـوع `Result< T, E>`، إذ يُملأ النوع المعمّم `T` بالنوع `String`، بينما يُملأ النوع المعمم `E` بالنوع `io::Error`.

تحصل الشيفرة البرمجية التي استدعت الدالة في حال عمـل الدالـة دون أي مشـاكل على القيمـة `Ok` الـتي تخزّن داخلها قيمةً من النوع `String` ألا وهو اسم المستخدم الذي قرأته الدالـة من الملـف، وإذا واجهت الدالـة خلال عملها أي خطأ، تحصل الشيفرة البرمجية التي استدعت الدالة على القيمة `Err` الـتي تخـزن داخلهـا نسـخةً من `io::Error` تحتوي على مزيدٍ من المعلومات حول المشاكل التي جرت. اخترنا `io::Error` نوعًـا للقيمـة المُعادة لأنه يوافق نوع قيمة الخطأ المُعاد من كلا العمليتين التي نستدعي فيهمـا الدالـة اللتـان قـد تفشـلان ألا وهما الدالة `File::open` والتابع `read_to_string`.

يبدأ متن الدالة باستدعاءٍ للدالة `File::open`، ثمّ نتعامل مع القيمة `Result` في `match` بطريقة مماثلـة للتعبير `match` في الشيفرة 4؛ فـإذا نجح عمـل الدالـة `File::open` يصـبح مقبض الملـف في متغـير النمـط `file` بقيمة المتغير القابل للتغيير `username_file` ويستمر تنفيذ الدالة، إلا أننا نسـتخدم الكلمـة المفتاحيـة `return` في حالـة `Err` عوضًـا عن اسـتدعاء `panic!` للخـروج من الدالـة وتمريـر قيمـة الخطـأ الناتجـة عن `File::open` في متغير النمط `e` إلى الشيفرة البرمجية التي استدعت الدالة.

إذًا، تُنشـئ الدالـة قيمـة `String` جديـدة في المتغـير `username` إذا كـان لـدينا مقبض ملـف في `username_file`، ثم تسـتدعي التـابع `read_to_string` باسـتخدام مقبض الملـف في المتغـير `username_file` لقراءة محتويات الملـف إلى المتغـير `username`. يعيـد التـابع `read_to_string` قيمـة `Result` أيضًا لأنها من الممكن أن تفشل على الرغم من نجاح `File::open`، لـذا فنحن بحاجـة تعبـير `match` آخر للتعامل مع قيمة `Result` على النحو التـالي: تنجح دالتنـا إذا نجح التـابع `read_to_string` ونُعيـد اسـم المستخدم من الملف الموجود في `username` والمغلّـف بقيمـة `Ok`، وإلا إذا فشـل `read_to_string` نُعيـد قيمة الخطأ بطريقة إعادة الخطأ ذاتها في `match` التي تعاملت مـع القيمـة المُعـادة من `File::open`، إلا أننـا لسنا بحاجة كتابة الكلمة المفتاحية `return` هنا لأن هذا هو آخر تعبير في الدالة.

ستتعامل الشيفرة البرمجية التي تستدعي هذه الشيفرة البرمجية مع حالـة الحصـول على قيمـة `Ok` تحتـوي على اسم مستخدم، أو قيمة `Err` تحتوي على قيمة من النـوع `io::Error`، ويعـود اختيـار الإجـراء المُتّخـذ إلى الشيفرة البرمجية التي اسـتدعت الدالـة، فيمكن للشـيفرة البرمجيـة أن تسـتدعي المـاكرو `panic!` وأن توقـف البرنـامج فـورًا في حـال الحصـول على قيمـة `Err`، أو اسـتخدام اسـم مسـتخدم افتراضـي، أو البحث على اسـم المستخدم في مكان آخـر عوضًـا عن الملـف. لا نمتلـك مـا يكفي من المعلومـات حـول الشـيء الـذي سـتفعله الشيفرة البرمجية التي استدعت الدالة، لذا فنحن ننشر معلومـات الخطـأ أو النجـاح للشـيفرة البرمجيـة للتعامـل معها بصورةٍ مناسبة.

يُعد نمط نشر الأخطاء هذا شائع جدًا في رست، وتقدم لنا رست عامـل إشـارة الاسـتفهام `?` لاسـتخدام هـذا النمط بسهولة.

ا. اختصار لنشر الأخطاء: عامل ?

توضح الشيفرة 7 تطبيقًا للدالة `read_username_from_file` بوظيفة مماثلة للشيفرة 6، إلا أننا نستخدم هنا العامل `?`.

اسم الملف: src/main.rs

```
#![allow(unused)]
fn main() {
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("hello.txt")?;
    let mut username = String::new();
    username_file.read_to_string(&mut username)?;
    Ok(username)
}
}
```

[الشيفرة 7: دالة تعيد أخطاء للشيفرة البرمجية المُستدعية باستخدام العامل `?`]

العامل `?` الموجود بعد القيمة `Result` مُعرَّف بحيث يعمل بطريقة مماثلة لعمل تعابير `match` التي عرفناها سابقًا بهدف التعامل مع قيم `Result` المختلفة في الشيفرة 6، فإذا كانت القيمة `Result` هي `Ok` تُعاد القيمة داخل `Ok` من التعبير هذا ويستمر تنفيذ البرنامج، بينما إذا كانت القيمة `Err` فتُعاد القيمة الموجودة داخل `Err` من الدالة ككُل وكأننا استخدمنا الكلمة المفتاحية `return` وبالتالي تُنشر قيمة الخطأ إلى الشيفرة البرمجية التي استدعت الدالة.

هناك فرقٌ ما بين ما يفعله التعبير `match` في الشيفرة 6 وبين ما يفعله العامل `?`؛ إذ أن الأخطاء التي تُستدعى عن طريقة العامل `?` تمرّ بدالة `from`، المعرّفة في السمة `From` في المكتبة القياسية التي تُستخدم لتحويل القيم من نوع إلى نوع آخر؛ فعندما يستدعي العامل `?` الدالة `from` يُحوَّل الخطأ المُتلقى إلى نوع الخطأ المعرف في نوع القيمة المُعادة ضمن الدالة الحالية، وهذا الأمر مفيد عندما تُعيد الدالة نوعًا واحدًا من الخطأ لتمثيل جميع حالات فشل الدالة، حتى لو كانت الأجزاء التي قد تفشل ضمن الدالة تفشل لأسباب مختلفة.

على سبيل المثال، يمكننا التعديل على الدالة `read_username_from_file` في الشيفرة 7 لتُعيد نوع خطأ مخصص نعرّفه اسمه `OurError`. إذا عرفنا أيضًا `impl From<io::Error> for OurError` لإنشاء نسخة من `OurError` من `io::Error` فهذا يعني أن العامل `?` المُستدعى في متن الدالة

`read_username_from_file` سيستدعي `from` ويحوّل أنواع الأخطاء دون الحاجة لكتابة شيفرة برمجية إضافية لهذا الغرض.

في سياق الشيفرة 7: سيُعيد العامل `?` في نهاية استدعاء `File::open` القيمة الموجودة داخل `Ok` إلى المتغير `username_file`، وإذا حدث خطأ ما فسيعيد العامل `?` قيمةً من `Err` إلى الشيفرة التي استدعت الدالة وتوقف تنفيذ الدالة مبكرًا، وينطبق الأمر ذاته على العامل `?` في نهاية استدعاء `read_to_string`.

يُغنينا العامل `?` عن كتابة أي شيفرات برمجية متكررة ويجعل من كتابة الدالة عمليةً أسهل وأسرع، إلا أنه يمكننا جعل الشيفرة البرمجية هذه أقصر أكثر عن طريق كتابة استدعاءات التوابع الواحدة تلو الأخرى مباشرةً بعد العامل `?` كما هو موضح في الشيفرة 8.

اسم الملف: src/main.rs

```
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username = String::new();

    File::open("hello.txt")?.read_to_string(&mut username)?;

    Ok(username)
}
```

[الشيفرة 8: كتابة استدعاءات التوابع بصورة متسلسلة بعد العامل `?`]

نقلنا عملية إنشاء `String` الجديد في `username` إلى بداية الدالة، ولم نغيّر ذلك الجزء، وبدلًا من إنشاء متغير `username_file`، كتبنا استدعاء `read_to_string` قبل نتيجة `File::open("hello.txt:)?` مباشرةً، إلا أن العامل `?` ما زال موجودًا في نهاية استدعاء `read_to_string` وما زلنا نُعيد قيمة `Ok` تحتوي على `username` عندما تنجح كل من `File::open` و `read_to_string` بدلًا من إعادة الأخطاء. وظيفة الشيفرة البرمجية مماثلة لكل من الشيفرة 6 والشيفرة 7 إلا أن الفارق هنا أن الشيفرة هذه أكثر سهولة للكتابة.

توضح الشيفرة 9 طريقةً أكثر اختصارًا باستخدام `fs::read_to_string`.

اسم الملف: src/main.rs

```
use std::fs;
use std::io;
```

```
fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

[الشيفرة 9: استخدام `fs::read_to_string` بدلًا من فتح وقراءة الملف بخطوتين منفصلتين]

تُعد قراءة محتويات ملف ما وإسنادها إلى سلسلة نصية عمليةً شائعةً جدًا، لذا تُقدّم المكتبة القياسية دالةً لتحقيق هذه العملية ألا وهي `fs::read_to_string`، إذ تفتح الملف ومن ثم تُنشئ سلسلةً نصيةً `String` جديدة وتقرأ محتويات الملف وتُسندها إلى السلسلة النصية وتُعيد السلسلة النصية `String` أخيرًا، إلا أن استخدام `fs::read_to_string` لا يُتيح لنا إمكانية شرح جميع حالات التعامل مع الأخطاء، وهذا السبب في تقديمنا للطريقة الأطول أولًا.

ب. أماكن استخدام العامل ?

يُمكن استخدام العامل `?` فقط في الدوال التي تُعيد نوعًا متوافقًا مع العامل `?`، وذلك لأن العامل `?` معرّف ليُجري عملية إعادة لقيمة بصورةٍ مبكرة خارج الدالة بطريقة تعبير `match` ذاتها الذي عرفناه في الشيفرة 6. نلاحظ في الشيفرة 6 أن `match` استخدم القيمة `Result` وأعاد الذراع القيمة `Err(e)`، ينبغي على النوع المُعاد من الدالة أن يكون `Result` لكي يكون متوافقًا مع التعليمة `return` هذه.

دعنا ننظر في الشيفرة 10 إلى الخطأ الذي سنحصل عليه في حال استخدمنا العامل `?` في الدالة `main` بنوع مُعاد غير متوافق مع الأنواع الخاصة بالعامل `?`:

اسم الملف: src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")?;
}
```

[الشيفرة 10: محاولة استخدام العامل `?` في الدالة `main` التي تُعيد `()` وهي قيمة غير متوافقة]

تفتح الشيفرة البرمجية السابقة ملفًا، وقد تفشل عملية فتحه. يتبع العامل `?` القيمة `Result` المُعادة من `File::open` إلا أن الدالة `main` تحتوي على النوع المُعاد `()` وليس `Result`، وعندما نصرّف الشيفرة البرمجية السابقة نحصل على رسالة الخطأ التالية:

```
$ cargo run
```

```
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
error[E0277]: the `?` operator can only be used in a function that
returns `Result` or `Option` (or another type that implements
`FromResidual`)
 --> src/main.rs:4:48
  |
| / fn main() {
| |     let greeting_file = File::open("hello.txt")?;
  | |                                                ^ cannot use the
`?` operator in a function that returns `()`
| | }
  | |_- this function should return `Result` or `Option` to accept `?`
  |
  = help: the trait `FromResidual<Result<Infallible, std::io::Error>>`
is not implemented for `()`


For more information about this error, try `rustc --explain E0277`.
error: could not compile `error-handling` due to previous error
```

يشير هذا الخطأ إلى أنه من غير المسموح استخدام العامل `?` إلا في دالة تُعيـد `Result`، أو `Option`، أو أي نوع آخر يطبّق `FromResidual`.

يوجد خياران لإصلاح الخطأ السابق: الأول هو تغيير نوع القيمة المُعادة من الدالة لتصبح متوافقةً مع القيمة التي تستخدم العامل `?` عليها وهذا خيار جيّد طالما لا يوجد أي قيود أخرى تمنعك من ذلك، أما الخيار الثاني فهو باستخدام `match` أو إحدى توابع `Result<T, E>` للتعامل مع `Result<T, E>` بالطريقة المناسبة.

ذكرت رسالة الخطأ أيضًا أنه يمكننـا اسـتخدام العامـل `?` مـع قيم `Option<T>` أيضًـا بالطريقـة ذاتهـا الـتي نستخدم فيها العامل مع `Result`، إلا أنه يمكنك استخدام العامل على `Option` فقط في حال كانت الدالة تُعيد `Option`. يشــبه ســلوك العامــل `?` عنــد اســتدعائه على `Option<T, E>` ســلوكه عنــد اســتدعائه على `Result<T, E>`، إذ تُعاد القيمة `None` كما هي مبكرًا من الدالة، وإذا كانت القيمة `Some` فالقيمة التي بـداخل `Some` هي القيمة الناتجة عن ذلك التعبير، وتستمر الدالة عندها بالتنفيـذ. تحتـوي الشـيفرة 11 على مثـال لدالـة تعثر على المحرف الأخير من السطر الأول في سلسلة نصية.

```
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

[الشيفرة 11: استخدام العامل `?` على قيمة `Option<T>`]

تُعيد الدالة `Option<char>` لأنه من الممكن أن يكون هناك محرف في النتيجة أو لا. تأخذ الشيفرة البرمجية السابقة شريحة السلسلة النصية string slice `text` وسيطًا، وتستدعي التابع `lines` عليها مما يُعيد مُكرّرًا iterator عبر السطور في السلسلة النصية، ولأن هذه الدالة تهدف لفحص السطر الأول فهي تستدعي `next` على المكرر للحصول على القيمة الأولى منه، وإذا كان `text` سلسلة نصية فارغة فسيُعيد استدعاء `next` القيمة `None` وفي هذه الحالة نستخدم العامل `?` لإيقاف التنفيذ وإعادة القيمة `None` من الدالة `last_char_of_first_line`. إذا لم يكن `text` سلسلةً نصيةً فارغة فسيُعيد استدعاء `next` قيمة `Some` تحتوي على شريحة سلسلة نصية تحتوي على السطر الأول من `text`.

يستخلص العامل `?` شريحة السلسلة النصية ويمكننا استدعاء `chars` على شريحة السلسلة النصية للحصول على مكرّر يحتوي على محارفه. ما نبحث عنه هنا هو المحرف الأخير من السطر الأول، لذلك نستدعي `last` للحصول على آخر عنصر موجود في المكرر وهي قيمة `Option` لأنه من الممكن أن يكون السطر الأول سلسلة نصية فارغة، إذا من الممكن مثلًا أن يبدأ `text` بسطر فارغ وأن يحتوي على محارف في السطور الأخرى مثل `"\nhi"`، فإذا كان هناك فعلًا محرف في نهاية السطر فإننا نحصل عليه داخل متغاير `Some`. يُعطينا العامل `?` الموجود في المنتصف طريقة موجزة للتعبير عن هذا المنطق مما يسمح لنا بتطبيق محتوى الدالة بسطر واحد، وإذا لم نستطع تطبيق العامل `?` على `Option`، سيتوجب علينا كتابة المنطق ذاته باستخدام عدد أكبر من استدعاءات للدوال أو باستخدام التعبير `match`.

لاحظ أنه يمكنك استخدام العامل `?` على `Result` داخل دالة تُعيد `Result`، ويمكنك استخدام العامل `?` على `Option` داخل دالة تُعيد `Option` إلا أنه لا يمكنك الخلط ما بين الاثنين، إذ لن يحول العامل `?` النوع `Result` إلى `Option` أو بالعكس تلقائيًا، ويمكنك في هذه الحالات استخدام توابع، مثل التابع `ok` على النوع `Result`، أو التابع `ok_or` على النوع `Option` لإجراء التحويل صراحةً.

استخدمت جميع دوال `main` حتى هذه اللحظة القيمة المُعادة `()`. تُعد الدالة `main` مميزةً لأنها نقطة بداية ونهاية البرامج التنفيذية وبالتالي هناك بعض القيود على الأنواع المُعادة لكي تتصرف البرامج على النحو الصحيح كما هو متوقع.

لحسن الحظ، تُعيد الدالة `main` النوع `Result<(), E>`. تحتوي الشيفرة 12 على الشيفرة البرمجية الموجودة في الشيفرة 10، إلا أننا عدلنا النوع المُعاد من الدالة `main` ليصبح `Result<(), Box<dyn Error>>` وأضفنا قيمة مُعادة `Ok(())` في النهاية، وستُصرَّف الشيفرة البرمجية نتيجةً لهذه التعديلات بنجاح:

```
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
```

```
    let greeting_file = File::open("hello.txt")?;

    Ok(())
}
```

[الشيفرة 12: تعديل الدالة `main` لتُعيد `Result<(), E>` لتسمح لنا باستخدام العامل `?` على قيم `Result`]

النوع `Box<dyn Error>` هو كائن سمة trait object وهو ما سنغطيه لاحقًا، ويكفي الآن معرفتك أن `Box<dyn Error>` يعني "أي نوع من الأخطاء". استخدام العامل `?` على قيمة `Result` في دالة `main` باستخدام قيمة الخطأ `Box<dyn Error>` هو أمر مسموح، وذلك لأنه يسمح لأي نوع `Err` أن يُعاد مبكرًا، وعلى الرغم من أن محتوى الدالة `main` سيعيد الأخطاء من النوع `std::io::Error` فقط إلا أن بصمة الدالة ستبقى صالحة بتحديد `Box<dyn Error>` إذا أُضيفت شيفرة برمجية تُعيد أخطاء أخرى داخل الدالة `main`.

يتوقف الملف التنفيذي عندما تُعيد الدالة `main` القيمة `Result<(), E>` وذلك بإعادة القيمة "0" إذا أعادت `main` القيمة `Ok(())` وقيمة غير صفرية إذا أعادت الدالة قيمة `Err`. تُعيد الملفات التنفيذية المكتوبة بلغة سي أعدادًا صحيحة عند مغادرة البرنامج؛ فالبرنامج الذي يتوقف بنجاح يُعيد العدد الصحيح "0"؛ بينما يُعيد البرنامج الذي يتوقف بسبب خطأ قيمة عدد صحيح لا تساوي "0". تُعيد رست أيضًا أعدادًا صحيحة من الملفات التنفيذية بصورةٍ مماثلة لهذا الاصطلاح.

قد تُعيد الدالة `main` أي نوع يطبّق السمة std::process::Termination التي تحتوي بدورها على دالة تدعى `report` تُعيد قيمة `ExitCode`، انظر إلى توثيق المكتبة القياسية للمزيد من المعلومات حول استخدام سمة `Termination` ضمن أنواعك.

الآن، بعد مناقشتنا التفاصيل الخاصة بالماكرو `panic!` وإعادة النوع `Result`، سنعود تاليًا إلى موضوع كيفية تحديد الاستخدام المناسب لكل حالة.

9.3 الاختيار ما بين الماكرو panic! والنوع Result للتعامل مع الأخطاء

كيف يمكننا الاختيار ما بين استدعاء الماكرو `panic!` وإعادة القيمة `Result` عند حدوث الأخطاء؟ عندما تهلع الشيفرة البرمجية (أي عند استدعاء الماكرو `panic!`)، فليس هناك أي طريقة لحل ذلك الخطأ، ويمكنك استدعاء `panic!` لأي خطأ كان سواءٌ كان خطأ يُمكن حلّه أو لا، فأنت من يتخذ القرار بجعل الخطأ هذا قابلًا للحل في شيفرتك البرمجية أم لا؛ فعندما تختار إعادة القيمة `Result`، فأنت تحاول منح الشيفرة البرمجية التي استدعت ذلك الفعل الذي تسبب بالخطأ (دالةٌ ما) بعض الخيارات للتعامل مع ذلك الخطأ بحيث تحاول الشيفرة البرمجية حله بطريقة ملائمة لكل حالة، أو أن تحدد أن قيمة الخطأ `Err` غير قابلة للحل مما يتسبب باستدعاء الماكرو `panic!` وبالتالي جعل الخطأ القابل للحل خطأ غير قابل للحل. إذًا، إعادة القيمة `Result` هي خيار افتراضي جيّد عندما تعرّف دالة ما قد تفشل في بعض الأحيان.

من المحبّذ في حالات كتابة الأمثلة والشيفرات البرمجية التجريبية كتابة شيفرة برمجية تهلع بـدلًا من إعـادة القيمة `Result`، دعنا ننظر إلى السـبب ومن ثم نـنـاقش الحـالات الـتي قـد لا يسـتطيع فيهـا المصـرّف إخبارنـا بإمكانية حدوث خطأ ما، إلا أنك تستطيع فعل ذلك. سنختتم أيضًا هـذا الفصـل بتوجيهـات عامـة تسـاعدك في تحديد الحالات التي يكون فيها جعل الشيفرة البرمجية تهلع بصورةٍ أفضل.

9.3.1 أمثلة وشيفرات برمجية تجريبية واختبارات

استخدام شيفرة برمجية تتعامل مع الأخطاء بصورةٍ جيدة عندما تكتب مثالًا ما لتوضيح مفهوم معين يجعـل من المثال أقل وضوحًا، ونسـتدعي عـادةً في الأمثلـة تابعًـا مثـل `unwrap` يمكن أن يهلـع مثـل موضـع مـؤقت placeholder من أجل الطريقة التي تريد التعامل مع الأخطاء في تطبيقك، والتي قد تختلف بناءً على ما تفعلـه بقية الشيفرة البرمجية.

التابعان `unwrap` و `expect` مفيدان جدًا أيضًا عند كتابة الشيفرات البرمجية التجريبية قبـل أن تتخـذ قـرار بخصوص التعامل مع الأخطاء، إذ يساعدك التابعان بترك علامات واضحة في شيفرتك البرمجية انتظـارًا للـوقت الذي تريد فيه جعل برنامجك أكثر متانةً بالتعامل مع الأخطاء.

إذا فشل استدعاء تابع ما ضمن اختبار فمن الأفضل جعل كل الاختبار يفشل حتى لـو كـان التـابع ذلـك غـير مضمّنٍ في الاختبار الأساسي، ولأن الماكرو `panic!` هو بمثابة إشـارة إلى أن الاختبـار سيفشـل، فـإن اسـتدعاء `unwrap` أو `expect` هو ما يجب حدوثه.

9.3.2 الحالات التي تعرف فيها معلومات أكثر من المصرف

من الملائم أيضًا استدعاء `unwrap` أو `expect` عنـد تواجـد منطـق مـا يضـمن أن `Result` سـتحتوي على قيمة `Ok` إلا أن هذا المنطق لا يمكن فهمه من قبل المصرّف، إذ ستتواجد قيمة `Result` بحاجـة للتعامـل معهـا وفحصها، فأي عملية تستدعيها قد تفشل عمومًا على الرغم من أن الأمر مستحيل منطقيًـا في هـذه الحالـة. من المقبول استدعاء `unwrap` إذا استطعت أن تتأكد بفحص الشيفرة البرمجيـة يـدويًا أنهـا لن تحتـوي على متغـاير `Err` أبدًا، والأفضل في هذه الحالة أن توثّق السبب الذي تعتقد أنـك لن تحصـل فيـه على متغـاير `Err` في نص `expect`. إليك مثالًا:

```
fn main() {
    use std::net::IpAddr;

    let home: IpAddr = "127.0.0.1"
        .parse()
        .expect("Hardcoded IP address should be valid");
}
```

نُنشئ هنا نسخةً من `IpAddr` بالمرور على السلسلة النصية المكتوبة في الشيفرة البرمجية، ويمكن ملاحظة أن "127.0.0.1" هو عنوان IP صالح وبالتالي من المقبول استخدام `expect` هنا، إلا أن وجود سلسلة نصية صالحة مكتوبة في الشيفرة البرمجية لا يغير من النوع المعاد للتابع `parse`، إذ أننا ما زلنا نحصل على قيمة `Result` وسيجبرنا المصرف على التعامل مع `Result` لأنه يفترض أن وجود المتغاير `Err` بداخل `Result` أمر ممكن الحدوث وذلك لأن المصرف ليس ذكي بالقدر الكافي ليرى أن السلسلة النصية تمثل عنوان IP صالح دومًا.

يوجد احتمال بوجود المتغاير `Err` إذا أتى عنوان IP مثل دخل من المستخدم بدلًا من كتابته مباشرةً في الشيفرة البرمجية وعندها علينا أن نتعامل مع `Result` بطريقة شاملة بدلًا من الطريقة الحالية. ذكر الافتراض أن عنوان IP هذا مكتوب في الشيفرة البرمجية وبالتالي سيُطلب منّا تغيير `expect` للحصول على شيفرة برمجية تتعامل مع الأخطاء بصورةٍ أفضل في المستقبل، وبالتالي فنحن بحاجة للحصول على عنوان IP من مصادر أخرى.

9.3.3 توجيهات للتعامل مع الأخطاء

يُنصح بجعل الشيفرة البرمجية تهلع عندما يمكن أن يؤدي الخطأ إلى حالة سيئة bad state للشيفرة البرمجية، ونقصد هنا بالحالة السيئة الحالة التي يتغير فيها افتراض assumption، أو ضمان guarantee، أو عقد contract، أو ثابت invariant، مثل الحصول على قيم غير صحيحة أو متناقضة أو مفقودة، إضافةً إلى واحدة أو أكثر من الحالات التالية:

- الحالة السيئة هي حالة غير متوقعة لا تحدث عادةً، مثل إدخال المستخدم بياناته بتنسيق خاطئ.
- يجب أن تعتمد شيفرتك البرمجية بعد حدوث هذه الحالة على فحص المشكلة بعد كل خطوة بدلًا من اتخاذ معطيات الحالة السيئة هذه.
- لا توجد طريقة مثالية لتمثيل هذه البيانات في الأنواع التي تستخدمها، وسننظر إلى مثال على ذلك في الفصول القادمة.

إذا استدعى أحدهم شيفرتك البرمجية ومرّر لها القيم التي تؤدي إلى حالة سيئة، فمن الأفضل إعادة الخطأ إذا استطعت، حتى يقرّر مستخدم المكتبة الإجراء الذي يريد اتخاذه في هذه الحالة، إلا أن الاستمرار بتنفيذ الشيفرة البرمجية في بعض الأحيان قد يكون غير آمن أو ضار، وفي هذه الحالة يكون استدعاء الماكرو `panic!` أفضل خيار لتنبيه المستخدم الذي يستخدم المكتبة بالخطأ الموجود في شيفرته البرمجية وكيف يستطيع إصلاحه خلال عملية التطوير. استخدام `panic!` أيضًا مناسب في حال استدعاء الشيفرة البرمجية الخارجية التي لا تستطيع التحكم بها وإعادة حالة غير صالحة لا يمكنك إصلاحها.

من الملائم عند الحالات التي تتوقع فيها حدوث فشل أن تُعيد القيمة `Result` بدلًا من استدعاء `panic!`، مثل تمرير بيانات مشوّهة وتحليلها، أو طلب HTTP يُعيد حالة تُشير إلى وصول حدٍ معدّل ما rate limit، وفي

هذه الحالة تُشير إعادة القيمة `Result` إلى أن الخطأ الذي حصل هو خطأ متوقع حدوثه عند استدعاء الشيفرة البرمجية، التي ستحدّد كيفية التعامل مع الخطأ.

يجب أن تتأكد شيفرتك البرمجية من القيم الصالحة أولًا وتهلع إذا لم تكن القيم صالحةً عندما تُجري الشيفرة عمليةً يمكن أن تضع المستخدم في خطرٍ ما عند استدعائها باستخدام قيم غير صالحة، وذلك لأسباب تتعلق بالأمان، إذ أن محاولة إجراء عمليةٍ على بيانات غير صالحة قد تعرّض شيفرتك البرمجية لثغرات أمنية، وهذا هو السبب الرئيس لاستدعاء المكتبة القياسية للماكرو `panic!` إذا حاولت الوصول إلى عنصر يقع خارج حدود الذاكرة out-of-bound memory access، مثل محاولة الوصول إلى حيز ذاكرة لا ينتمي إلى هيكل البيانات الحالي، وهي مشكلة أمنية شائعة.

تحتوي الدوال عادةً على ما يُدعى بالعقود contracts، إذ أن تصرُّف الدوال مضمونٌ فقط في حال كان الدخل يوافق بعض المتطلبات، فإذا أُخلّ بالعقد، سنحصل على حالة هلع وهذا الأمر منطقي، لأن الإخلال بالعقد يعني أن هناك خطأ من طرف مستدعي الشيفرة البرمجية، وهو نوع من الأخطاء لا تجد للشيفرة البرمجية المستدعية ضرورةً لمعالجته، وفي الحقيقة لا توجد هناك أي طريقة منطقية حتى تتعافى الشيفرة البرمجية المُستدعية، إذ يقع إصلاح الخطأ على عاتق المبرمج الذي استدعى هذه الشيفرة البرمجية. تسبب عقود الدوال عند خرقها حالة هلع، وتُشرح العقود عادةً في توثيق الواجهة البرمجية API الخاص بالدالة.

الحصول على الكثير من الأخطاء في جميع الدوال أمرٌ رتيب ومزعج، ولحس الحظ يمكننا استخدام نظام أنواع لغة رست (وبالتالي التحقق من الأنواع من المصرّف) لإجراء العديد من الفحوصات نيابةً عنك. إذا كانت الدالة تقبل نوعًا محددًا للمعامل، فيمكنك متابعة منطق شيفرتك البرمجية على أساس هذا النوع وأنت مطمئن، لأن المصرف سيتأكد من استخدام النوع الصحيح دومًا، فعلى سبيل المثال سيتوقع برنامجك قيمةً ما بدلًا من عدم وجود قيمة إذا كان لديك نوع معيّن بدلًا من `Option`، وعندها لن يتوجب على شيفرتك البرمجية أن تتعامل مع متغاير `Some` و `None` على حدى، بل ستتحقق من حالة واحدة فقط وهو وجود قيمة، وبالتالي لن تستطيع تصريف أي شيفرة برمجية لا تمرر قيمة إلى الدالة ولن يتوجب عندها التحقق من أن الدالة تحصل على قيمة خلال وقت التشغيل. مثال على ما سبق هو استخدام نوع العدد الصحيح عديم الإشارة unsigned integer مثل `u32`، وهذا يضمن لك أن تكون قيمة المعامل موجبةً دومًا.

9.3.4 إنشاء أنواع مخصصة بهدف التحقق

دعنا نتطرق إلى فكرة استخدام نظام أنواع رست للتأكد من أن القيمة صالحة، وذلك بإنشاء نوع مخصص للتحقق. تذكّر لعبة تخمين الأرقام التي أنشأناها سابقًا في الفصل 2، إذ طلبنا من المستخدم حينها تخمين رقم يقع بين 1 و100، ولم نتحقق حينها من أن تخمين المستخدم كان ضمن هذا المجال قبل أن نتحقق من أن التخمين يطابق الرقم السري (الإجابة الصحيحة)، إذ أننا تأكدنا وقتها أن القيمة موجبة فقط، وفي هذه الحالة فإن البرنامج لن يكون خاطئًا، لأن طباعة "تخمينك كبير" أو "تخمينك صغير" سيعمل كم هو مطلوب، إلا أنه من الأفضل تحسين البرنامج بحيث يرشد المستخدم نحو تخمين صحيح في حال كان تخمينه ضمن المجال

ويختلف سلوك البرنامج في حال كان التخمين خارج المجال المطلوب، كما هو الحال عندما يكتب المستخدم أحرفًا بدل أرقام.

نستطيع تحقيق ذلك عن طريق النظر إلى تخمين المستخدم على أنه نوع `i32` بدلًا من `u32` للسماح بقيم سالبة ومن ثم التأكد من أن الرقم متواجد ضمن المجال كما يلي:

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    loop {
        // --snip--

        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");

        let guess: i32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        if guess < 1 || guess > 100 {
            println!("The secret number will be between 1 and 100.");
            continue;
        }

        match guess.cmp(&secret_number) {
```

```
            // --snip--
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

يتحقق تعبير `if` فيما إذا كانت القيمة خارج المجال ويخبر المستخدم بالمشكلة، ثم يستدعي `continue` للبدء بالتكرار التالي من الحلقة لسؤال المستخدم عن التخمين مرةً أخرى. يمكننا الاستمرار بتنفيذ المقارنات بعد تعبير `if` وذلك بين `guess` والرقم السري بمعرفة أن `guess` الآن هي ضمن المجال من 1 إلى 100، إلا أن الحل السابق ليس بالحل المثالي، فوجود عملية تحقق مثل هذه في برنامج يعمل فقط على القيم التي تقع ما بين 1 و100 عملية رتيبة، إذ علينا تكرار عملية التحقق داخل كل دالة في هذا البرنامج، كما قد يؤثر ذلك على أداء البرنامج.

يمكننا إنشاء نوع جديد عوضًا عمّا سبق وأن نضع عملية التحقق في دالة، ومن ثم يمكننا إنشاء نسخة عن النوع بدلًا من تكرار عملية التحقق في كل مكان، وبهذه الطريقة يصبح استخدام الدوال للنوع الجديد أكثر أمانًا في بصمتها signature، إذ أنها ستستخدم القيم المسموحة في البرنامج فقط. توضح الشيفرة 13 طريقةً لتعريف النوع `Guess` الذي يُنشئ نسخةً من `Guess` إذا استقبلت الدالة `new` قيمةً ما بين 1 و100.

```
#![allow(unused)]
fn main() {
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.",
value);
        }
```

```
            Guess { value }
        }

        pub fn value(&self) -> i32 {
            self.value
        }
    }
}
```

[الشيفرة 13: نوع Guess سيستمر فقط في حال كانت القيم بين 1 و100]

نعرّف هيكلًا أولًا باسم Guess يحتوي على الحقل value الذي يخزن بداخله قيمة من النوع i32، وهو الحقل الذي سنخزّن فيه تخمين المستخدم. نطبّق بعدها دالةً مرتبطة بالهيكل Guess تدعى new، إذ تُنشئ هذه الدالة نسخةً من قيم Guess وهي معرفة بحيث تتلقى معاملًا واحدًا يُدعى value نوعه i32 وأن تعيد هيكلًا من النوع Guess. تفحص الشيفرة البرمجية داخل الدالة new القيمة value لتتأكد من أنها تقع ما بين 1 و100، وإذا لم تحقق value هذا الشرط، نستدعي الماكرو panic! الذي سينبّه المبرمج الذي يكتب الشيفرة البرمجية المُستدعية للدالة أن شيفرته البرمجية تحتوي على خطأ يجب إصلاحه لأن إنشاء Guess بقيمة value خارج النطاق المحدد سيخرق الاتفاق الذي تعتمد عليه الدالة Guess::new.

يجب أن تُناقش الحالات التي قد تهلع فيها الدالة Guess::new في توثيق الواجهة البرمجية العلني الخاص بالدالة، وسنغطّي اصطلاحات التوثيق التي تُشير فيها لاحتمال حصول حالة هلع panic! في توثيق الواجهة البرمجية API لاحقًا.

نُنشئ هيكل Guess جديد بحقل value قيمته مساوية إلى المعامل value ونُعيد Guess إذا لم تتخطى value الاختبار، ثم نطبّق تابعًا يدعى value يستعير self ولا يمتلك أي معاملات أخرى، ويعيد قيمةً من نوع i32، ويُدعى هذا النوع من التوابع بالجالب getter لأن الهدف منه يكمن في الحصول على بيانات من حقوله وإعادتها. هذا التابع العام public method مهم لأن الحقل value في الهيكل Guess هو هيكل خاص private، ومن المهم لحقل value أن يكون خاصًا بحيث لا يُسمح للشيفرة البرمجية التي تستخدم الهيكل Guess بأن تضبط قيمة value مباشرةً؛ إذ يجب على الشيفرة البرمجية التي تقع خارج الوحدة module أن تستخدم الدالة Guess::new لإنشاء نسخة من Guess للتأكد من أنه لا توجد أي طريقة أن يحتوي الهيكل Guess على قيمة حقل value دون فحصها ومطابقتها للشروط ضمن الدالة Guess::new.

يمكن لدالة تحتوي معاملًا أو تعيد أرقام ضمن المجال 1 إلى 100 التصريح ضمن بصمتها بأنها أنها تأخذ أو تعيد Guess بدلًا عن i32 وعندها لن تحتاج الدالة إلى إجراء أي عمليات تحقّق إضافية ضمنها.

9.4 خاتمة

صُمّمت مزايا التعامل مع الأخطاء في رست لتساعدك في كتابة شيفرة برمجية أكثر متانة. يُشير الماكرو `panic!` إلى أن برنامجك قد دخل في حالة لا يستطيع فيها التعامل مع الخطأ مما يعطيك خيار إيقاف البرنامج بدلًا من الاستمرار مع القيم غير الصالحة أو الخاطئة. يستخدم المعدد `Result` نظام أنواع رست للإشارة إلى أن العمليات قد تفشل بطريقة يمكن لشيفرتك البرمجية أن تتعافى منها، ويمكنك استخدام `Result` لتخبر الشيفرة البرمجية التي استدعت شيفرتك البرمجية أنها بحاجة للتعامل مع حالات النجاح أو الفشل. سيجعل استخدام `panic!` و `Result` في الحالات المناسبة من شيفرتك البرمجية أكثر موثوقية في وجه المشاكل التي ستحصل بين حينٍ وآخر.

الآن وبعد أن رأيت طرقًا مفيدة لاستخدام الأنواع المعمّمة generics بواسطة المكتبة القياسية مثل المعدد `Option` و `Result`، حان الوقت لنتكلم عن كيفية عمل الأنواع المعممة وكيف يمكنك استخدامها في شيفرتك البرمجية.

دورة تطوير تطبيقات الويب
باستخدام لغة Ruby

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

10. الأنواع المعممة Generic Types والسمات Traits ودورات الحياة Lifetimes

تحتوي كل لغة برمجة على عدد من الأدوات للتعامل مع تكرار المفاهيم بفعالية، وتمثّل الأنواع المعممة generic types في لغة رست هذه الأداة، والتي تتضمن بدائل مجرّدة لأنواع حقيقية concrete أو خاصيات أخرى. يمكننا التعبير عن سلوك الأنواع المعممة أو كيف ترتبط مع أنواع معممة أخرى دون معرفة نوع القيمة التي ستكون بداخلها عند تصريف وتشغيل الشيفرة البرمجية.

يمكن أن تأخذ الدوال بعض الأنواع المعممة معاملات لها بدلًا من أنواع حقيقية، مثل `i32` أو `String` بطريقة مماثلة لما ستكون عليه دالة تأخذ معاملات بقيم غير معروفة لتشغيل الشيفرة البرمجية ذاتها باستخدام عدّة قيم حقيقية. استخدمنا في الحقيقة الأنواع المعممة سابقًا في الفصل 6 عندما تكلمنا عن `Option<T>` وفي الفصل 8 عن `Vec<T>` و `HashMap<K, V>` وفي الفصل 9 عن `Result<T, E>`. سننظر في هذا الفصل إلى كيفية تعريف نوع أو دالة أو تابع خاص بك باستخدام الأنواع المعممة.

دعنا ننظر أولًا إلى كيفية استخراج دالة ما للتقليل من عملية تكرار الشيفرات البرمجية، ثمّ سنستخدم الطريقة ذاتها لإنشاء دالة معمّمة باستخدام دالتين مختلفتين فقط بأنواع معاملاتهما، كما سنشرح أيضًا كيفية استخدام الأنواع المعممة ضمن تعريف هيكل struct أو تعريف مُعدّد enum.

سنتعلّم بعدها كيفية استخدام السمات traits لتعريف السلوك في سياق الأنواع المعممة، إذ يمكنك استخدام السمات مع الأنواع المعممة لتقييد قبول أنواع تحتوي على سلوك معين بدلًا من احتوائها لأي نوع.

وأخيرًا، سنناقش دورات الحياة lifetimes وهي مجموعة من الأنواع المعممة التي تعطي المصرّف معلومات حول ارتباط المراجع references ببعضها بعضًا، وتسمح لنا دورات الحياة بإعطاء المصرف كمًا كافيًا

من المعلومات عن القيم المُستعارة borrowed values حتى يتسنّى له التأكد من المراجع التي ستكون صالحة في أكثر من موضع مقارنةً بالمواضع التي يمكن للمصرف التحقق منها بنفسه دون مساعدتنا.

10.1 إزالة التكرار باستخراج دالة

تسمح لنا الأنواع المعممة باستبدال أنواع محددة مع موضع مؤقت placeholder يمثل أنواع عدّة بهدف التخلص من الشيفرة البرمجية المتكررة. دعنا ننظر إلى كيفية التخلص من الشيفرات البرمجية المتكررة دون استخدام الأنواع المعممة قبل أن نتكلم عن كيفية كتابتها، وذلك عن طريق استخراج الدالة التي ستستبدل قيمًا معيّنة بموضع مؤقت يمثّل قيمًا متعددة، ثم نطبق الطريقة ذاتها لاستخراج دالة معممة. سنبدأ بالتعرف على الشيفرات البرمجية المتكررة التي يمكن أن تستخدم الأنواع المعممة عن طريق التعرف على الشيفرات البرمجية المتكررة الممكن استخراجها إلى دالة.

دعنا نبدأ ببرنامج قصير موضّح في الشيفرة 1، إذ يعثر هذا البرنامج على أكبر رقم موجود في قائمة ما.

اسم الملف: src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);
}
```

[الشيفرة 1: العثور على أكبر رقم في قائمة من الأرقام]

نخزّن هنا قائمة من الأرقام الصحيحة في المتغير `number_list` ونعيّن مرجعًا إلى العنصر الأول في القائمة ضمن متغير يدعى `largest`، ثمّ نمرّ على العناصر الموجودة في القائمة بالترتيب ونفحص إذا كان الرقم الحالي أكبر من الرقم الذي خزّننا مرجعه في `largest`؛ فإذا كانت الإجابة نعم، نستبدل المرجع السابق بمرجع الرقم الحالي؛ وإلا -إذا كان الرقم الحالي أصغر أو تساوي من الرقم `largest`- لا نغير قيمة المتغير وننتقل إلى

الرقم الذي يليه في القائمة. يجب أن يمثل `largest` مرجعًا لأكبر رقم في القائمة بعد النظر إلى كل الأرقام، وهو في هذه الحالة 100.

تغيّرت مهمتنا الآن: علينا كتابة برنامج يفحص أكبر رقم ضمن قائمتين مختلفتين من الأرقام، ولفعل ذلك يمكننا نسخ الشيفرة البرمجية في الشيفرة 1 مرةً أخرى واستخدام المنطق ذاته في موضعين مختلفين ضمن البرنامج كما هو موضح في الشيفرة 2.

اسم الملف: src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);
}
```

[الشيفرة 2: شيفرة برمجية تجد أكبر رقم في قائمتين من الأرقام]

على الرغم من أن الشيفرة البرمجية السابقة تعمل بنجاح إلا أن نسخ الشيفرة البرمجية عملية رتيبة ومعرضة للأخطاء، علينا أيضًا أن نتذكر تعديل الشيفرة البرمجية في عدة مواضع إذا أردنا التعديل على منطق البرنامج.

لنُنشئ حلًا مجرّدًا للتخلص من التكرار وذلك بتعريف دالة تعمل على أي قائمة من الأرقام الصحيحة بتمريرها مثل معامل للدالة. يجعل هذا الحل من شيفرتنا البرمجية أكثر وضوحًا ويسمح لنا بالتعبير عن مفهوم العثور على أكبر رقم ضمن قائمة ما بصورةٍ مجرّدة.

نستخرج الشيفرة البرمجية التي تبحث عن أكبر عدد إلى دالة تدعى `largest` في الشيفرة 3، من ثم نستدعي الدالة لإيجاد أكبر عدد في القائمتين الموجودتين في الشيفرة 2، مما يمكننا من استخدام الدالة على أي قائمة تحمل عناصر من النوع `i32`.

اسم الملف: src/main.rs

```
fn largest(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let result = largest(&number_list);
    println!("The largest number is {}", result);
}
```

[الشيفرة 3: شيفرة برمجية مجردة لإيجاد أكبر رقم ضمن قائمتين]

تقبل الدالة `largest` معاملًا يدعى `list` ويمثّل أي شريحة slice حقيقية من قيم `i32`، ونتيجةً لذلك يمكننا استدعاء الدالة وتنفيذ الشيفرة البرمجية بحسب القيم المحددة التي نمررها للدالة.

اختصارًا لما سبق، إليك الخطوات التي اتبعناها للوصول من الشيفرة 2 إلى الشيفرة 3:

1. التعرف على الشيفرة البرمجية المتكررة.
2. استخراج الشيفرة البرمجية المتكررة إلى محتوى دالة وتحديد القيم التي نمررها للدالة والقيم التي تعيدها الدالة في بصمة الدالة signature.
3. تحديث مواضع نسخ الشيفرة البرمجية لتستدعي الدالة بدلًا من ذلك.

سنستخدم الخطوات ذاتها لاحقًا مع الأنواع المعممة للتقليل من الشيفرات البرمجية المكررة، إذ تسمح الأنواع المعممة للشيفرة البرمجية بالعمل على الأنواع المجردة بالطريقة ذاتها التي يتعامل فيها محتوى الدالة على قائمة مجرّدة بدلًا من قيم محددة.

على سبيل المثال، دعنا نفترض وجود دالتين: دالة تعثر على أكبر رقم في شريحة من قيم `i32` وأخرى تعثر على أكبر قيمة في شريحة من قيم `char`، كيف يمكننا التخلص من التكرار هنا؟ هذا ما سنناقشه تاليًا.

10.2 أنواع البيانات المعممة Generic Data Types

نستخدم الأنواع المُعمَّمة لإنشاء تعاريف لعناصر مثل بصمات الدوال function signatures أو الهياكل structs، بحيث تمكننا من استخدام عدّة أنواع بيانات ثابتة. دعنا ننظر أولًا إلى كيفية تعريف الدوال والهياكل والمُعدّدات enums والتوابع methods باستخدام الأنواع المعممة، ثم سنناقش كيف تؤثر الأنواع المعممة على أداء الشيفرة البرمجية.

10.2.1 في تعاريف الدوال

نضع الأنواع المعممة عند تعريف دالة تستخدمها في بصمة الدالة function signature، وهو المكان الذي نحدد فيه عادةً أنواع بيانات المعاملات ونوع القيمة المُعادة، إذ يكسب ذلك شيفرتنا البرمجية مرونةً أكبر ويقدم مزايا أكثر للشيفرة البرمجية المُستدعية لدالتنا مع منع تكرار الشيفرة البرمجية في الوقت ذاته.

لنستمرّ في مثال الدالة `largest`، توضح الشيفرة 4 دالتين يعثران على أكبر قيمة في شريحة slice ما، وسنجمع هاتين الدالتين في دالة واحدة تستخدم الأنواع المعممة.

اسم الملف: src/main.rs

```
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];
```

```
    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
}
```

[الشيفرة 4: دالتان تختلفان عن بعضهما بالاسم ونوع البيانات في بصمتهما]

الدالة `largest_i32` هي الدالة التي استخرجناها من الشيفرة 3 التي تعثر على أكبر قيمة `i32` في شريحة، بينما تعثر الدالة `largest_char` على أكبر قيمة `char` في شريحة، ولدى الدالتين المحتوى ذاته، لذا دعنا نتخلص من التكرار باستخدام الأنواع المعممة مثل معاملات في دالة وحيدة.

نحتاج إلى تسمية نوع المعامل حتى نكون قادرين على استخدام عدة أنواع في دالة واحدة جديدة، كما نفعل عندما نسمّي قيم معاملات الدالة، ويمكنك هنا استخدام معرّف بمثابة اسم نوع معامل، إلا أننا سنستخدم `T` لأن أسماء المعاملات في لغة رست قصيرة اصطلاحًا وغالبًا ما تكون حرفًا واحدًا، كما أن اصطلاح رست في تسمية الأنواع قائمٌ على نمط سنام الجمل CamelCase، وتسمية النوع `T` هو اختصار لكلمة النوع "type" وهو الخيار الشائع لمبرمجي لغة رست.

علينا أن نصرّح عن اسم المعامل عندما نستخدمه في متن الدالة وذلك في بصمة الدالة حتى يعرف المصرّف معنى الاسم، كما ينبغي علينا بصورةٍ مشابهة تعريف اسم نوع المعامل في بصمة الدالة قبل أن نستطيع استخدامه داخلها. لتعريف الدالة المعممة `largest` نضع تصاريح اسم النوع داخل قوسين مثلثين `<>` بين اسم الدالة ولائحة المعاملات بالشكل التالي:

```
fn largest<T>(list: &[T]) -> &T {
```

نقرأ التعريف السابق كما يلي: الدالة `largest` هي دالة معممة تستخدم نوعًا ما اسمه `T`، ولدى هذه الدالة معاملٌ واحدٌ يدعى `list` وهو قائمة من القيم نوعها `T`، وتعيد الدالة `largest` مرجعًا إلى قيمة نوعها أيضًا `T`.

توضح الشيفرة 5 تعريف الدالة المُدمجة باستخدام نوع البيانات المعمم في بصمتها، كما توضح الشيفرة أيضًا كيفية استدعاء الدالة باستخدام شريحة من قيم `i32` أو من قيم `char`. لاحظ أن الشيفرة البرمجية لم تُصرّف بعد، إلا أننا سنصلح ذلك لاحقًا.

اسم الملف: src/main.rs

```
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

[الشيفرة 5: دالة `largest` تستخدم معاملات من أنواع معممة؛ إلا أن الشيفرة لا تُصرَّف بنجاح بعد]

إذا صرّفنا الشيفرة البرمجية السابقة، سنحصل على الخطأ التالي:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0369]: binary operation `>` cannot be applied to type `&T`
 --> src/main.rs:5:17
  |
| 5 |         if item > largest {
  |            ---- ^ ------- &T
  |            |
  |            &T
  |
help: consider restricting type parameter `T`
  |
| 1 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> &T {
  |             ++++++++++++++++++++++

For more information about this error, try `rustc --explain E0369`.
error: could not compile `chapter10` due to previous error
```

تـذكر رسـالة الخطـأ المسـاعدة `std::cmp::PartialOrd` وهي سـمة trait، وسـنتحدث عن السـمات لاحقًا. يكفي معرفتك حتى اللحظـة أن مفـاد الخطـأ هـو أن محتـوى الدالـة `largest` لن يعمـل لجميـع الأنـواع المحتملة للنوع `T`، وذلك لأننا نريـد مقارنـة قيم النـوع `T` في محتـوى الدالـة ويمكننـا الآن اسـتخدام أنـواع يمكن

لقيمها أن تُرتَّب. يمكننا لتمكين المقارنات استخدام السمة `std::cmp::PartialOrd` في المكتبة القياسية على الأنواع. إذا اتبعنا النصيحة الموجودة في رسالة الخطأ فسنحدّ من الأنواع الصالحة في `T` إلى الأنواع التي تطبّق السمة `PartialOrd`، وسيُصرَّف المثال بنجاح لأن المكتبة القياسية تطبّق السمة `PartialOrd` على كلٍ من النوعين `i32` و `char`.

10.2.2 في تعاريف الهياكل

يمكننا أيضًا تعريف الهياكل، بحيث تستخدم أنواع معممة مثل معامل ضمن حقل أو أكثر باستخدام `<>`. نعرّف في الشيفرة 6 هيكل `Point<T>` يحتوي على الحقلين `x` و `y` وهي قيم إحداثيات من أي نوع.

اسم الملف: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

[الشيفرة 6: هيكل `Point<T>` يخزن بداخله القيمتين `x` و `y` نوعهما `T`]

طريقة الكتابة في استخدام الأنواع المعممة في تعريف الهيكل مشابهة لطريقة الكتابة المستخدمة في تعاريف الدالة سابقًا، إذ نصرح أولًا عن اسم نوع المعامل داخل أقواس مثلثة بعد اسم الهيكل، ثم نستخدم النوع المعمم في تعريف الهيكل في المواضع التي نحدد فيها أنواع بيانات ثابتة في حالات أخرى.

لاحظ أننا استخدمنا نوعًا معممًا واحدًا فقط لتعريف `Point<T>` وبالتالي يخبرنا هذا التعريف أن الهيكل `Point<T>` هو هيكل معمم باستخدام نوع `T` وأن الحقلين `x` و `y` يحملان النوع ذاته أيًا يكن. لن تُصرَّف الشيفرة البرمجية إذا أردنا إنشاء نسخة من الهيكل `Point<T>` يحمل قيمًا من أنواع مختلفة كما نفعل في الشيفرة 7.

اسم الملف: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}
```

```
fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

[الشيفرة 7: يجب أن يكون للحقلين x و y النوع ذاته لأنهما يحملان النوع المعمم ذاته T]

نخبر المصرف في هذا المثال عند إسنادنا القيمة العددية الصحيحة "5" إلى x أن النوع المعمم T سيكون عددًا صحيحًا لهذه النسخة من Point<T>. نحصل على خطأ عدم مطابقة النوع التالي عندما نحدد أن y قيمتها "4.0" وهي معرّفة أيضًا بحيث تحمل قيمة x ذاتها:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0308]: mismatched types
 --> src/main.rs:7:38
  |
|     let wont_work = Point { x: 5, y: 4.0 };
  |                                      ^^^ expected integer, found
floating-point number

For more information about this error, try `rustc --explain E0308`.
error: could not compile `chapter10` due to previous error
```

نستخدم معاملات الأنواع المعممة المتعددة لتعريف الهيكل Point بحيث يكون كلًا من x و y من نوع معمم ولكن مختلف. على سبيل المثال، نغيّر في الشيفرة 8 تعريف Point لتصبح دالةً معممةً تحتوي النوعين T و U، إذ يكون نوع x هو T و y من النوع U.

اسم الملف: src/main.rs

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
```

```
}
```

[الشيفرة 8: دالة `Point<T, U>` المعممة التي تحتوي على نوعين بحيث يكون لكلٍ من المتغيرين `x` و `y` نوع مختلف]

جميع نسخ `Point` الآن مسموحة، ويمكنك استخدام عدّة أنواع معممة مثل معاملات في تعريف الدالة إلا أن استخدام الكثير منها يجعل شيفرتك البرمجية صعبة القراءة. إذا احتجت كثيرًا من الأنواع المعممة في شيفرتك البرمجية فهذا يعني أنه عليك إعادة هيكلة شيفرتك البرمجية إلى أجزاء أصغر.

10.2.3 في تعاريف المعدد

نستطيع تعريف المعددات، بحيث تحمل أنواع بيانات معممة في متغيراتها variants كما هو الأمر في الهياكل. دعنا ننظر إلى مثال آخر باستخدام المعدد `Option<T>` الموجود ضمن المكتبة القياسية الذي ناقشناه سابقًا في الفصل 6:

```
enum Option<T> {
    Some(T),
    None,
}
```

يجب أن تفهم هذا التعريف بحلول هذه النقطة بمفردك، فكما ترى معدّد `Option<T>` هو معدد معمم يحتوي على النوع `T` ولديه متغيران: `Some` الذي يحمل قيمةً واحدةً من النوع `T` و `None` الذي لا يحمل أي قيمة. يمكننا التعبير عن المفهوم المجرّد للقيمة الاختيارية باستخدام المعدد `Option<T>`، ولأن `Option<T>` هو معدد معمم، فهذا يعني أنه يمكننا استخدامه بصورةٍ مجرّدة بغض النظر عن النوع الخاص بالقيمة الاختيارية.

يمكن للمعددات أن تستخدم أنواعًا معممةً متعددة أيضًا، والمعدد `Result` الذي استخدمناه سابقًا في الفصل 9 هو مثال على ذلك:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

المعدد `Result` هو معدد مُعمم يحتوي على نوعين، هما: `T` و `E`، كما يحتوي على متغايرين، هما: `Ok` الذي يحمل قيمة من النوع `T` و `Err` الذي يحمل قيمة من النوع `E`، يسهّل هذا التعريف عملية استخدام المعدد `Result` في أي مكان يوجد فيه عملية قد تنجح (في هذه الحالة إعادة قيمة من نوع ما `T`)، أو قد تفشل (في هذه الحالة إعادة خطأ من قيمة ما `E`)، وهذا هو ما استخدمناه لنفتح الملف في الشيفرة 3 من الفصل 9 عندما كان النوع `T` يحتوي على النوع `std::fs::File` عند فتح الملف بنجاح وكان يحتوي `E` على النوع `std::io::Error` عند ظهور مشاكل في فتح الملف.

يمكنك اختصار حالات التكرار عندما تصادف حالات في شيفرتك البرمجية تحتوي على تعاريف هياكل ومعددات مختلفة فقط بنوع القيمة التي يحمل كل منها، وذلك عن طريق استخدام الأنواع المعممّة عوضًا عنها.

10.2.4 في تعاريف التابع

يمكننا تطبيق التوابع على الهياكل والمعددات (كما فعلنا سابقًا في الفصل 5) واستخدام الأنواع المعممة في تعريفها أيضًا. توضح الشيفرة 9 الهيكل `Point<T>` الذي عرفناه في الشيفرة 6 مصحوبًا بتابع يدعى `x` داخله.

اسم الملف: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

[الشيفرة 9: تطبيق تابع تدعى `x` على الهيكل `Point<T>` وهو تابع يعيد مرجعًا إلى الحقل `x` الذي نوعه `T`]

عرّفنا هنا تابعًا يدعى `x` داخل `Point<T>` يعيد مرجعًا إلى البيانات الموجودة في الحقل `x`. لاحظ أنه علينا التصريح عن `T` قبل `impl` حتى يتسنى لنا استخدام `T` لتحديد أننا نطبّق التوابع الموجودة في النوع `Point<T>`. تتعرّف رست على وجود النوع بين أقواس مثلثة في `Point` على أنه نوع معمّم وذلك بالتصريح عن `T` على أنه نوع مُعمّم بعد `impl` بدلًا عن النظر إلى النوع على أنه نوع ثابت. يمكننا اختيار اسم مختلف عن اسم معامل النوع المعمم المصرح في تعريف الهيكل لمعامل النوع المعمم هذا، إلا أن استخدام الاسم ذاته هي الطريقة الاصطلاحية. تُعرَّف التوابع المكتوبة ضمن `impl` التي تصرّح عن النوع المعمّم ضمن أي نسخة من هذا النوع بغض النظر عن النوع الثابت الذي يستبدل هذا النوع المعمم في نهاية المطاف.

يمكننا أيضًا تحديد بعض القيود على الأنواع المعممة عند تعريف التوابع الخاصة بالنوع، فيمكننا مثلًا تطبيق تابع على نسخ `Point<f32>` فقط بدلًا من نسخ `Point<T>` التي تحتوي على أي نوع مُعمّم. نستخدم في الشيفرة 10 النوع الثابت `f32` وبالتالي لا نصرّح عن أي نوع بعد `impl`.

اسم الملف: src/main.rs

```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

[الشيفرة 10: كتلة `impl` تُطبَّق فقط على هيكل بنوع ثابت معين موجود في معامل النوع المعمم `T`]

تشير الشيفرة البرمجية السابقة إلى أن النوع `Point<f32>` سيتضمن التابع `distance_from_origin`، لكن لن تحتوي النسخ الأخرى من `Point<T>`، إذ تمثّل `T` نوعًا آخر ليس `f32` على تعريف هذا التابع داخلها. يقيس هذا التابع مسافة النقطة عن مبدأ الإحداثيات (0.0, 0.0) ويستخدم عمليات حسابية متاحة فقط لأنواع قيم العدد العشري floating point.

لا تطابق معاملات النوع المُعمم في تعريف الهيكل معاملات النوع المعمم الموجودة في بصمة الهيكل نفسه دومًا. لاحظ أننا نستخدم النوعين المعمّمين `X1` و `Y1` في الشيفرة 11 اللذين ينتميان إلى الهيكل `Point` و `X2` و `Y2` لبصمة التابع `mixup` لتوضيح المثال أكثر. تُنشئ نسخة `Point` جديدة باستخدام قيمة `x` من `self Point` (ذات النوع `X1`) وقيمة `y` من النسخة `Point` التي مرّرناها (ذات النوع `Y2`).

اسم الملف: src/main.rs

```
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
```

```
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

[الشيفرة 11: تابع يستخدم أنواع معممة مختلفة من تعريف الهيكل]

عرّفنا في `main` الهيكل `Point` الذي يحتوي على النوع `i32` للحقل `x` بقيمة 5، وحقل من النوع `f64` يدعى `y` بقيمة 10.4. يمثل المتغير `p2` هيكلًا من النوع `Point` يحتوي على شريحة سلسلة نصية string slice داخله في الحقل `x` بقيمة "Hello"، وقيمة من النوع `char` في الحقل `y` بقيمة c.

يعطينا استدعاء `mixup` على النسخة `p1` باستخدام `p2` مثل معامل `p3`، وهو هيكل سيحتوي داخله على قيمة من النوع `i32` في الحقل `x` لأن `x` أتى من `p1`، وسيحتوي `p3` على حقل `y` داخله قيمة من نوع `char` لأن `y` أتى من `p2`، وبالتالي سيطبع استدعاء الماكرو `println!` التالي:

```
p3.x = 5, p3.y = c
```

كان الهدف من هذا المثال توضيح حالة يكون فيها المعاملات المعمّمة مصرّح عنها في `impl` وبعضها الآخر مصرّح عنها في تعريف التابع، إذ أنّ المعاملات المعممة `X1` و`Y1` مصرّحٌ عنهما هنا بعد `impl` لأنهما يندرجان تحت تعريف الهيكل، بينما تصريح المعاملين `X2` و `Y2` كان بعد `fn mixup` لأنهما متعلقان بالتابع فقط.

10.2.5 تأثير استخدام المعاملات المعممة على أداء الشيفرة البرمجية

قد تتسائل عمّا إذا كان هناك تراجع في أداء البرنامج عند استخدام الأنواع المعمّاة مثل معاملات، والخبر الجيد هنا أن استخدام الأنواع المعمّاة لن يجعل من البرنامج أبطأ ممّا سيكون عليه إذا استخدمت أنواعًا ثابتة.

تنجح رست بتحقيق ذلك عن طريق إجراء عملية توحيد شكل monomorphization الشيفرة البرمجية باستخدام الأنواع المعماة وقت التصريف؛ وعملية توحيد الشكل هي عملية تحويل الشيفرة البرمجية المعممة إلى شيفرة برمجية محددة عن طريق ملئها بالأنواع الثابتة المستخدمة عند التصريف، ويعكس المصرف في هذه

المرحلة ما يفعله عندما يُنشئ دالة معمّاة في الشيفرة 5؛ إذ ينظر المصرّف إلى الأماكن التي يوجد بها شيفرة برمجية معماة ويولّد شيفرة برمجية تحتوي على أنواع ثابتة تُستدعى منها الشيفرة البرمجية المعمّاة.

دعنا ننظر إلى كيفية عمل هذه الخطوة باستخدام المعدد المعمم `Option<T>` الموجود في المكتبة القياسية:

```
let integer = Some(5);
let float = Some(5.0);
```

تُجري رست عملية توحيد الشكل عندما تصرَّف الشيفرة البرمجية السابقة، ويقرأ المصرف خلال العملية القيم التي استُخدمت في نسخ `Option<T>` ويتعرف على نوعين مختلفين من `Option<T>` أحدهما `i32` والآخر `f64`، وبالتالي يتحول التعريف المعمم للنوع `Option<T>` إلى تعريفين، أحدهما تعريف للنوع `i32` والآخر للنوع `f64` ويُستبدل التعريفان بالتعريف المعمّم.

هذا ما تبدو عليه الشيفرة البرمجية السابقة بعد إجراء عملية توحيد الشكل (يستخدم المصرف أسماءً مختلفة عمّا نستخدم هنا في المثال التوضيحي):

اسم الملف: src/main.rs

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

يُستبدل النوع المعمم `Option<T>` بتعاريف الأنواع المحددة عن طريق المصرف، ولأن رست تُصرف الشيفرة البرمجية المعممة إلى شيفرة برمجية ذات نوع ثابت لكل نسخة فلا يوجد هناك أي تراجع في أداء الشيفرة البرمجية عند استخدام الأنواع المعممة، إذ تعمل الشيفرة البرمجية عند تشغيلها بأداء مماثل لما قد

يكون عليه أداء الشيفرة البرمجية التي تكرّر كل تعريف يدويًا، وتجعل عملية توحيد الشكل من الأنواع المعممة في رست ميزة فعّالة جدًا عند وقت التشغيل.

10.3 السمات Traits: تعريف سلوك مشترك

يمكن أن تعرّف السمة وظيفة نوع محدد ويمكن مشاركتها مع عدّة أنواع، ويمكننا استخدام السمات لتعريف سلوك مشترك بطريقة مجردة، ويمكننا استخدام حدود السمة trait bounds لتحديد أن النوع المعمّم يمكن أن يكون أي نوع يمتلك سلوكًا محددًا.

> السمات مشابهة لميزة تدعى الواجهات interfaces في لغات برمجة أخرى، إلا أن هناك بعض الاختلافات.

10.3.1 تعريف سمة Trait

يتكون سلوك النوع من توابع يمكننا استدعائها على هذا النوع، ونقول أن عدّة أنواع تشارك السلوك ذاته إذا أمكننا استدعاء التوابع ذاتها على جميع هذه الأنواع، ويُعد تعريف السمة طريقةً لجمع بصمات التوابع method signatures لتعريف مجموعة من السلوكيات المهمة لتحقيق غرض ما.

على سبيل المثال، دعنا نفترض وجود عدّة هياكل تحمل أنواع وكميات مختلفة من النص، إذ يحمل الهيكل `NewsArticle` حقلًا لمحتوى إخباري في موقع معين، ويمكن أن تحتوي `Tweet` على نص طوله 280 محرفًا بالحد الأعظمي، إضافةً إلى البيانات الوصفية metadata التي تشير إلى كون التغريدة جديدة، أو إعادة تغريد retweet، أو رد على تغريدة أخرى.

نريد أن نُنشئ وحدة مكتبة مصرَّفة crate library تجمع الأخبار تدعى `aggregator`، بحيث تعرض ملخصًا للبيانات التي قد تجدها في نسخ `NewsArticle` أو `Tweet`، ثمّ سنستدعي الملخص لأي من النسخ باستدعاء التابع `summarize`. توضح الشيفرة 12 تعريف السمة العامة `Summary` التي تعبّر عن هذا السلوك.

اسم الملف: src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

[الشيفرة 12: سمة `Summary` تتألف من السلوك الموجود في التابع `summarize`]

نصرّح هنا عن سمة باستخدام الكلمة المفتاحية `trait` متبوعةً باسم السمة وهي `Summary` في هذه الحالة، كما نصرّح أيضًا عن السمة بكونها عامة `pub` بحيث تستخدم الوحدات المصرّفة هذه الوحدة المصرفة كما سنرى في الأمثلة القادمة. نصرّح عن بصمات التابع داخل القوسين المعقوصين curly brackets،

إذ تصف البصمات سلوك الأنواع التي تطبق هذه السمة، والتي هي في هذه الحالة `fn summarize(&self) -> String`.

بعد التصريح عن بصمة التابع، يمكننا استخدام الفاصلة المنقوطة بدلًا من الأقواس المعقوصة. ويجب على كل نوع ينفّذ هذه السمة أن يوفّر سلوكه المخصص لمتن التابع. سيفرض المصرّف أن أي نوع له السمة `Summary` سيكون له تابع باسم `summarize` مُعرّف بتلك البصمة المُحدَّدة.

يمكن أن تحتوي السمة عدّة توابع في متنها، إذ أن بصمات التوابع محتواة في كل سطر على حدة، وينتهي كل سطر بفاصلة منقوطة.

10.3.2 تطبيق السمة على نوع

الآن وبعد أن عرَّفنا البصمات المطلوبة لتوابع السمة `Summary` يمكننا تطبيقها على الأنواع الموجودة في مجمّع الوسائط aggregator media. توضح الشيفرة 13 تنفيذًا للسمة `Summary` في الهيكل `NewsArticle` الذي يستخدم كل من العنوان والمؤلف والمكان لإنشاء قيمة مُعادة من `summarize`. نعرّف من أجل الهيكل `Tweet` الدالة `summarize` بحيث تحصل على اسم المستخدم متبوعًا بالنص الكامل الموجود في التغريدة وذلك بفرض أن التغريدة محدودة بمقدار 280 محرف.

اسم الملف: src/lib.rs

```
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author,
self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
```

```
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

[الشيفرة 13: تطبيق السمة `Summary` على كل من النوعين `NewsArticle` و `Tweet`]

تطبيق سمة على نوع هي عملية مشابهة لتطبيق توابع اعتيادية، إلا أن الفارق هنا هو أننا نضع اسم السمة التي نريد تطبيقها بعد `impl`، ثم نستخدم الكلمة المفتاحية `for` ونحدد اسم النوع الذي نريد تطبيق السمة عليه. نضع داخل كتلة `impl` بصمات التابع المعرفة في تعريف السمة، وبدلًا من إضافة الفاصلة المنقوطة بعد كل بصمة سنستخدم الأقواس المعقوصة ونملأ داخلها متن التابع مع السلوك المخصص الذي نريد من توابع السمة أن تمتلكه لنوع معين.

الآن وبعد أن طبّقنا السمة `Summary` في وحدة المكتبة المصرفة على `NewsArtilce` و `Tweet`، يمكن لمستخدمي الوحدة المصرّفة استدعاء توابع السمة على نسخٍ من `NewsArticle` و `Tweet` بالطريقة ذاتها التي نستدعي بها توابع اعتيادية، إلا أن الفارق الوحيد هنا هو أن المستخدم يجب أن يُضيف السمة إلى النطاق scope إضافةً إلى الأنواع. إليك مثالًا عن كيفية استخدام وحدة المكتبة المصرفة `aggregator` من قِبل وحدة ثنائية مصرّفة binary crate:

```
use aggregator::{Summary, Tweet};

fn main() {
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
```

```
}
```

تطبع الشيفرة البرمجية السابقة ما يلي:

```
new tweet: horse_ebooks: of course, as you probably already know,
people
```

يُمكن أن تضيف الوحدات المصرّفة الأخرى المعتمدة على الوحدة المصرفة aggregator السمة Summary إلى النطاق لتطبيق Summary على أنواعها الخاصة، إلا أن القيد الوحيد هنا الذي يجب ملاحظته هو أنه يمكننا تطبيق السمة على نوع نريده فقط إذا كانت سمة واحدة على الأقل أو نوعًا واحدًا على الأقل محليًا local بالنسبة لوحدتنا المصرّفة؛ إذ يمكننا على سبيل المثال تطبيق سمات المكتبة القياسية مثل Display على نوع مخصص مثل Tweet بمثابة جزء من وظيفة وحدتنا المصرفة aggregator، لأن النوع Tweet هو محلي بالنسبة إلى الوحدة المصرفة aggregator، كما يمكننا أيضًا تطبيق Summary على النوع Vec<T> في الوحدة المصرفة aggregator لأن السمة Summary هي سمة محلية بالنسبة لوحدتنا المصرفة aggregator.

في المقابل، لا يمكننا تطبيق سمة خارجية على أنواع خارجية، فعلى سبيل المثال لا يمكننا تطبيق السمة Display على النوع Vec<T> داخل الوحدة المصرفة aggregator، لأن Display و Vec<T> ليستا معرفتين في المكتبة القياسية أو محليةً بالنسبة للوحدة المصرفة aggregator. يُعد هذا القيد جزءًا من خاصية تدعى **الترابط المنطقي coherence** وبالأخص قاعدة اليتيم orphan rule وتسمى القاعدة بهذا الاسم لأن نوع الأب غير موجود، وتتأكد هذه القاعدة من أن الشيفرة البرمجية الخاصة بالمبرمجين الآخرين لن تتسبب بعطل شيفرتك البرمجية والعكس صحيح، وبدون هذه القاعدة يمكن للوحدتين المصرفتين تطبيق السمة ذاتها على النوع ذاته، وعندها لن تستطيع رست معرفة أي من التنفيذين يجب استخدامه.

10.3.3 التنفيذات الافتراضية

من المفيد في بعض الأحيان تواجد سلوك افتراضي لبعض التوابع الموجودة في سمة ما أو جميعها بدلًا من طلب كتابة متن لكل التوابع ضمن كل نوع، بحيث يمكننا إعادة الكتابة على السلوك الافتراضي للتابع إذا أردنا تطبيق السمة على نوع معيّن.

نحدد في الشيفرة 14 سلسلةً نصيةً افتراضية للتابع summarize ضمن السمة Summary بدلًا من تعريف بصمة التابع كما فعلنا في الشيفرة 12.

اسم الملف: src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String {
```

```
            String::from("(Read more...)")
        }
    }
```

[الشيفرة 14: تعريف سمة Summary بتنفيذ افتراضي خاص بالتابع summarize]

نحدد كتلة impl فارغة بكتابة impl Summary for NewsArticle {} لاستخدام التنفيذ الافتراضي لتلخيص نسخ NewsArticle.

على الرغم من أننا لا نعرف بعد الآن التابع summarize على NewsArticle مباشرةً إلا أننا قدمنا متنًا افتراضيًا وحددنا أن NewsArticle تستخدم السمة Summary، ونتيجةً لذلك يمكننا استدعاء التابع summarize على نسخة من NewsArticle كما يلي:

```
    let article = NewsArticle {
        headline: String::from("Penguins win the Stanley Cup
Championship!"),
        location: String::from("Pittsburgh, PA, USA"),
        author: String::from("Iceburgh"),
        content: String::from(
            "The Pittsburgh Penguins once again are the best \
             hockey team in the NHL.",
        ),
    };

    println!("New article available! {}", article.summarize());
```

تطبع الشيفرة البرمجية السابقة ما يلي:

```
New article available! (Read more...)
```

لا يتطلب إنشاء تنفيذ افتراضي تعديل أي شيء بخصوص تنفيذ Summary على Tweet في الشيفرة 13، وذلك لأن طريقة الكتابة على التنفيذ الافتراضي مماثلة لصيغة تنفيذ تابع سمة لا يحتوي على تنفيذ افتراضي.

يمكن أن تستدعي التنفيذات الافتراضية توابع أخرى في السمة ذاتها حتى لو كانت التوابع الأخرى لا تحتوي على تنفيذ افتراضي، وبذلك يمكن أن تقدم السمة الكثير من المزايا المفيدة باستخدامها لتنفيذٍ محدد في جزء صغير منها، على سبيل المثال يمكننا أن نعرف السمة Summary بحيث تحتوي على تابع summarize_author يحتوي على تنفيذ داخله ومن ثم تابع summarize يحتوي على تنفيذٍ افتراضي يستدعي التابع summarize_author:

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}
```

لاستخدام هذا الإصدار من Summary علينا أن نعرف summarize_author عند تطبيق السمة على النوع:

```
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

يمكننا استدعاء summarize على نسخة من هيكل Tweet بعد تعريفنا التابع summarize_author، وعندها سيستدعي التنفيذ الافتراضي للتابع summarize تعريف التابع summarize_author الذي أضفناه، ولأننا كتبنا summarize_author فنحن منحنا للسمة Summary سلوكًا للتابع summarize دون كتابة المزيد من الأسطر البرمجية.

```
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
```

تطبع الشيفرة البرمجية السابقة ما يلي:

```
new tweet: (Read more from @horse_ebooks...)
```

لاحظ أنه ليس من الممكن استدعاء التنفيذ الافتراضي من تنفيذٍ كتبنا فوقه override لنفس التابع.

10.3.4 السمات مثل معاملات

الآن، وبعد أن تعلمنا كيفية تعريف وتطبيق السمات، أصبح بإمكاننا النظر إلى كيفية استخدام السمات لتعريف الدوال التي تقبل العديد من الأنواع المختلفة، وسنستخدم هنا السمة `Summary` التي طبقناها على النوعين `NewsArtilce` و `Tweet` في الشيفرة 13 لتعريف الدالة `notify` التي تستدعي التابع `summarize` على المعامل `item` وهو نوع ينفّذ السمة `Summary`. لتحقيق ذلك علينا أن نكتب صيغة `impl Trait` بالشكل التالي:

```
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

بدلًا من استخدام نوع ثابت للمعامل `item` نحدد الكلمة المفتاحية `impl` ومن ثم اسم السمة، إذ يقبل هذا المعامل أي نوع ينفّذ السمة التي حددناها. يمكننا استدعاء أي تابع في `notify` على `item` يحتوي على السمة `Summary` مثل `summarize`، إذ يمكننا استدعاء `notify` وتمرير أي نسخة من `NewsArticle` أو `Tweet`. لن تصرَّف الشيفرة البرمجية التي تستدعي الدالة باستخدام نوع آخر مثل `String` أو `i32` وذلك لأن الأنواع هذه لا تنفّذ `Summary`.

ا. صيغة حدود السمة

تكون صيغة `impl Triat` جيدة للاستخدامات البسيطة، إلا أنها طريقة مختصرة عن طريقة أطول تُعرف بحدود السمة trait bound، وتبدو على النحو التالي:

```
pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

تماثل هذه الكتابة الطويلة الكتابة في القسم السابق إلا أنها أطول، إذ أننا نضع حدود السمة في تصريح معاملات النوع المعمم بعد النقطتين وداخل أقواس مثلثة angle brackets.

تُعد صيغة `impl Trait` مناسبة وتجعل من شيفرتنا البرمجية أبسط في العديد من الحالات البسيطة إلا أن كتابة حدود السمة بشكلها الكامل تسمح لنا بتحديد تفاصيل أدق في بعض الحالات، على سبيل المثال يمكننا كتابة معاملين ينفّذان السمة `Summary` وكتابة هذا الأمر بصيغة `impl Trait`، وسيبدو بهذا الشكل:

```
pub fn notify(item1: &impl Summary, item2: &impl Summary) {
```

يُعد استخدام صيغة `impl Trait` ملائمًا إذا أردنا لهذه الدالة السماح للمعاملين `item1` و `item2` أن يكونا من نوعين مختلفين (طالما ينفّذ كلاهما `Summary`). إذا أردنا إجبار المعاملين على استخدام النوع ذاته يجب أن نستخدم حدود السمة على النحو التالي:

```
pub fn notify<T: Summary>(item1: &T, item2: &T) {
```

يقيّد النوع المعمّم `T` المحدد على أنه نوع لكل من المعاملين `item1` و `item2` الدالة بأنه يجب عليها قبول القيمتين فقط إذا كان كل من `item1` و `item2` لهما النوع ذاته.

ب. تحديد حدود سمة عديدة باستخدام صيغة +

يمكننا تحديد أكثر من حد سمة واحد، لنقل أننا نريد `notify` أن تستخدم تنسيق طباعة معيّن بالإضافة إلى `summarize` على `item`، عندها نحدد في تعريف `notify` أنه يجب على `item` أن تنفّذ كلًا من `Display` و `Summary` بنفس الوقت، ويمكننا فعل ذلك باستخدام الصيغة `+`:

```
pub fn notify(item: &(impl Summary + Display)) {
```

الصيغة `+` صالحة أيضًا مع حدود السمات على الأنواع المعممة:

```
pub fn notify<T: Summary + Display>(item: &T) {
```

يمكن لمتن الدالة `notify` أن يستدعي `summarize` مع استخدام `{}` لتنسيق `item` وذلك مع وجود حدّين للسمة.

ج. حدود سمة أوضح باستخدام بنى where

لاستخدام حدود سمة عديدة بعض السلبيات إذ أن كل نوع معمم يحتوي على حد سمة خاص به، لذا من الممكن للدوال التي تحتوي على عدة أنواع معممة مثل معاملات أن تحتوي الكثير من المعلومات بخصوص حدود السمة بين اسم الدالة ولائحة معاملاتها مما يجعل بصمة الدالة صعبة القراءة، ولذلك تحتوي رست على طريقة كتابة بديلة لتحديد حدود السمة داخل بنية `where` بعد بصمة الدالة، وبالتالي يمكننا استخدام البنية `where` على النحو التالي:

```
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
{
```

بدلًا من كتابة التالي:

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -
> i32 {
```

أصبحت الآن بصمة الدالة أكثر وضوحًا إذ تحتوي على اسم الدالة ولائحة معاملاتها والنـوع الـذي تُعيـده على سطر واحد بصورةٍ مشابهة لدالة لا تحتوي على الكثير من حدود السمة.

10.3.5 إعادة الأنواع التي تنفذ السمات

يمكننا أيضًا استخدام صيغة `impl Triat` في مكان الإعادة لإعادة قيمة من نوع ما يطبّق سمة، كمـا هـو موضح هنا:

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    }
}
```

نستطيع تحديد أن الدالة `returns_summarizable` تُعيد نوعًا يطبق السمة `Summary` باستخدام `impl` على أنـه نـوع مُعـاد دون تسـمية النـوع الثـابت، وفي هـذه الحالـة تعيـد الدالـة `Summary` القيمة `Tweet` إلا أنه ليس من الضروري أن تعلم الشـيفرة الـتي تسـتدعي الدالـة `returns_summarizable` بذلك.

إمكانية تحديد قيمة مُعادة فقط عن طريق السمة التي تطبقهـا مفيـد جـدًا، بـالأخص في سـياق المغلفـات closures والمكررات iterators وهما مفهومان سنتكلم عنهمـا لاحقًـا، إذ تُنشـئ المغلفـات والمكـررات أنواعًـا يعرفها المصرف فقط، أو أنواعًا يتطلب تحديدها كتابةً طويلةً إلا أن الصيغة `impl Trait` تسمح لك بتحديد أن الدالة تُعيد نوعًا ما يطبّق السمة `Iterator` دون الحاجة لكتابة نوع طويل.

يمكنك استخدام `impl Trait` فقط في حال إعادتك لنوع واحد، على سبيل المثال تُعيد الشيفرة البرمجية التالية إما `NewsArticle`، أو `Tweet` بتحديد النوع المُعاد باستخدام `impl Summary` إلا أن ذلك لا ينجح:

```
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
```

```
            headline: String::from(
                "Penguins win the Stanley Cup Championship!",
            ),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from(
                "The Pittsburgh Penguins once again are the best \
                 hockey team in the NHL.",
            ),
        }
    } else {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from(
                "of course, as you probably already know, people",
            ),
            reply: false,
            retweet: false,
        }
    }
}
```

إعادة إما `NewsArticle` أو `Tweet` ليس مسموحًا بسبب القيود الـتي يفرضـها اسـتخدام الصـيغة `impl` `Trait` وكيفية تنفيذها في المصرّف، وسنتكلم لاحقًا عن كيفية كتابة دالة تحقق هذا السلوك لاحقًا.

10.3.6 استخدام حدود السمة لتنفيذ التوابع شرطيا

يمكننا تنفيذ التوابع شرطيًا للأنواع التي تنفّذ سمةً ما عند استخدام هـذه السـمة بواسـطة كتلـة `impl` الـتي تستخدم الأنواع المعممة مثل معاملات. على سبيل المثال، ينفّذ النـوع `Pair<T>` في الشـيفرة 15 الدالـة `new` دومًا لإعادة نسخةٍ جديدة من `Pair<T>` (تذكر أن `self` هو اسم نوع مستعار للنـوع الموجـود في الكتلـة `impl` وهو `Pair<T>` في هذه الحالة)، إلا أنه في كتلة `impl` التالية ينفّـذ `Pair<T>` التـابع `cmp_display` فقـط إذا كان النوع `T` الداخلي ينفّذ السمة `PartialOrd` التي تمكّن المقارنة بالإضـافة إلى سـمة `Display` الـتي تمكّن الطباعة.

اسم الملف: src/lib.rs

```
use std::fmt::Display;
```

```
struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

[الشيفرة 15: تنفيذ توابع شرطيًا على نوع معمّم بحسب حدود السمة]

يمكننا أيضًا تنفيذ سمة شرطيًا لأي نوع ينفّذ سمةً أخرى، وتنفيذ السمة على أي نوع يحقق حدود السمة يسمّى بالتنفيذات الشاملة blanket implementations ويُستخدم بكثرة في مكتبة رست القياسية؛ على سبيل المثال تنفِّذ المكتبة القياسية السمة `ToString` على أي نوع ينفّذ السمة `Display`، وتبدو كتلة `impl` في المكتبة القياسية بصورةٍ مشابهة لما يلي:

```
impl<T: Display> ToString for T {
    // --snip--
}
```

ولأن المكتبة القياسية تستخدم التنفيذ الشامل هذا فيمكننا استدعاء التابع `to_string` المعرف باستخدام السمة `ToString` على أي نوع ينفّذ السمة `Display` على سبيل المثال يمكننا تحويل الأعداد الصحيحة إلى قيمة موافقة لها في النوع `String` وذلك لأن الأعداد الصحيحة تنفّذ السمة `Display`:

```
let s = 3.to_string();
```

يمكنك ملاحظة التنفيذات الشاملة في توثيق السمة في قسم "المنفّذين implementors".

تسمح لنا السمات وحدود السمات بكتابة شيفرة برمجية تستخدم الأنواع المعممة مثل معاملات، وذلك للتقليل من تكرار الشيفرة البرمجية، إضافةً إلى تحديدنا للمصرف بأننا نريد لقيمة معممة أن يكون لها سلوك معين، ويمكن للمصرف عندئذ استخدام معلومات حدود السمة للتحقق من أن جميع الأنواع الثابتة المستخدمة في شيفرتنا البرمجية تحتوي على السلوك الصحيح. سنحصل في لغات البرمجة المكتوبة ديناميكيًا dynamically typed على خطأ عند وقت التشغيل runtime إذا استدعينا تابعًا على نوع لم يعرّف هذا التابع، إلا أن رست تنقل هذه الأخطاء إلى وقت التصريف بحيث تجبرنا على تصحيح المشاكل قبل أن تُنفَّذ شيفرتنا البرمجية.

إضافةً لما سبق، لا يتوجب علينا كتابة شيفرة برمجية تتحقق من السلوك عند وقت التشغيل لأننا تحققنا من السلوك عند وقت التصريف، ويحسّن ذلك أداء الشيفرة البرمجية دون الحاجة للتخلي عن مرونة استخدام الأنواع المعممة.

10.4 التحقق من المراجع باستخدام دورات الحياة Lifetimes

تعدّ دورات الحياة نوعًا آخر من الأنواع المعمّمة generic وقد استعملناها سابقًا دون معرفتنا، إذ تتأكد دورات الحياة أن المراجع references صالحة طوال حاجتنا لها بدلًا من التأكد أن لنوع ما سلوك معيّن.

أغفلنا عند مناقشتنا للمراجع والاستعارة borrowing سابقًا في الفصل 4 أن كل مرجع له دورة حياة في رست، وهو نطاق المرجع الذي يبقى فيه صالحًا، وفي معظم الأحيان تكون دورات الحياة ضمنية واستنتاجية كما هو الحال بكون الأنواع استنتاجية، إذ أننا نحدد الأنواع فقط عندما يمكن وجود أكثر من نوع واحد في حالة ما، وبطريقة مشابهة، علينا أن نشير إلى دورات الحياة عندما ترتبط دورة حياة خاصة بمرجع بطرق عدّة مختلفة، إذ تتطلب منا رست تحديد العلاقة بين دورة حياة المعامل المعمّم للتأكد من أن المرجع الفعلي المستخدم وقت التشغيل سيكون صالحًا.

مفهوم الإشارة إلى دورات الحياة غير موجود في معظم لغات البرمجة، لذا قد تشعر بأن محتوى هذا الفصل غير مألوف بالنسبة لك، على الرغم من أننا لن نتكلم عن دورات الحياة بالتفصيل هنا بل سنتكلم عن الطرق الشائعة التي قد تصادف بها طريقة كتابة دورة حياة بحيث تألف هذا المفهوم.

10.4.1 منع المراجع المعلقة dangling references بدورات الحياة

هدف دورات الحياة الأساسي هو منع المراجع المعلّقة dangling references إذ تسبب للبرنامج إشارته إلى مرجع بيانات لا يتطابق مع البيانات التي نريدها، ألقِ نظرةً على البرنامج في الشيفرة 16، إذ يحتوي على نطاق خارجي وداخلي.

```
fn main() {
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

[الشيفرة 16: محاولة لاستخدام مرجع خرجت قيمته عن النطاق]

> تصرّح الأمثلة في الشيفرة 16 والشيفرة 17 والشيفرة 23 عن متغيرات دون إعطائها قيم أولية، لذا لا يوجد اسم المتغير في النطاق الخارجي. قد يبدو ذلك للوهلة الأولى تعارضًا مع مبدأ عدم وجود قيم فارغة null values في رست، إلا أننا سنحصل على خطأ عند التصريف إذا حاولنا استخدام متغير قبل منحه قيمة، وهو ما يؤكد عدم سماح رست بوجود قيم فارغة.

يصرح النطاق الخارجي عن متغير يدعى r دون إسناد قيمة أولية له، بينما يصرح النطاق الداخلي على متغير يدعى x بقيمة أولية 5. نحاول في النطاق الداخلي ضبط قيمة r لتصبح مرجعًا إلى القيمة x وعندما ينتهي النطاق الداخلي نحاول طباعة القيمة الموجودة في r. لن تُصرَّف هذه الشيفرة البرمجية وذلك لأن r يمثل مرجعًا لمتغير خرج عن النطاق قبل أن نستخدمه. إليك رسالة الخطأ:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `x` does not live long enough
 --> src/main.rs:6:13
  |
|         r = &x;
  |             ^^ borrowed value does not live long enough
|     }
  |     - `x` dropped here while still borrowed
|
|     println!("r: {}", r);
  |                       - borrow later used here

For more information about this error, try `rustc --explain E0597`.
```

```
error: could not compile `chapter10` due to previous error
```

لا "يعيش" المتغير x طويلًا، والسبب في ذلك هو أن x سيخرج عن النطاق عند انتهاء النطاق الـداخلي في السطر 7، بينما سيبقى r صالحًا في النطاق الخارجي لأن نطاقه أكبر، وعنـدها نقـول أنـه "سـيعيش" أطـول. إذا سـمحت رسـت لهـذه الشـيفرة البرمجيـة بالعمـل فهـذا يعـني أن r سـيمثل مرجعًـا لمكـان محـرر في الـذاكرة deallocated بعد خروج x من النطاق ولن يعمل أي شيء باستخدام r على النحو المطلـوب، إذًا كيـف تتحقـق رست من صلاحية هذه الشيفرة البرمجية؟ باستخدام مدقق الاستعارة borrow checker.

10.4.2 مدقق الاستعارة

لمصرّف رست مدقق استعارة يقارن بين النطاقات لتحديد أن جميـع عمليـات الاسـتعارة صـالحة، وتوضـح الشيفرة 17 إصدارًا مماثلًا للشيفرة 16 ولكن بتوضيح دورة الحياة لكل من المتغيرات.

```
fn main() {
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```

[الشيفرة 17: دورة حياة r يشار إليها باستخدام 'a بينما يشار إلى دورة حياة x باستخدام 'b]

أشرنا هنا إلى دورة حياة r باستخدام 'a ودورة حيـاة x باسـتخدام 'b، وكمـا تـرى، فـإن الكتلـة 'b الداخليـة أصغر بكثير من كتلة دورة حياة 'a الخارجية. تقارن رست عند وقت التصريف ما بين حجم دورتي الحيـاة هـاتين وتجد أن r لها دورة حياة 'a إلا أنها تمثل مرجعًا إلى موقع ذاكرة دورة حياته 'b، وبالتالي يُرفَض البرنـامج لأن 'b أقصر من 'a، أي أن الغرض الذي نستخدم المرجع إليه يعيش أقصر من المرجع ذاته.

نصلح الشيفرة البرمجية السابقة في الشيفرة 18، إذ لا يوجد لدينا مراجع معلقة بعد الآن، وتُصـرَّف الشـيفرة البرمجية بنجاح دون أي أخطاء.

```
fn main() {
    let x = 5;            // ----------+-- 'b
                          //           |
```

```
        let r = &x;           // --+-- 'a  |
                              //   |       |
        println!("r: {}", r); //   |       |
                              // --+       |
    }                         // ----------+
```

[الشيفرة 18: مرجع صالح لأن للبيانات دورة حياة أطول من المرجع]

دورة حياة `x` التي تدعى `'b` أكبر من `'a`، مما يعني أن `r` يمكن أن يمثل مرجعًا للمتغير `x`، لأن رست تعلم أن المرجع في `r` صالح ما دام `x` صالح.

الآن وبعد أن تعرفت على دورات حياة المراجع وكيف تحلل رست دورات الحياة للتأكد من أن المراجع ستكون دائمًا صالحة، حان وقت التعرف إلى دورات الحياة المعمّمة الخاصة بالمعاملات والقيمة المُعادة في سياق الدوال.

10.4.3 دورات الحياة المعممة في الدوال

دعنا نكتب دالةً تُعيد أطول شريحة نصية string slice من شريحتين نصيتين، إذ ستأخذ هذه الدالة شريحتين نصيتين وتُعيد شريحةً نصيةً واحدة.

اسم الملف: src/main.rs

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}
```

[الشيفرة 19: دالة `main` تستدعي الدالة `longest` لإيجاد أطول شريحة نصية من شريحتين نصيتين]

يجب أن تطبع الشيفرة 19 بعد تطبيق الدالة `longest` ما يلي:

```
The longest string is abcd
```

لاحظ أننا نريد أن تأخذ الدالة شرائح النصية وهي مراجع وليست سلاسل نصية لأننا لا نريد للدالة `longest` أن تأخذ ملكية معاملاتها. عُد للفصل 4 الذي تكلمنا فيه عن شرائح السلاسل النصية مثل معاملات لمعرفة المزيد حول سبب استخدامنا للمعاملات في الشيفرة 19 كما هي.

لن تُصرّف الشيفرة البرمجية إذا حاولنا كتابة الدالة `longest` كما هو موضح في الشيفرة 20.

اسم الملف: src/main.rs

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

[الشيفرة 20: تنفيذ الدالة `longest` الذي يُعيد أطول شريحة نصية من شريحتين إلا أنه لا يُصرَّف بنجاح]

نحصل على الخطأ التالي الذي يتحدث عن دورات الحياة:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
| fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime
parameter
  |
  = help: this function's return type contains a borrowed value, but
the signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
| fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++          ++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `chapter10` due to previous error
```

تساعدنا رسالة الخطأ في معرفة أن النوع المُعاد يجب أن يكون له معامل بدورة حياة معممة لأن رست لا تعلم إذا كان المرجع المُعاد يمثل مرجعًا إلى `x` أو `y`، وفي الحقيقة لا نعلم نحن أيضًا بدورنا لأن كتلة `if` في متن الدالة يُعيد مرجعًا للمتغير `x` وكتلة `else` تُعيد مرجعًا للمتغير `y`.

لا نعلم القيم الثابتة التي ستُمرر لهذه الدالة عندما نعرفها، لذا لا نعلم إذا ما كانت حالة `if` محققة أو حالة `else`، كما أننا لا نعرف دورة الحياة الثابتة للمراجع التي ستُمرر للدالة، لذا لا يمكننا النظر إلى النطاق كما فعلنا في الشيفرة 17 والشيفرة 18 للتأكد إذا ما كان المرجع المُعاد صالحًا دومًا، ولا يمكن لمدقق الاستعارة معرفة ذلك أيضًا لأنه لا يعرف أيّ من دورتي الحياة لكل من `x` و `y` ستكون مرتبطة بدورة الحياة الخاصة بالقيمة المُعادة؛ ولتصحيح هذا الخطأ نُضيف معاملًا ذا دورة حياة معممة يعرّف العلاقة ما بين المراجع حتى يستطيع مدقق الاستعارة إجراء تحليله.

10.4.4 طريقة كتابة دورة الحياة

لا تغيّر طريقة كتابة دورة الحياة على طول حياة المراجع، إذ تصف طريقة الكتابة العلاقة ما بين دورات الحياة لعدة مراجع بين بعضها بعضًا دون التأثير على دورات الحياة بذاتها. يمكن أن تقبل الدوال المراجع بأي دورة حياة بتحديد معامل دورة حياة معممة كما تقبل أي نوع عند تخصيص معامل من نوع معمم في بصمتها.

طريقة كتابة دورة الحياة غير مألوفة جدًا، إذ يجب أن تبدأ أسماء معاملات دورات الحياة بالفاصلة العليا `'` وعادةً ما تكون أسمائها قصيرة ومكتوبة بأحرف قصيرة كما هو الحال مع الأنواع المعممة. يستخدم معظم الناس الاسم `'a` بمثابة اسم أول دورة حياة، ومن ثم نضع معامل دورة الحياة بعد إشارة `&` الخاصة بالمرجع باستخدام المسافة للفصل بين طريقة كتابة دورة الحياة ونوع المرجع.

إليك بعض الأمثلة على ذلك: مرجع لقيمة من نوع `i32` دون معامل دورة حياة، ومرجع لقيمة من نوع `i32` بمعامل دورة حياة يدعى `'a` ومرجع قابل للتعديل mutable لقيمة من نوع `i32` بالاسم `'a` ذاته.

```
&i32        // مرجع
&'a i32     // مرجع مع دورة حياة صريحة
&'a mut i32 // مرجع قابل للتعديل مع دورة حياة صريحة
```

لا تعني كتابة دورة الحياة بمفردها بالشكل السابق الكثير، إذ أن الهدف من هذه الطريقة هو إخبار رست بعلاقة المراجع فيما بينها في معاملات دورة الحياة المعممة. دعنا ننظر إلى كيفية تحقيق ذلك في سياق الدالة `longest`.

10.4.5 توصيف دورة الحياة في بصمات الدالة

نحتاج للتصريح عن معاملات دورة الحياة المعممة داخل أقواس مثلثة حتى نستطيع استخدام توصيف دورة الحياة في بصمات الدوال، وذلك بين اسم الدالة وقائمة معاملاتها كما فعلنا سابقًا في معاملات النوع المعمم.

نريد من بصمة الدالة أن توضح القيود التالية: سيكون المرجع المُعاد صالح طالما أن كلا المعاملين صالحان؛ وهذه هي العلاقة بين دورات حياة المعاملات والقيمة المعادة. سنسمّي دورة حياة بالاسم `'a`، ثم نُضيفها لكل مرجع كما هو موضح في الشيفرة 21.

اسم الملف: src/main.rs

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

[الشيفرة 21: تعريف الدالة `longest` الذي يحدد أن دورة الحياة لجميع المراجع في بصمة الدالة هي `'a`]

يجب أن تعمل الشيفرة البرمجية السابقة بنجاح وأن تمنحنا النتيجة المرجوة عند استخدامها ضمن الدالة `main` كما فعلنا في الشيفرة 19 السابقة.

تخبر بصمة الدالة رست بأن الدالة تأخذ معاملين لبعض دورات الحياة `'a` وكلاهما شريحة نصية يعيشان على الأقل بطول دورة حياة `'a`، كما تخبر بصمة الدالة رست بأن شريحة السلسلة النصية المُعادة من الدالة ستعيش على الأقل بطول دورة الحياة `'a`، وهذا يعني عمليًا أن دورة حياة المرجع المُعاد من الدالة `longest` مماثلة لأقصر دورة حياة من دورات حياة القيم التي استخدمنا مراجعها في وسطاء الدالة، وهذه هي العلاقة التي نريد أن تستخدمها رست عند تحليل هذه الشيفرة البرمجية.

تذكر أننا لا نعدّل من دورات حياة القيم الممررّة أو المُعادة عندما نحدد دورة الحياة المعاملات في بصمة الدالة، وإنما نحدد أنه يجب على مدقق الاستعارة أن يرفض أي قيمة لا تتوافق مع القيود المذكورة. لاحظ أن الدالة `longest` لا تحتاج لمعرفة أيّ من المتغيرين `x` و `y` سيعيش لمدة أطول، بل فقط بحاجة لمعرفة أن نطاق ما سيُستبدل بدورة الحياة `'a` التي ستطابق بصمة الدالة.

نكتب توصيف دورات الحياة عند استخدامها مع الدوال في بصمة الدالة وليس في متنها، إذ يصبح توصيف دورة الحياة جزءًا من عقد contract الدالة كما هو الحال بالنسبة للأنواع ضمن بصمة الدالة. احتواء بصمة الدالة على عقد دورة الحياة يعني أن التحليل الذي يجريه مصرف رست سيصبح أبسط، وإذا وُجدت مشكلة بطريقة توصيف الدالة أو طريقة استدعائها يمكن لأخطاء المصرف أن تُشير إلى ذلك الجزء ضمن الشيفرة البرمجية والقيود التي خرقتها بصورةٍ أدقّ. إذا قدّم مصرف رست بعض الاستنتاجات حول العلاقة المقصودة لدورات الحياة، سيكون في هذه الحالة قادرًا على إعلامنا باستخدام الشيفرة الخاصة بنا وفق عدة خطوات لكنه سيكون بعيدًا عن السبب الحقيقي وراء المشكلة.

عند تمرير المراجع الثابتة إلى `longest` تكون دورة الحياة الثابتة المُستبدلة بدورة الحياة `'a` جزءًا من نطاق `x` الذي يتداخل مع نطاق `y`، بمعنى آخر، تحصل دورة الحياة المعممة `'a` على دورة حياة ثابتة مساوية إلى أصغر دورة حياة (أصغر دورة بين الدورتين الخاصة بالمتغير `y` والمتغير `x`).

دعنا ننظر إلى نتيجة استخدام توصيف دورة الحياة وكيف يقيّد ذلك من دالة `longest` بتمرير المراجع التي لها دورات حياة ثابتة مختلفة، وتمثّل الشيفرة 22 مثالًا مباشرًا على ذلك.

اسم الملف: src/main.rs

```
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {}", result);
    }
}
```

[الشيفرة 22: استخدام الدالة `longest` مع مراجع لقيم من نوع `String` تمتلك دورات حياة ثابتة مختلفة]

تكون القيمة `string1` صالحةً في المثال السابق حتى الوصول لنهاية النطاق الخارجي، بينما تبقى `string2` صالحة حتى نهاية النطاق الداخلي، وأخيرًا تمثل `result` مرجعًا لقيمة صالحة حتى نهاية النطاق الخارجية. نفّذ الشيفرة البرمجية السابقة وسترى أن مدقق الاستعارة لن يعترض على الشيفرة البرمجية وستُصرَّف وتطبع ما يلي:

```
The longest string is long string is long
```

دعنا نجرّب مثالًا يوضح أن دورة حياة المرجع في `result` يجب أن تكون أصغر من دورة حياة كلا الوسيطين؛ إذ سننقل التصريح عن المتغير `result` خارج النطاق الداخلي مع المحافظة على عملية إسناد قيمة إلى المتغير `result` داخل النطاق حيث توجد `string2`، ثم سننقل `println!` الذي يستخدم `result` خارج النطاق الداخلي بعد انتهائه. لن تُصرَّف الشيفرة 23 بنجاح.

اسم الملف: src/main.rs

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
```

```
    println!("The longest string is {}", result);
}
```

[الشيفرة 23: محاولة استخدام `result` بعد خروج `string2` من النطاق]

نحصل على رسالة الخطأ التالية عندما نحاول تصريف الشيفرة البرمجية:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `string2` does not live long enough
 --> src/main.rs:6:44
  |
|         result = longest(string1.as_str(), string2.as_str());
  |                                            ^^^^^^^^^^^^^^^
borrowed value does not live long enough
|     }
  |     - `string2` dropped here while still borrowed
|     println!("The longest string is {}", result);
  |                                          ------ borrow later used
here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` due to previous error
```

يوضح الخطأ أنه يجب على `result` أن يكون صالحًا حتى تُنفَّذ التعليمة `println!`، كما يجب على المتغير `string2` أن يكون صالحًا حتى نهاية النطاق الخارجي، وتعلم رست ذلك بسبب توصيفنا لدورات حياة معاملات الدالة والقيم المُعادة باستخدام معامل دورة الحياة ذاته `'a`.

يمكننا النظر إلى هذه الشيفرة البرمجية على أننا بشر ورؤية أن `string1` أطول من `string2` وبالتالي سيحتوي المتغير `result` على مرجع للمتغير `string1`، ولأن `string1` لم يخرج من النطاق بعد، فسيبقى مرجع `string1` صالحًا حتى تستخدمه تعليمة `println!`، إلا أن المصرف لا ينظر إلى المرجع بكونه صالحًا في هذه الحالة إذ أننا أخبرنا رست أن دورة حياة المرجع المُعاد بواسطة الدالة `longest` هو بطول أصغر دورة حياة مرجع مُمرّر لها، وبالتالي لا يسمح مدقق الاستعارة للشيفرة 23 بامتلاك الفرصة للحصول على مرجع غير صالح.

جرّب كتابة المزيد من الأمثلة لتجربة الحالات والقيم ودورات حياة المراجع المختلفة المُمرّر إلى الدالة `longest` ولاحظ كيفية استخدام المرجع المُعاد، وتنبّأ فيما إذا كانت تجربتك ستُصرَّف ويوافق عليها مدقق الاستعارة أم لا قبل أن تحاول تصريفها، ومن ثم جرّب تصريفها لترى إن كنت مصيبًا أم لا.

10.4.6 التفكير في سياق دورات الحياة

تعتمد الطريقة التي تحدد فيها دورة حياة المعاملات على الغرض من الدالة، فعلى سبيل المثال إذا عدّلت من كتابة الدالة `longest` لتُعيد دائمًا المعامل الأول بدلًا من المعامل الذي يمثل أطول شريحة نصية فلن نحتاج عندئذ لتحديد دورة حياة المعامل `y`. تُصرَّف الشيفرة البرمجية التالية بنجاح:

اسم الملف: src/main.rs

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

حددنا معامل دورة حياة مُمثّل بالاسم `'a` للمعامل `x` والقيمة المُعادة، إلا أننا لم نحدد دورة حياة للمعامل `y` لأن ليس لدورة حياة `y` أي علاقة بدورة حياة `x` أو القيمة المُعادة.

يجب أن يطابق معامل دورة حياة القيمة المُعادة دورة حياة أحد من المعاملات عند إعادة مرجع من دالة ما، وإذا لم يشير المرجع المُعاد إلى واحد من المعاملات فيجب أن يشير إلى قيمة أُنشئت داخل الدالة ذاتها، إلا أن هذا المرجع سيكون مرجعًا معلَّقًا، لأن القيمة ستخرج من النطاق في نهاية الدالة. ألقِ نظرةً على المحاولة التالية لتطبيق الدالة `longest` التي لن تُصرَّف بنجاح:

اسم الملف: src/main.rs

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

على الرغم من أننا حددنا معامل دورة الحياة `'a` للنوع المُعاد إلا أن الشيفرة البرمجية لن تُصرَّف لأن دورة حياة القيمة المُعادة غير مرتبطة بدورة حياة المعاملات إطلاقًا. إليك رسالة الخطأ التي سنحصل عليها:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0515]: cannot return reference to local variable `result`
  --> src/main.rs:11:5
   |
|     result.as_str()
   |     ^^^^^^^^^^^^^^^ returns a reference to data owned by the
current function
```

```
For more information about this error, try `rustc --explain E0515`.
error: could not compile `chapter10` due to previous error
```

تكمن المشكلة هنا في أن `result` يخرج من النطاق ويُحرَّر من الذاكرة بنهاية الدالة `longest`، إلا أننا نحاول أيضًا إعادة مرجع للقيمة `result` من الدالة في ذات الوقت، ولا يوجد هناك أي وسيلة لتحديد معاملات دورة الحياة بحيث نتخلص من المرجع المُعلَّق ولن تسمح لنا رست بإنشاء مرجع معلّق. الحل الأمثل في هذه الحال هو بجعل القيمة المُعادة نوع بيانات مملوك owned data type بدلًا من استخدام مرجع، بحيث تكون الدالة المُستدعاة حينها مسؤولة عن تحرير القيمة فيما بعد.

يتمثّل توصيف دورة الحياة بربط دورات حياة معاملات مختلفة والقيم المُعادة من الدوال، بحيث تحصل رست على معلومات كافية بعد الربط للسماح بعمليات آمنة على الذاكرة ومنع عمليات قد تتسبب بالحصول على مؤشرات معلّقة أو تخرق أمان الذاكرة.

10.4.7 توصيف دورة الحياة في تعاريف الهيكل

كانت الهياكل التي عرفناها لحد اللحظة تحتوي على أنواع مملوكة، إلا أنه يمكننا تعريف الهياكل بحيث تحتوي على مراجع وفي هذه الحالة علينا توصيف دورة حياة لكل من المراجع في تعريف الهيكل. تحتوي الشيفرة 24 على هيكل يدعى `ImportantExcerpt` يحتوي على شريحة سلسلة نصية.

اسم الملف: src/main.rs

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("Could not
find a '.'");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

[الشيفرة 24: هيكل يحتوي على مرجع، وبذلك يتطلب توصيف دورة الحياة]

يحتوي الهيكل على حقل يدعى `part` يخزّن داخله شريحة سلسلة نصية وهي مرجع، وينبغي علينا هنا التصريح عن اسم معامل دورة الحياة المعممة داخل أقواس مثلثة بعد اسم الهيكل كما هو الحال مع الأنواع

المعممة وذلك حتى يتسنّى لنا استخدام معامل دورة الحياة في متن تعريف الهيكل، وتعني طريقة الكتابة هذه أنه لا يوجد أي نسخة من `ImportantExcerpt` تعيش أطول من المرجع الموجود في الحقل `part`.

تُنشئ الدالة `main` هنا نسخةً من الهيكل `ImportantExcerpt` بحيث يحتوي على مرجع للجملة الأولى من `String` والمملوك من قبل المتغير `novel`، والبيانات في `novel` موجودةٌ قبل إنشاء نسخة من `ImportantExcerpt`، بالإضافة إلى ذلك فإن `novel` لا تخرج من النطاق إلى أن يخرج `ImportantExcerpt` من النطاق. إذًا، فالمرجع الموجود في نسخة `ImportantExcerpt` صالح.

10.4.8 إخفاء دورة الحياة

تعلمنا أنه لكل مرجع ما دورة حياة ويجب أن نحدّد معاملات دورة الحياة للدوال أو للهياكل التي تستخدم المراجع، إلا أننا كتبنا دالةً في السابق (الشيفرة 9) كما هي موضحة في الشيفرة 25، وقد صُرِّفت بنجاح دون استخدام توصيف دورة الحياة.

اسم الملف: src/lib.rs

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

[الشيفرة 25: دالة عرفناها سابقًا وصرّفت بنجاح دون استخدام توصيف دورة الحياة على الرغم من كون كل من المعاملات والقيمة المعادة مراجع]

السبب في تصريف الشيفرة السابقة بنجاح هو سبب تاريخي، إذ لن تُصرَّف الشيفرة البرمجية هذه في الإصدارات السابقة من رست (قبل 1.0)، وذلك لحاجة كل مرجع لدورة حياة صريحة. إليك ما ستبدو عليه بصمة الدالة في ذلك الوقت من تطوير اللغة:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

وجد فريق تطوير رست بعد كتابة الكثير من الشيفرات البرمجية باستخدام اللغة أن معظم مبرمجي رست يُدخلون توصيف دورة الحياة ذاته مرةً بعد الأخرى في حالات معيّنة، وكان يمكن توقع هذه الحالات واتباعها بأنماط للتعرف عليها، وبالتالي برمج المطوّرون هذه الأنماط إلى شيفرة المصرّف البرمجية بحيث يتعرّف عليها مدقق الاستعارة ويستنتج دورات الحياة في هذه الحالات دون الحاجة لكتابة توصيف دورة الحياة مباشرةً.

هذه النقطة في تاريخ تطوير رست مهمة لأنه من الممكن ظهور المزيد من الأنماط مستقبلًا وإضافتها إلى المصرف، وبذلك قد لا نحتاج لاستخدام توصيف دورات الحياة مباشرةً في العديد من الحالات.

تُدعى هذه الأنماط الموجودة لتحليل المراجع بقواعد إخفاء دورة الحياة lifetime elision rules، إلا أن هذه القواعد ليست للمبرمجين حتى يتبعونها بل هي مجموعة من الحالات التي سينظر إليها المصرّف، إذ لن تحتاج لاستخدام توصيف دورات الحياة مباشرةً إذا كانت شيفرتك البرمجية تندرج ضمن واحدة من هذه الحالات.

لا تقدّم قواعد الإخفاء القدرة على الاستنتاج بصورةٍ كاملة، إذ لن يستطيع المصرف تخمين دورات الحياة الخاصة بالمراجع الأخرى إذا طبقت رست هذه القواعد بصورةٍ حتمية ووُجد غموض ما بخصوص أي دورات الحياة تنتمي للمراجع، ففي هذه الحالة يعرض لك المصرف رسالة خطأ بدلًا من التخمين، ويمكنك حينها تصحيح هذا الخطأ عن طريق إضافة توصيفٍ لدورة الحياة.

تُدعى دورات الحياة لمعاملات دالة أو تابع بدورات حياة الدخل input lifetimes بينما تُدعى دورات الحياة الخاصة بالقيم المُعادة بدورات حياة الخرج output lifetimes.

يستخدم المصرف ثلاث قواعد لمعرفة دورات حياة المراجع عندما لا يوجد هناك توصيف مباشر لها: تُطبَّق القاعدة الأولى على دورات حياة الدخل والثانية والثالثة على دورات حياة الخرج. يتوقف المصرّف ويعطينا خطأ إذا تحقق من القواعد الثلاث ولم يتعرف على كل دورات حياة المراجع، وتنطبق هذه القواعد على تعاريف `fn` بالإضافة إلى كتل `impl`.

تتمثل القاعدة الأولى بإسناد المصرف معامل دورة حياة لكل معامل يشكّل مرجع، بكلمات أخرى: تحصل دالةً تحتوي على معامل واحد على معامل دورة حياة واحد `fn foo<'a>(x: &'a i32)`، بينما تحصل دالة تحتوي على معاملين على دورتَي حياة منفصلتين `foo<'a, 'b>(x: &'a i32, y: &'b i32)`، وهلمّ جرًا.

تنص القاعدة الثانية على وجود معامل دورة حياة دخل واحد فقط، وتُسند دورة الحياة هذه إلى جميع معاملات دورة حياة الخرج: `fn foo<'a>(x: &'a i32) -> &'a i32`.

أخيرًا، تنص القاعدة الثالثة على إسناد دورة الحياة الخاصة بـ `self` لجميع معاملات دورة حياة الخرج، إذا وُجدت عدّة معاملات دورة حياة دخل وكان أحدها `&self` أو `&mut self` لأنها تابع. تجعل القاعدة الثالثة من التوابع أسهل قراءةً لأنها تُغنينا عن استخدام الكثير من الرموز في تعريفها.

لنفترض أننا المصرّف. دعنا نطبّق هذه القواعد لمعرفة دورات حياة المراجع في بصمة الدالة `first_word` في الشيفرة 25. تبدأ بصمة الدالة دون أي دورات حياة مرتبطة بالمراجع:

```
fn first_word(s: &str) -> &str {
```

يطبّق المصرف القاعدة الأولى التي تقتضي بأن كل معامل سيحصل على دورة حياة خاصة بها، دعنا نسمّي دورة الحياة باسم `'a` كالمعتاد. أصبحت لدينا بصمة الدالة بالشكل التالي:

```
fn first_word<'a>(s: &'a str) -> &str {
```

نطبق القاعدة الثانية لوجود دورة حياة دخل واحدة، وتحدِّد القاعدة الثانية أن دورة حياة معامل الداخل تُسند إلى دورة حياة الخرج، فتصبح بصمة الدالة كما يلي:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

أصبح الآن لجميع المراجع الموجودة في بصمة الدالة دورة حياة، ويمكن للمصرف أن يستمرّ بتحليله دون حاجة المبرمج لتوصيف دورات الحياة في بصمة الدالة.

دعنا ننظر إلى مثال آخر، نستخدم هذه المرة الدالة `longest` التي لا تحتوي على معاملات دورة حياة عندما بدأنا بكتابتها في الشيفرة 20 سابقًا:

```
fn longest(x: &str, y: &str) -> &str {
```

نطبّق القاعدة الأولى: يحصل كل معامل على دورة حياة خاصة به. لدينا الآن في هذه الحالة معاملين بدلًا من واحد، لذا سنحصل على دورتين حياة:

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

يمكنك رؤية أن القاعدة الثانية لا تنطبق على هذه الحالة لوجود أكثر من دورة حياة دخل واحدة، كما أن القاعدة الثالثة لا تنطبق لأن `longest` دالة وليست تابع، إذًا لا يوجد في معاملاتها `self`. لم نتوصل إلى دورة حياة النوع المُعادة بعد تطبيق القواعد الثلاث، وهذا هو السبب في حصولنا على خطأ عند محاولة تصريف الشيفرة 20، إذ أن المصرف تحقق من قواعد إخفاء دورة الحياة ولكنه لم يتعرف على جميع دورات حياة المراجع في بصمة الدالة.

سننظر إلى دورات الحياة في سياق التوابع بما أن القاعدة الثالثة تنطبق فقط في بصمات التوابع، مما سيكشف لنا السبب في كون توصيف دورات الحياة ضمن التوابع غير مُستخدم معظم الأحيان.

10.4.9 توصيف دورة الحياة في تعاريف التابع

نستخدم طريقة الكتابة الخاصة بمعاملات الأنواع المعممة ذاتها عند تطبيق التوابع ضمن هياكل تحتوي على دورات حياة، إذ نصرح ونستخدم معاملات دورة الحياة بناءً على ارتباطها بحقول الهيكل أو معاملات التابع والقيم المُعادة، إذ يجب أن يُصرَّح عن أسماء دورات الحياة الخاصة بحقول الهياكل بعد الكلمة المفتاحية `impl` ومن ثم استخدامها بعد اسم الهيكل لأن دورات الحياة هذه تشكل جزءًا من نوع الهيكل.

قد ترتبط المراجع في بصمة التابع داخل الكتلة `impl` بدورات حياة المراجع الخاصة بحقول الهيكل، وقد تكون مستقلةً عن بعضها الآخر، كما أن قوانين إخفاء دورة الحياة تجعل من توصيف دورات الحياة غير ضروري في بصمات التابع معظم الأحيان. دعنا ننظر إلى بعض الأمثلة باستخدام هيكل يدعى `ImportantExcerpt` وهو هيكل عرّفناه سابقًا في الشيفرة 24.

لنستخدم أولًا تابعًا يدعى `level` يحتوي على معامل واحد يمثل مرجعًا إلى `self` ويُعيد قيمة من النوع `i32` (أي لا تمثّل مرجعًا):

```
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

التصريح عن معامل دورة الحياة بعد `impl` واستخدامه بعد اسم النوع مطلوب، إلا أنه من غير المطلوب توصيف دورة حياة مرجع `self` بفضل قاعدة إخفاء دورة الحياة الأولى.

إليك مثالًا ينطبق عليه قاعدة إخفاء دورة الحياة الثالثة:

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

هناك دورتا حياة دخل، لذا يطبق رست القاعدة الأولى ويمنح لكل من `&self` و `announcment` دورة حياة خاصة بهما، ومن ثم يحصل النوع المُعادة على دورة الحياة `&self` لأن إحدى معاملات التابع قيمته `&self`، وبهذا يجري التعرف على جميع دورات الحياة الموجودة.

10.4.10 دورة الحياة الساكنة

يجب أن نناقش واحدةً من دورات الحياة المميزة ألا وهي `'static` وهي تُشير إلى أن المرجع يمكن أن يعيش طوال فترة البرنامج، ولدى جميع السلاسل النصية نوع دورة الحياة الساكنة `'static`، ويمكننا توصيفه بالشكل التالي:

```
let s: &'static str = "I have a static lifetime.";
```

يُخزّن النص الموجود في السلسلة النصية في ملف البرنامج التنفيذي مباشرةً أي أنه مرئي طوال الوقت، بالتالي فإن دورة حياة جميع السلاسل النصية المجردة literals هي `'static`.

قد تجد اقتراحات لاستخدام دورة الحياة `static'` في رسائل الخطأ، إلا أنه يجب عليك أن تفكر فيما إذا كان المرجع بذاته يعيش طيلة دورة حياة البرنامج أم لا إذا أردت اتباع هذا الاقتراح وفيما إذا كنت تريد هذا الشيء حقًا أم لا، وتنتج رسالة الخطأ التي تقترح دورة حياة `'static` معظم الأحيان من محاولة إنشاء مرجع معلّق أو حالة عدم تطابق ما بين دورات الحياة الموجودة، وفي هذه الحالة فالحل الأمثل هو بحل هذه المشاكل وليس بتحديد دورة الحياة الساكنة `'static`.

10.4.11 معاملات الأنواع المعممة وحدود السمة ودورات الحياة معا

دعنا ننظر إلى طريقة تحديد معاملات الأنواع المعممة وحدود السمة ودورات الحياة في دالة واحدة سويًا.

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

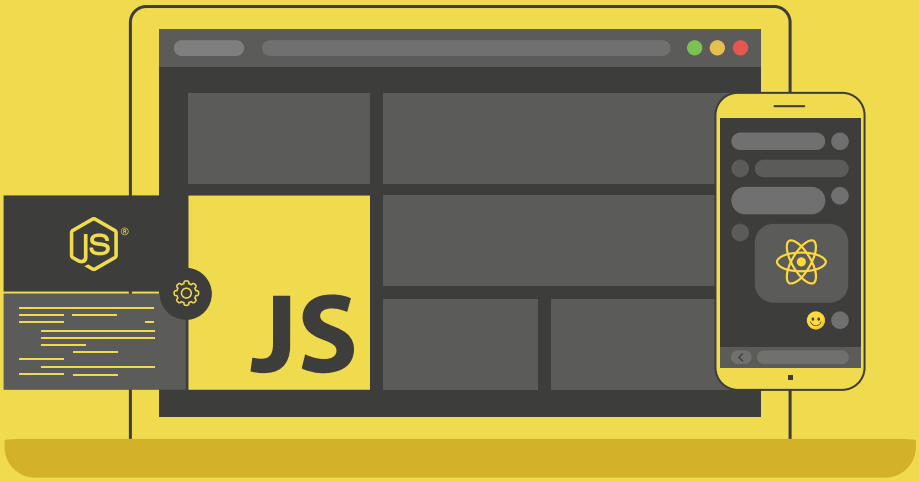
تمثل الشيفرة البرمجية السابقة دالة `longest` من الشيفرة 21 سابقًا التي تُعيد أطول شريحة نصية من شريحتين نصيتين، إلا أننا أضفنا هنا معاملًا جديدًا يدعى `ann` من نوع معمّم `T` الذي يُمكن أن يُملأ بأي نوع يطبّق السمة `Display` كما هو محدد في بنية `where`، وسيُطبع هذا المعامل الإضافي باستخدام `{}` وهذا هو السبب في جعل حدود السمة `Display` ضرورية. نكتب كل من تصاريح معاملات دورة الحياة `'a` ومعامل النوع المعمم `T` في القائمة ذاتها داخل الأقواس المثلثة بعد اسم الدالة وذلك لأن دورات الحياة هي نوع من الأنواع المعمّمة.

10.5 خاتمة

غطّينا الكثير من المعلومات في هذا الفصل، إذ يجب أن تعلم بحلول هذه النقطة عن معاملات النوع المعمّم والسمات وحدود السمة ومعاملات دورة الحياة المعممة، مما يعني أنك مستعد لكتابة شيفرة برمجية دون تكرار في العديد من الحالات. تسمح لك معاملات النوع المعمم بتنفيذ الشيفرة البرمجية على أنواع مختلفة، بينما تتأكد السمات وحدود السمة من أن سلوك النوع سيكون مطابقًا لما تتوقعه الشيفرة البرمجية حتى لو كان نوعًا معمّمًا. تعلّمنا بعدئذ كيفية استخدام توصيف دورة الحياة للتأكد من أن هذه الشيفرة البرمجية المرنة لن تتسبب بأي مراجع معلّقة، وجميع خطوات التأكد من هذا تحصل عند وقت التصريف مما يعني أن الأداء لن يتأثر سلبيًا بها.

صدّق أو لا تصدق، هناك الكثير من الأشياء لتعلمها حول المواضيع التي ناقشناها في هذا الفصل، إذ سنناقش لاحقًا في الفصل 17 كائنات السمة trait objects وهي طريقة أخرى لاستخدام السمات، كما أن هناك المزيد من الحالات المعقدة التي تحتوي على استخدام لتوصيف دورة الحياة بصورةٍ متقدّمة، فإذا أردت النظر لهذه الحالات اقرأ مرجع رست. سنتعلّم تاليًا كيفية كتابة الاختبارات في رست بحيث تتأكد أن شيفرتك البرمجية ستعمل بالطريقة التي يجب أن تعمل بها.

دورة تطوير التطبيقات
باستخدام لغة JavaScript



احترف تطوير التطبيقات بلغة جافا سكريبت
انطلاقًا من أبسط المفاهيم وحتى بناء تطبيقات حقيقية

التحق بالدورة الآن

11. كتابة الاختبارات الآلية

"قد يكون اختبار البرامج أداةً فعالةً جدًا للكشف عن وجود الأخطاء، إلا أنها أداة غير كافية لتوضيح غياب الأخطاء" هذا ما قاله أيدسكر دايكسترا Edsger W. Dijkstra في مقال "المبرمج المتواضع The Humble Programmer" عام 1972، إلا أن ذلك لا يعني أنه لا يجب علينا تعلّم كيفية إجراء الاختبارات.

تعتمد صحة برامجنا على تنفيذ شيفرتنا البرمجية لما أردنا فعله تحديدًا، وقد صُمّمَت رست بحرص شديد بخصوص صحة البرامج إلا أن كشف صحة البرنامج عمليةً معقدة وغير سهلة البرهان، ويحمل نظام أنواع رست جزءًا كبيرًا من هذه المسؤولية إلا أن نظام الأنواع لا يستطيع الكشف عن كل الأخطاء، ولذلك تدعم رست كتابة اختبارات برمجية مؤتمتة automated software tests.

لنقل أننا كتبنا دالةً تدعى `add_two` هدفها إضافة القيمة 2 إلى أي رقم نمرّره إليها، بالتالي يجب على بصمة الدالة أن تقبل عددًا صحيحًا بمثابة معامل وأن تعيد عددًا صحيحًا أيضًا في النتيجة. عندما نطبّق هذه الدالة ونصرّفها، تتحقق رست من الأنواع المستخدمة بالإضافة لمدقق الاستعارة borrow checking الذي تعلمناه سابقًا، وذلك للتأكد من أننا لا نمرّر قيمةً من النوع `String` أو مرجعًا غير صالح لهذه الدالة على سبيل المثال، إلا أنه لا يمكن لرست التحقق من أن هذه الدالة ستُنجز تحديدًا ما نريد منها أن تُنجز ألا وهو إعادة قيمة المعامل زائد 2 بدلًا من زائد 10 أو ناقص 50 على سبيل المثال، وهنا يأتي دور الاختبارات.

يمكننا كتابة اختبارات تتأكد من أن القيمة المعادة من الدالة `add_two` هي 5 عندما نمرّر 3 لها على سبيل المثال، ويمكننا تنفيذ هذه الاختبارات بعد كل تعديل نُجريه على الشيفرة البرمجية للتأكد من أن السلوك الصحيح للدالة لم يتأثر بالتعديلات التي أجريناها.

اختبار الشيفرة البرمجية مهارةٌ معقّدة؛ فعلى الرغم من تغطيتنا لجميع جوانب كتابة اختبار جيّد في فصل واحد إلا أننا سنناقش أيضًا طريقة رست في التعامل مع الاختبارات كما أننا سنتحدث عن التوصيفات annotations والماكرو macros المتاحة لك عند كتابة الاختبارات والسلوك الافتراضي والخيارات الموجودة

لتنفيذ الاختبارات، إضافةً إلى كيفية تنظيم الاختبارات في وحدة اختبار unit test واختبارات تكامل integration tests.

11.1 كتابة الاختبارات

الاختبارات Tests هي مجموعة من دوال رست تتأكد من أن الشيفرة البرمجية الأساسية تعمل كما هو مطلوب منها، ويؤدي متن دوال الاختبار عادةً هذه العمليات الثلاث:

1. تجهيز أي بيانات أو حالة ضرورية.
2. تنفيذ الشيفرة البرمجية التي تريد اختبارها.
3. التأكد من أن النتائج وفق المتوقع.

لننظر إلى المزايا التي توفرها رست لكتابة الاختبارات التي تؤدي العمليات الثلاث السابقة، ويتضمن ذلك السمة `test` وبعض الماكرو والسمة `should_panic`.

11.1.1 بنية دالة الاختبار

تُوصَف دالة الاختبار في رست باستخدام السمة `test`؛ والسمات attributes هي بيانات وصفية metadata تصف أجزاءً من شيفرة رست البرمجية، ومثال على هذه السمات هي السمة `derive` التي استخدمناها مع الهياكل سابقًا. لتغيير دالة عادية إلى دالة اختبار نُضيف `#[test]` قبل السطر الذي نكتب فيه `fn`، إذ تبني رست ملف تنفيذ اختبار ثنائي عند تنفيذ الاختبارات باستخدام الأمر `cargo test`، وتختبر الدوال المُشار إليها بأنها دوال اختبار وتعرض لك تقريرًا يوضح أي الدوال التي فشلت وأيها التي نجحت.

تُولَّد وحدة اختبار test module مع دالة اختبار تلقائيًا عندما نُنشئ مشروع مكتبة جديدة باستخدام كارجو Cargo، وتمنحنا هذه الوحدة قالبًا لكتابة الاختبارات التي نريد، بحيث لا يتوجب عليك النظر إلى هيكل الاختبار وطريقة كتابته كل مرة تُنشئ فيها مشروعًا جديدًا، ويمكنك إضافة دوال اختبار ووحدات اختبار إضافية قدر ما تشاء.

سننظر سويًا إلى بعض جوانب عمل الاختبارات بتجربة قالب الاختبار قبل اختبار أي شيفرة برمجية فعليًا، ثم سنكتب اختبارات واقعية تستدعي شيفرةً برمجيةً كتبناها لاحقًا وتتأكد من صحة سلوكها.

ننشئ مشروع مكتبة جديدة نسميه `adder` يضيف رقمين إلى بعضهما:

```
$ cargo new adder --lib
     Created library `adder` project
$ cd adder
```

يجب أن تبدو محتويات الملف src/lib.rs في مكتبة `adder` كما هو موضح في الشيفرة 1.

اسم الملف: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

[الشيفرة 1: وحدة الاختبار والدالة المولّدة تلقائيًا باستخدام `cargo new`]

لنتجاهل أول سطرين ونركّز على الدالةحاليًا. لاحظ التوصيف `#[test]`: تُشير هذه السمة إلى أن هـذه دالـة اختبار، ما يعني أن منفّذ الاختبار سيعامل هذه الدالة على أنها اختبار، وقد تحتـوي شـيفرتنا البرمجيـة أيضًـا على دوال ليست بدوال اختبار في وحدة `tests` وذلـك بهـدف مسـاعدتنا لضـبط حـالات معينـة، أو إجـراء عمليـات شائعة، لذا نحدّد دومًا فيما إذا كانت الدالة دالة اختبار.

يُستخدم متن الدالة في المثال الماكرو `assert_eq!` للتأكد من أن `result` تحتـوي على القيمـة 4 (وهي تحتوي على نتيجة جمع الرقم 2 مع 2). تمثّل عملية التأكد هذه عملية اختبارًا تقليديًا، دعنا ننفذ الاختبار لـنرى إذا ما كان سينجح أم لا.

ينفِّذ الأمر `cargo test` جميع الاختبارات الموجودة في المشروع كما هو موضح في الشيفرة 2.

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.57s
     Running unittests src/lib.rs (target/debug/deps/adder-
92948b65e88960b4)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

   Doc-tests adder
```

```
running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

[الشيفرة 2: الخرج الناتج عن عملية تنفيذ الاختبارات المولّدة تلقائيًا]

يُصرّف كارجو الاختبار ويشغّله؛ إذ نجد في أحد السطور `running 1 test`، ثم سطر يليه يوضح اسم دالة الاختبار المولّدة التي تدعى `it_works` وأن نتيجة ذلك الاختبار هي `ok`، وتعني النتيجة النهائية `test result: ok.` أن جميع الاختبارات نجحت، بينما يشير الجزء `1 passed; 0 failed` إلى عدد الاختبارات الناجحة وعدد الاختبارات الفاشلة.

من الممكن تجاهل الاختبار بحيث لا يُنفّذ في حالات معينة وسنتكلم عن هذا الأمر لاحقًا، إلا أن ملخص نتيجة الاختبارات يوضح `0 ignored` لأننا لم نفعل ذلك هنا، كما يمكننا تمرير وسيط إلى الأمر `cargo test` بحيث ينفذ الاختبارات التي يوافق اسمها السلسلة النصية ويدعى هذا بالترشيح filtering وسنتكلم عن هذا الموضوع لاحقًا. تظهر نهاية الملخص `0 filtered out` لأننا لم نستخدم الترشيح على الاختبارات التي ستُنفّذ.

يشير `0 measured` إلى الاختبارات المعيارية التي تقيس الأداء، إذ أن الاختبارات المعيارية benchmark tests متاحةٌ فقط في رست الليلية nightly Rust -في وقت كتابة هذه الكلمات- ويمكنك النظر إلى التوثيق المتعلق بالاختبارات المعيارية لتعلُّم المزيد.

يبدأ الجزء التالي من خرج الاختبار بالجملة `Doc-tests adder` وهو نتيجة لأي من اختبارات التوثيق، إلا أنه لا توجد لدينا أي اختبارات توثيق حاليًا، لكن يمكن لرست تصريف أي مثال شيفرة برمجية موجودة في توثيق واجهتنا البرمجية API، وتساعدنا هذه الميزة بالمحافظة على التوثيق وشيفرتنا البرمجية على نحوٍ متوافق. سنناقش كيفية كتابة اختبارات التوثيق لاحقًا، وسنتجاهل قسم الخرج `Doc-tests` حاليًا.

لنبدأ بتخصيص الاختبار ليوافق حاجتنا؛ إذ سنغيّر أولًا اسم الدالة `it_works` إلى اسم مختلف مثل `exploration` كما يلي:

اسم الملف: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }
```

```
}
```

نشغِّل `cargo test` مجددًا. يعرض لنا الخرج الآن `exploration` بدلًا من `it_works`:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.59s
     Running unittests src/lib.rs (target/debug/deps/adder-
92948b65e88960b4)

running 1 test
test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

نضيف الآن اختبارًا جديدًا، إلا أننا سنجعل هذا الاختبار يفشل عمدًا، إذ تفشل الاختبارات عندما يهلع panic شيءٌ ما داخل دالة الاختبار. يُجرى كل اختبار ضمن خيط thread جديد وعندما يرى الخيط الرئيس أن خيط الاختبار قد "انتهى" يُعَلَّم الاختبار بأنه فشل.

تكلمنا سابقًافي الفصل 9 عن طرق أبسط لهلع الشيفرة البرمجية باستخدام الماكرو `panic!`. الآن، نُدخل الاختبار الجديد مثل دالة تسمى `another` إلى الملف src/lib.rs كما هو موضح في الشيفرة 3.

اسم الملف: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }
```

```
    #[test]
    fn another() {
        panic!("Make this test fail");
    }
}
```

[الشيفرة 3: إضافة اختبار جديد يفشل لأننا نستدعي الماكرو `panic!`]

شغِّل الاختبارات مجددًا باستخدام `cargo test`، يجب أن يكون الخرج مشابهًا لما هو موجود في الشيفرة 4، وهو يوضّح أن اختبار `exploration` نجح بينما فشل `another`.

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.72s
     Running unittests src/lib.rs (target/debug/deps/adder-
92948b65e88960b4)

running 2 tests
test tests::another ... FAILED
test tests::exploration ... ok

failures:

---- tests::another stdout ----
thread 'main' panicked at 'Make this test fail', src/lib.rs:10:9
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace



failures:
    tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0
filtered out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

[الشيفرة 4: نتائج الاختبارات، إذ نجح اختبار وفشل آخر]

يعرض السطر `test tests::another` النتيجة `FAILED` بدلًا من `ok`، ويظهر لنا قسمين جديدين بين النتائج الفردية والملخص: إذ يعرض الأول السبب لفشل كل من الاختبارات بالتفصيل، وفي هذه الحالة نحصل على التفاصيل الخاصة بفشل `another` وهي أن `'Make this test fail'` في السطر 10 ضمن الملف src/lib.rs؛ بينما يعرض القسم التالي أسماء جميع الاختبارات التي فشلت، وهي معلومة مفيدة في حال وجد لدينا العديد من الاختبارات مع العديد من التفاصيل لكل اختبار فشل. يمكن استخدام اسم الاختبار الذي فشل لتشغيل الاختبار وحده والحصول على معلومات أدق لتنقيح الأخطاء، وسنتكلم عن طرق تشغيل الاختبارات لاحقًا.

يعرض سطر الملخص في النهاية نتيجة الاختبارات كاملةً، إذ أن نتيجة الاختبار هي `FAILED` ووجِد لدينا اختبارٌ نجح وآخر فشل.

الآن بعد أن تعرفنا إلى كيفية عرض نتائج الاختبار في حالات مختلفة، ننظر إلى ماكرو مختلفة عن `panic!` مفيدة في الاختبارات.

11.1.2 التحقق من النتائج باستخدام الماكرو assert!

تُعد الماكرو `assert!` الموجودة في المكتبة القياسية مفيدةً عندما تريد التأكد من أن شرطًا ما ضمن الاختبار يُقيَّم إلى `true`، ونمرّر للماكرو `assert!` وسيطًا يمكن تقييمه لقيمة بوليانية boolean؛ فإذا كانت القيمة `true` لا يحدث شيء عندها ونجتاز الاختبار بنجاح؛ وإذا حصلنا على القيمة `false` فهذا يعني فشل الاختبار، ويستدعي الماكرو `assert!` عندها الماكرو `panic!`. يساعدنا استخدام الماكرو `assert!` في التحقق من شيفرتنا البرمجية بالطريقة التي نريدها.

استخدمنا في الأمثلة البرمجية سابقًا هيكلًا يدعى `Rectangle` وتابع `can_hold`، وتجد المثال مكررًا هنا في الشيفرة 5. دعنا نضع هذه الشيفرة البرمجية في الملف src/lib.rs ومن ثم نكتب بعض الاختبارات باستخدام الماكرو `assert!`.

اسم الملف: src/lib.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
```

```
    }
}
```

[الشيفرة 5: استخدام الهيكل `Rectangle` وتابعه `can_hold` من مثال سابق]

يمكن أن يُعيد التابع `can_hold` قيمةً بوليانية مما يعني أننا نستطيع استخدامه مع الماكرو `assert!`. نكتب اختبارًا لنتدرّب على التابع `can_hold` بإنشاء نسخة `Rectangle` في الشيفرة 6، وتحمل النسخة عرضًا بمقدار 8 وطولًا بمقدار 7 ومن ثم نتأكد من أنها تستطيع حمل hold نسخة `Rectangle` أخرى بعرض 5 وطول 1.

اسم الملف: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn larger_can_hold_smaller() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(larger.can_hold(&smaller));
    }
}
```

[الشيفرة 6: اختبار للتابع `can_hold` يتحقق فيما إذا كان المستطيل الكبير يتسع مستطيلًا أصغر]

لاحظ أننا أضفنا سطرًا جديدًا داخل وحدة `tests` ألا وهو `use super::*;`، إذ تمثّل وحدة `test` وحدةً اعتيادية تتبع قواعد الظهورالاعتيادية visibility rules التي ناقشناها سابقًا في الفصل 7، ولأن وحدة `tests` هي وحدة داخلية inner module، فنحن بحاجة لإضافة الشيفرة البرمجية التي نريد إجراء الاختبار عليها في الوحدة الخارجية outer module إلى نطاق scope الوحدة الداخلية، ونستخدم هنا `glob` بحيث يكون كل شيء نعرفه في الوحدة الخارجية متاحًا للوحدة `tests`.

سمّينا الاختبار بالاسم `larger_can_hold_smaller` وأنشأنا نسـختين من الهيكـل `Rectangle` ومن ثم استدعينا الماكرو `assert!` ومرّرنا النتيجـة باسـتدعاء `larger.can_hold(&smaller)`. يجب أن يُعيـد هذا التعبير القيمة `true` إذا اجتاز الاختبار بنجاح، دعنا نرى بأنفسنا.

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests src/lib.rs (target/debug/deps/rectangle-
6584c4561e48942e)

running 1 test
test tests::larger_can_hold_smaller ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

   Doc-tests rectangle

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

اجتــاز الاختبــار فعلًا. دعنــا نضــيف اختبــارًا آخــر بالتأكــد من أن المســتطيل الصــغير لا يتســع داخل المستطيل الكبير:

اسم الملف: src/lib.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
```

```
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn larger_can_hold_smaller() {
        // --snip--
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(larger.can_hold(&smaller));
    }

    #[test]
    fn smaller_cannot_hold_larger() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(!smaller.can_hold(&larger));
    }
}
```

لأن النتيجة الصحيحة من الدالة `can_hold` هي `false` في هذه الحالة، فنحن بحاجة لنفي النتيجة قبل أن نمررها إلى الماكرو `assert!` وبالتالي سيجتاز الاختبار إذا أعادت `can_hold` القيمة `false`:

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests src/lib.rs (target/debug/deps/rectangle-
6584c4561e48942e)

running 2 tests
test tests::larger_can_hold_smaller ... ok
test tests::smaller_cannot_hold_larger ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

   Doc-tests rectangle

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

اجتزنا اختبارين متتالين، مرحى لنا. دعنا نرى ما الذي سيحدث الآن لنتائج الاختبار إذا أضفنا خطأً برمجيًا عن عمد في شيفرتنا البرمجية؛ نفعل ذلك بالتغيير من كتابة متن التابع `can_hold` باستبدال إشارة أكبر من إلى إشارة أصغر من عند المقارنة بين عرض كل من المستطيلين:

```
impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width < other.width && self.height > other.height
    }
}
```

نحصل على التالي عند تنفيذ الاختبارات:

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
```

```
     Running unittests src/lib.rs (target/debug/deps/rectangle-
6584c4561e48942e)


running 2 tests

test tests::larger_can_hold_smaller ... FAILED

test tests::smaller_cannot_hold_larger ... ok


failures:


---- tests::larger_can_hold_smaller stdout ----

thread 'main' panicked at 'assertion failed:
larger.can_hold(&smaller)', src/lib.rs:28:9

note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace



failures:

    tests::larger_can_hold_smaller


test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0
filtered out; finished in 0.00s


error: test failed, to rerun pass '--lib'
```

تنبّهت اختباراتنا للخطأ، لأن `larger.width` هو 8 و `smaller.width` هـو 5، والمقارنـة بين العرضَـين في `can_hold` تُعيد النتيجة `false` إذ أن 8 ليست أقل من 5.

11.1.3 اختبار المساواة باستخدام الماكرو assert_eq! و assert_ne!

نستخدم المساواة كثيرًا لاختبار البرنامج وذلك بين القيمة التي تعيدها الشيفرة البرمجية عند التنفيذ والقيمة التي تتوقع من الشيفرة البرمجية أن تعيدها، ويمكننا تحقيـق ذلـك باسـتخدام المـاكرو `assert!` وتمريـر تعبـير باستخدام العامـل `==`، إلا أن هنـاك طريقـة أخـرى أكـثر شـيوعًا موجـودة في المكتبـة القياسـية ألا وهي المـاكرو `assert_eq!` و `assert_ne!`؛ إذ يقارن كلًا من الماكرو المذكورَين سابقًا القيمة للتحقق من المساواة أو عـدم المساواة، كما أننا سنحصل على طباعة للقيمتين إذا فشـل الاختبـار، بينمـا يـدلنا المـاكرو `assert!` فقـط على حصــوله على القيمــة `false` من التعبــير `==` دون طباعــة القيم الــتي أدت لحصــولنا للقيمــة `false` في المقام الأول.

نكتب في الشـيفرة 11 دالـة تـدعى `add_two` تُضـيف القيمـة "2" إلى معاملهـا، ثم نفحص هـذه الدالـة باستخدام الماكرو `assert_eq!`.

اسم الملف: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_adds_two() {
        assert_eq!(4, add_two(2));
    }
}
```

[الشيفرة 7: اختبار الدالة `add_two` باستخدام الماكرو `assert_eq!`]

لنتحقّق من عمل الدالة:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests src/lib.rs (target/debug/deps/adder-
92948b65e88960b4)

running 1 test
test tests::it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

   Doc-tests adder

running 0 tests
```

```
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

نمـرّر القيمـة "4" على أنهـا وسـيط إلى المـاكرو `assert_eq!` وهي قيمـة تسـاوي قيمـة اسـتدعاء `add_two(2)`. السطر الخاص بالاختبار هـو: `test tests::it_adds_two ... ok` وتُشـير `ok` إلى أن نجاح الاختبار.

دعنا نضيف خطأً برمجيًا متعمّدًا على شيفرتنا البرمجية لنرى كيف ستكون رسالة فشـل الاختبـار باسـتخدام الماكرو `assert_eq!`. لنغيّر من تطبيق `add_two` لنضيف قيمة "3" بدلًا من "2":

```
pub fn add_two(a: i32) -> i32 {
    a + 3
}
```

نشغِّل الاختبارات مجددًا:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests src/lib.rs (target/debug/deps/adder-
92948b65e88960b4)

running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `4`,
 right: `5`', src/lib.rs:11:9
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace



failures:
    tests::it_adds_two
```

```
test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0
filtered out; finished in 0.00s


error: test failed, to rerun pass '--lib'
```

نجح الاختبار بـالتعرف على الخطـأ، إذ أن الاختبـار `it_adds_two` فشـل وتخبرنـا رسـالة الخطـأ أن سـبب الفشل هو `assertion failed: (left == right)` بالإضافة لتحديد قيمة كل من `left` و `right`، وتساعدنا رسالة الخطأ هذه بالبدء بعملية تنقيح الأخطاء، إذ أن قيمة `left` هي "4" إلا أن `right` -القيمـة الـتي حصلنا عليها من `add_two(2)`- هي "5". يصبح هذا الأمر مفيدًا جدًا عندما يكون لدينا الكثير من الاختبارات.

يُطلـق على معامـل التأكـد من المسـاواة للتوابـع في بعض لغـات البرمجة وأطـر العمـل اسـم `expected` و `actual` ويكـون تـرتيب تحديـد المعـاملات مهمًّا، إلا أنهمـا يحملان اسـم `left` و `right` في رسـت ولا يهمّ ترتيبهما في عند تمريرهما للماكرو، إذ يمكننا كتابة التأكيد assertion في الاختبار بالشكل:

```
assert_eq!(add_two(2), 4)
```

الذي سينتج رسالة الخطأ ذاتها التي حصلنا عليها سابقًا، وهي:

```
assertion failed: (left == right)
```

ينجح الاختبار باستخدام الماكرو `assert_ne!` إذا كانت القيمتين المُرّرتين له غير متساويتين، ويفشل في حال المساواة. نستفيد من هذا الماكرو عندما لا نعلم ماذا ستكون القيمة الناتجة تحديدًا إلا أننا نعلم أنهـا لا يجب أن تكون مساويةً لقيمة معينة؛ على سبيل المثال إذا كنا نختبر دالة يتغير دخلها بحسب يوم الأسبوع الـذي ننفّـذ فيه الاختبار، سيكون الأفضل في هذه الحالة هو التأكد من أن القيمة التي حصلنا عليها من خرج الدالة لا تساوي قيمة الدخل.

يُستخدم كـل من المـاكرو `assert_eq!` و `assert_ne!` كلًا من العامـل `==` و `!=` على الـترتيب، وعنـدما تفشل عملية التأكيد يطبع الماكرو معاملاته باستخدام تنسيق تنقيح الأخطـاء، مـا يعـني أن القيم الـتي سـتُقارن فيما بينها يجب أن تُطبّق السمتين `PartialEq` و `Debug`، وتطبق جميع الأنواع البدائية ومعظم أنواع المكتبـة القياسية هاتين السمتين. ستحتاج لتطبيق `PartialEq` للهياكـل والمعـددات enums الـتي تعرّفهـا بنفسـك للتأكد من مسـاواة النـوعين، كمـا سـتحتاج أيضًـا لتطـبيق `Debug` لطباعـة القيم عنـدما يفشـل التأكيـد. كلا من السـمتين المـذكورتين هي سـمات قابلـة للاشـتقاق derivable كمـا ذكرنـا سـابقًا في الشـيفرة 12 من الفصل 5 وهـذا يعـني أننـا نسـتطيع إضـافة `#[derive(PartialEq, Debug)]` مباشـرةً في تعريـف الهيكل أو المعدّد. أنظر الملحق (ت) للمزيد من التفاصيل حول السمات القابلة للاشتقاق.

11.1.4 إضافة رسائل فشل مخصصة

يمكنك إضافة رسائل مخصّصة لطباعتها مع رسالة الخطأ مثل معامل اختياري للماكرو `assert!` و `assert_eq!` و `assert_ne!`. تمرّر أي معاملات تُحدد بعد المعاملات المطلوبة للماكرو بدورها إلى الماكرو `format!` (الذي ناقشناه سابقًا في الفصل 8)، بحيث يمكنك تمرير سلسلة نصية منسّقة تحتوي على `{}` بمثابة موضع مؤقت والقيم بداخلها. الرسائل المخصصة مفيدة لتوثيق معنى التأكيد؛ إذ سيكون لديك فهم واضح عن المشكلة في الشيفرة البرمجية عندما يفشل الاختبار.

على سبيل المثال، دعنا نفترض وجود دالة تحيّي الناس بالاسم ونريد أن نختبر إذا كان الاسم الذي نمرّره إلى الدالة يظهر في الخرج:

اسم الملف: src/lib.rs

```
pub fn greeting(name: &str) -> String {
    format!("Hello {}!", name)
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(result.contains("Carol"));
    }
}
```

لم يُتّفق على متطلبات هذا البرنامج بعد، ونحن متأكدون أن النص "Hello" في بداية نص التحية سيتغير لاحقًا، لذا قررنا أننا لا نريد أن نعدل النص عند تغيير المتطلبات، وتحققنا بدلًا من ذلك من المساواة بين القيمة المُعادة من الدالة `greeting` للتأكد من أن الخرج يحتوي على النص الموجود في معامل الدخل.

لنضيف خطأ برمجي متعمد على الشيفرة البرمجية بتغيير `greeting` بحيث لا تحتوي على الاسم لنرى ما ستكون نتيجة فشل الاختبار الافتراضي:

```
pub fn greeting(name: &str) -> String {
    String::from("Hello!")
}
```

تكون نتيجة تشغيل الاختبار على النحو التالي:

```
$ cargo test
   Compiling greeter v0.1.0 (file:///projects/greeter)
    Finished test [unoptimized + debuginfo] target(s) in 0.91s
     Running unittests src/lib.rs (target/debug/deps/greeter-
170b942eb5bf5e3a)

running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
thread 'main' panicked at 'assertion failed:
result.contains(\"Carol\")', src/lib.rs:12:9
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace



failures:
    tests::greeting_contains_name

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0
filtered out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

تشير النتيجة إلى أن التأكيد فشل في السطر الذي يحتويه، إلا أن رسالةً أكثر إفادة ستحتوي على القيمة الموجودة في دالة `greeting`. نضيف رسالة فشل مخصصة مؤلفة من سلسلة نصية منسّقة تحتوي على موضع مؤقت يُملأ بالقيمة الفعلية التي نحصل عليها من الدالة `greeting`:

```
    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(
            result.contains("Carol"),
            "Greeting did not contain name, value was `{}`",
```

```
            result
        );
    }
```

نحصل عندما نشغّل الاختبار الآن على رسالة خطأ أكثر وضوحًا:

```
$ cargo test
   Compiling greeter v0.1.0 (file:///projects/greeter)
    Finished test [unoptimized + debuginfo] target(s) in 0.93s
     Running unittests src/lib.rs (target/debug/deps/greeter-
170b942eb5bf5e3a)

running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
thread 'main' panicked at 'Greeting did not contain name, value was
`Hello!`', src/lib.rs:12:9
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace



failures:
    tests::greeting_contains_name

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0
filtered out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

يمكننا استخدام القيمة التي حصلنا عليها من خرج الاختبار مما سيساعدنا في تنقيح الخطأ الذي تسبب بذلك بدلًا من الفعل الذي توقعناه من الدالة.

11.1.5 التحقق من حالات الهلع باستخدام should_panic

من المهم بالإضافة للتحقق من القيم المعادة التحقق من أن شيفرتنا البرمجية تتعامل مع حالات الخطأ كما نتوقع، على سبيل المثال تذكر النوع `Guess` الذي أنشأناه في أمثلة سابقة (في الفصل 9)، إذ استخدمت بعض

الشيفرات البرمجية `Guess` واعتمدت على أن نسخ `Guess` ستحتوي على قيمة تتروح بين 1 و100. يمكننا كتابة اختبار يتسبب بحالة هلع إذا حاولنا إنشاء نسخة من `Guess` خارج المجال، وذلك عن طريق إضافة سمة `should_panic` إلى دالة الاختبار، وينجح الاختبار إذا كانت الشيفرة البرمجية داخل الدالة تهلع، وإلا يفشل الاختبار في حالة عدم هلع الدالة.

توضح الشيفرة 8 اختبارًا يتأكد أن حالات الخطأ المتعلقة بالدالة `Guess::new` تحصل عندما نتوقع حصولها.

اسم الملف: src/lib.rs

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.",
value);
        }

        Guess { value }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

[الشيفرة 8: اختبار أن حالة ما ستتسبب بالهلع `panic!`]

نضع السمة `#[should_panic]` بعد السمة `#[test]` وقبل دالة الاختبار التي نريد تطبيق السمة عليها.دعنا نلقي نظرةً على النتيجة بعد نجاح الاختبار:

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished test [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests src/lib.rs (target/debug/deps/guessing_game-
57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

   Doc-tests guessing_game

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

يبدو أن الأمر على ما يرام. لنضيف خطأ برمجي عن عمد إلى شيفرتنا البرمجية بإزالة الشرط الذي يتسبب بهلع الدالة `new` إذا كانت القيمة أكبر من 100:

```
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!("Guess value must be between 1 and 100, got {}.",
value);
        }

        Guess { value }
    }
}
```

يفشل الاختبار في الشيفرة 8 عندما نشغّله:

```
$ cargo test
```

```
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished test [unoptimized + debuginfo] target(s) in 0.62s
     Running unittests src/lib.rs (target/debug/deps/guessing_game-
57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----
note: test did not panic as expected

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0
filtered out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

لا نحصل على رسالة مفيدة جدًّافي هذه الحالة، لكن إذا نظرنا إلى دالة الاختبار نرى توصيفها بالتالي: `#[should_panic]`. حالة الفشل التي حصلنا عليها تعني أن الشيفرة البرمجية في دالة الاختبار لم تتسبب بحالة هلع.

يمكن أن تصبح الاختبارات التي تستخدم `should_panic` غير دقيقة، إذ ينجح اختبار `should_pass` في حال هلع الاختبار لسبب مختلف عمّا كنا متوقعين، ولجعل اختبارات `should_panic` أكثر دقة، يمكننا إضافة معامل اختياري يدعى `expected` للسمة `should_panic`،وسيتأكد عندها الاختبار من أن رسالة الخطأ تحتوي على النص الذي زوّدناه به. على سبيل المثال، لنفترض أن الشيفرة البرمجية الخاصة بالدالة `Guess` في الشيفرة 9 احتوت على هلع الدالة `new` برسائل مختلفة بحسب إذا ما كانت القيمة صغيرة أو كبيرة.

اسم الملف: src/lib.rs

```
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!(
```

```
                "Guess value must be greater than or equal to 1, got
{}.",
                value
            );
        } else if value > 100 {
            panic!(
                "Guess value must be less than or equal to 100, got
{}.",
                value
            );
        }

        Guess { value }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic(expected = "less than or equal to 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

[الشيفرة 9: اختبار `panic!` برسالة هلع تحتوي على سلسلة نصية جزئية محددة]

سينجح الاختبار لأن القيمة التي نضعها في معامل `expected` الخاص بالسمة `should_panic` هي سلسلة نصية جزئية للرسالة التي تعرضها الدالة `Guess::new` عند الهلع. يمكننا تحديد كامل رسالة الهلع التي نتوقعها، وستكون في هذه الحالة:

```
Guess value must be less than or equal to 100, got 200.
```

يعتمد ما تختار أن تحدده على رسالة الهلع إذا ما كانت مميزة أو ديناميكية ودقة الاختبار التي تريدها، وتُعد سلسلةً نصيةً جزئيةً لرسالة الهلع كافية في هذه الحالة للتأكد من أن الشيفرة البرمجية في دالة الاختبار تنفّذ حالة `else if value > 100`.

دعنا نضيف خطأ برمجي جديد مجددًا لرؤية ما الذي سيحصل للاختبار `should_panic` باستخدام رسالة `expected` وذلك بتبديل محتوى الكتلة `if value < 1` مع `else if value > 100`:

```
        if value < 1 {
            panic!(
                "Guess value must be less than or equal to 100, got {}.",
                value
            );
        } else if value > 100 {
            panic!(
                "Guess value must be greater than or equal to 1, got {}.",
                value
            );
        }
```

يفشل الاختبار `should_panic` هذه المرة عند تشغيله:

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests src/lib.rs (target/debug/deps/guessing_game-57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----
thread 'main' panicked at 'Guess value must be greater than or equal to 1, got 200.', src/lib.rs:13:13
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
note: panic did not contain expected string
      panic message: `"Guess value must be greater than or equal to 1, got 200."`,
 expected substring: `"less than or equal to 100"`
```

```
failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0
filtered out; finished in 0.00s

error: test failed, to rerun pass '--lib'
```

تشير رسالة الخطأ إلى أن هذا الاختبار هلع فعلًا كما توقعنا إلا أن رسالة الهلع لم تتوافق مع السلسلة النصية الجزئية:

```
'Guess value must be less than or equal to 100'
```

إذ حصلنا على رسالة الهلع التالية في هذه الحالة:

```
Guess value must be greater than or equal to 1, got 200.
```

لنبدأ الآن بالبحث عن الخطأ.

11.1.6 استخدام Result<T, E> في الاختبارات

تهلع جميع اختباراتنا لحد هذه اللحظة عند الفشل، إلا أنه يمكننا كتابة اختبارات تستخدم `Result<T, E>`. إليك اختبارًا من الشيفرة 1 إذ أعدنا كتابته باستخدام `Result<T, E>` ليعيد `Err` بدلًا من الهلع:

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() -> Result<(), String> {
        if 2 + 2 == 4 {
            Ok(())
        } else {
            Err(String::from("two plus two does not equal four"))
        }
    }
}
```

للدالة `it_works` الآن النوع المُعاد `Result<(), String>`، ونُعيد `Ok(())` عندما ينجح الاختبار و `Err` مع `Sring` بداخله عندما يفشل وذلك بدلًا من استدعاء الماكرو `assert_eq!`.

تمكّنك كتابة الاختبارات بحيث تعيد `Result<T, E>` من استخدام عامل إشارة الاستفهام في محتوى الاختبار بحيث تكون وسيلةً ملائمةً لكتابة الاختبارات التي يجب أن تفشل إذا أعادت أي عملية داخلها المتغاير `Err`.

لا يمكنك استخدام التوصيف `#[should_panic]` على الاختبارات التي تستخدم `Result<T, E>`. لا تستخدم عامل إشارة الاستفهام على `Result<T, E>` للتأكد من أن عملية ما تُعيد المتغاير `Err` بل استخدم `assert!(value.is_err())` بدلًا من ذلك.

الآن وبعد أن تعلمنا الطرق المختلفة لكتابة الاختبارات، حان الوقت لننظر إلى ما يحدث عندما ننفذ الاختبارات ونكتشف الخيارات المختلفة التي يمكننا استخدامها مع `cargo test` في القسم التالي.

11.2 التحكم بتنفيذ الاختبارات

يصرّف `cargo test` شيفرتك البرمجية بالطريقة نفسها التي يصرّف فيها الأمر `cargo run` شيفرتك البرمجية ويشغّلها، إلا أن `cargo test` يصرّف الشيفرة البرمجية في نمط الاختبار ويشغّل ملف الاختبار الثنائي. السلوك الافتراضي للملف الثنائي الناتج عن `cargo test` هو تشغيل جميع الاختبارات على التوازي والحصول على الخرج خلال تشغيل الاختبار ومنع الخرج من العرض مما يجعل من الخرج المتعلق بنتائج الاختبار أكثر وضوحًا، إلا أنه يمكنك كتابة خيارات في سطر الأوامر للتغيير من هذا السلوك الافتراضي.

تنتمي بعض الخيارات في سطر الأوامر إلى `cargo test` بينما ينتمي بعضها لملف الاختبار الثنائي الناتج، وللفصل بين النوعين من الوسطاء نضع الوسطاء الخاصة بالأمر `cargo test` متبوعةً بالفاصل `--` ومن ثم الوسطاء الخاصة بملف الاختبار الثنائي. يعرض تنفيذ الأمر `cargo test --help` الخيارات الممكن استخدامها مع `cargo test`، بينما يعرض تنفيذ `cargo test -- --help` الخيارات التي يمكنك استخدامها بعد الفاصل.

11.2.1 تشغيل الاختبارات على نحو متعاقب أو على التوازي

تُنفّذ الاختبارات على التوازي افتراضيًا عند تشغيل عدة اختبارات باستخدام خيوط threads، مما يعني أن تنفيذها سيكون سريعًا وستحصل على نتيجتك بصورةٍ أسرع، وبما أن الاختبارات تُنفّذ في الوقت ذاته فهذا يعني أنها يجب ألا تعتمد على بعضها بعضًا أو تحتوي على حالة مشتركة بما فيه بيئة مشتركة مثل مسار العمل الحالي current working directory أو متغيرات البيئة environment variables.

على سبيل المثال لنقل أن اختباراتك تنفذ شيفرة برمجية تُنشئ ملفًا على قرص باسم test-output.txt وتكتب بعض البيانات على هذا الملف، يقرأ عندها كل اختبار البيانات في ذلك الملف ويتأكد أن الملف يحتوي على قيمة معينة وهي قيمة مختلفة بحسب كل اختبار، ولأن الاختبارات تُشغّل في الوقت ذاته فهذا يعني أن أحد الاختبارات قد يكتب على محتويات الملف في وقت تنفيذ اختبار آخر وقراءته للملف وسيفشل عندها

الاختبار الثاني ليس لعدم صحة الشيفرة البرمجية بل لأن الاختبارات أثرت على بعضها بعضًا عند تشغيلها على التوازي. يكمن أحد الحلول هنا بالتأكد أن كل اختبار يكتب إلى ملف مختلف، وهناك حلّ آخر يتمثل بتشغيل الاختبارات على التتالي كلّ حسب دوره.

إذا لم ترد تشغيل الاختبارات على التوازي، أو أردت تحكمًا أكبر على أرقام الخيوط التي تريد استخدامها فيمكنك عندئذ استخدام الراية `--test-threads` متبوعةً بعدد الخيوط التي تريد استخدامها مع الاختبار الثنائي. ألقِ نظرةً على المثال التالي:

```
$ cargo test -- --test-threads=1
```

نضبط عدد خيوط الاختبار إلى "1"، وهذا يجعل البرنامج يعرف أننا لا نريد تشغيل الاختبارات على التوازي، إذ يستغرق تشغيل الاختبارات باستخدام خيط واحد وقتًا أكبر من تشغيلها على التوازي، إلا أن الاختبارات لن تتداخل في عمل بعضها بعضًا إذا تشاركت في حالة ما.

11.2.2 عرض خرج الدالة

تلتقط مكتبة اختبار رست تلقائيًا كل شيء يُطبع إلى الخرج القياسي إذا نجح الاختبار، على سبيل المثال إذا استدعينا `println!` في اختبار ما ونجح هذا الاختبار فلن نرى خرج `println!` في الطرفية بل سنرى فقط السطر الذي يشير إلى نجاح الاختبار، بينما سنرى ما طُبع إلى الخرج القياسي إذا فشل الاختبار مصحوبًا مع رسالة الفشل.

تحتوي الشيفرة 10 على مثال بسيط يحتوي على دالة تطبع قيمة معاملها وتُعيد 10 إضافةً إلى اختبار ينجح وآخر يفشل.

اسم الملف: src/lib.rs

```
fn prints_and_returns_10(a: i32) -> i32 {
    println!("I got the value {}", a);
    10
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn this_test_will_pass() {
        let value = prints_and_returns_10(4);
```

```
        assert_eq!(10, value);
    }

    #[test]
    fn this_test_will_fail() {
        let value = prints_and_returns_10(8);
        assert_eq!(5, value);
    }
}
```

[الشيفرة 10: اختبارات للدالة التي تستدعي `println!`]

نحصل على الخرج التالي عند تشغيل هذه الاختبارات باستخدام `cargo test`:

```
$ cargo test
   Compiling silly-function v0.1.0 (file:///projects/silly-function)
    Finished test [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests src/lib.rs (target/debug/deps/silly_function-
160869f38cff9166)

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `5`,
 right: `10`', src/lib.rs:19:9
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace



failures:
    tests::this_test_will_fail
```

```
test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0
filtered out; finished in 0.00s


error: test failed, to rerun pass '--lib'
```

لاحظ أن الخرج قد التُقط ولا يوجد فيه `I got the value 4` وهو ما نطبعه عند تشغيل الاختبار الذي سينجح، بينما يظهر الخرج `I got the value 8` من الاختبار الذي فشل في قسم خرج ملخص الاختبار الذي يوضح أيضًا سبب فشل الاختبار.

يمكننا اختبار رست بعرض خرج الاختبارات الناجحة باستخدام `--show-output` إذا أردنا رؤية القيم المطبوعة للاختبارات الناجحة أيضًا.

```
$ cargo test -- --show-output
```

نحصل على الخرج التالي عند تشغيل الاختبارات الموجودة في الشيفرة 10 مجددًا باستخدام الراية `--show-output`:

```
$ cargo test -- --show-output
   Compiling silly-function v0.1.0 (file:///projects/silly-function)
    Finished test [unoptimized + debuginfo] target(s) in 0.60s
     Running unittests src/lib.rs (target/debug/deps/silly_function-
160869f38cff9166)

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

successes:

---- tests::this_test_will_pass stdout ----
I got the value 4



successes:
    tests::this_test_will_pass

failures:
```

```
---- tests::this_test_will_fail stdout ----
I got the value 8
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `5`,
 right: `10`', src/lib.rs:19:9
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace



failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0
filtered out; finished in 0.00s


error: test failed, to rerun pass '--lib'
```

11.2.3 تشغيل مجموعة من الاختبارات باستخدام اسم

قد يستغرق تشغيل كافة الاختبارات في بعض الأحيان وقتًا طويلًا، وقد تريد تشغيل مجموعة من الاختبارات مرتبطة فقط بجزئية معينة ضمن شيفرتك البرمجية، ويمكنك اختيار الاختبارات التي تريد تنفيذها بتمرير اسم الاختبار أو أسماء الاختبارات مثل وسطاء إلى الأمر `cargo test`.

لتوضيح كيفية تشغيل مجموعة من الاختبارات نُنشئ أولًا ثلاث اختبارات للدالة `add_two` كما هو موضح في الشيفرة 11 ونختار أي الاختبارات التي نريد تشغيلها.

اسم الملف: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn add_two_and_two() {
```

```
        assert_eq!(4, add_two(2));
    }

    #[test]
    fn add_three_and_two() {
        assert_eq!(5, add_two(3));
    }

    #[test]
    fn one_hundred() {
        assert_eq!(102, add_two(100));
    }
}
```

[الشيفرة 11: ثلاثة اختبارات مع ثلاثة أسماء مختلفة]

ستُنفّذ جميع الاختبارات على التوازي إذا شغّلنا الاختبارات دون تمرير أي وسطاء كما فعلنا سابقًا:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.62s
     Running unittests src/lib.rs (target/debug/deps/adder-
92948b65e88960b4)

running 3 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

ا. تشغيل الاختبارات بصورة فردية

يمكننا تمرير اسم أي دالة اختبار للأمر `cargo test` لتنفيذ الاختبار بصورةٍ فردية:

```
$ cargo test one_hundred
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.69s
     Running unittests src/lib.rs (target/debug/deps/adder-
92948b65e88960b4)

running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered
out; finished in 0.00s
```

جرى تشغيل الاختبار بالاسم `one_hundred` فقط، إذ لم يطابق الاختباران الآخران هـذا الاسـم. يسـمح لنـا الخرج بمعرفة أن هناك المزيد من الاختبارات التي لم ننفذها بعرض `2 filtered out` في النهاية.

لا يمكننـا تحديـد أسـماء اختبـارات متعـددة بهـذه الطريقـة، إذ تُسـتخدم القيمـة الأولى المُعطـاة للأمر `cargo test` فقط، إلا أن هناك وسيلة أخرى لتنفيذ عدة اختبارات.

11.2.4 تنفيذ عدة اختبارات عن طريق الترشيح

يمكننا تحديد جزء من اسم اختبار بحيث يُنفّذ أي اختبار يطابق اسمه هذه القيمة. على سبيل المثال، يمكننا تنفيذ اختبارين من الاختبارات الثلاثة السابقة عن طريق `add` وذلك بكتابة الأمر `cargo test add`:

```
$ cargo test add
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests src/lib.rs (target/debug/deps/adder-
92948b65e88960b4)

running 2 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered
out; finished in 0.00s
```

يشغّل هذا الأمر جميع الاختبارات التي تحتوي على add في اسمها، ويستثني هنا الاختبار one_hundred، كما يجب ملاحظة أن الوحدة التي تحتوي على الاختبار بداخلها تصبح جزءًا من اسم الاختبار، أي يمكننا تشغيل جميع الاختبارات الموجودة في وحدة معينة عن طريق استخدام اسمها.

11.2.5 تجاهل بعض الاختبارات إلا في حال طلبها

قد يكون لدينا في بعض الأحيان اختبارات معينة تستغرق وقتًا طويلًا لتنفيذها وقد ترغب باستثنائها من التشغيل عند كتابة الأمر cargo test. يمكنك هنا استخدام توصيف الاختبارات التي تستغرق وقتًا طويلًا باستخدام السمة ignore لاستثنائها بدلًا من كتابة جميع الاختبارات التي تريد تشغيلها مثل وسطاء باستثناء تلك الاختبارات، كما هو موضح هنا:

اسم الملف: src/lib.rs

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}

#[test]
#[ignore]
fn expensive_test() {
    // شيفرة برمجية تستغرق عدة ساعات للتنفيذ
}
```

نضيف #[ignore] بعد سطر #[test] ضمن الاختبار الذي نريد استثناءه، والآن عند تشغيل الاختبارات يُنفّذ الاختبار it_works دون expensive_test:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.60s
     Running unittests src/lib.rs (target/debug/deps/adder-
92948b65e88960b4)

running 2 tests
test expensive_test ... ignored
test it_works ... ok
```

```
test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered
out; finished in 0.00s


   Doc-tests adder


running 0 tests


test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

تُدرج الدالة `expensive_test` تحت `ignored`، وإذا أردنا تشغيل الاختبارات التي تجاهلناها فقط نكتب `cargo test -- --ignored`:

```
$ cargo test -- --ignored
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests src/lib.rs (target/debug/deps/adder-
92948b65e88960b4)


running 1 test
test expensive_test ... ok


test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered
out; finished in 0.00s


   Doc-tests adder


running 0 tests


test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

نتأكد من خلال التحكم بالاختبارات التي تُنفَّذ من أن نتائج `cargo test` ستكون سريعة، يمكنك تنفيذ `cargo test -- --ignored` عندما تكون في حالة تريد فيها التحقق من الاختبارات التي تندرج تحت `ignored` ولديك الوقت لانتظار النتائج، بينما تستطيع تنفيذ الأمر التالي إذا أردت تشغيل جميع الاختبارات المتجاهلة وغير المتجاهلة دفعةً واحدة.

```
cargo test -- --include-ignored
```

11.3 تنظيم الاختبارات

يُعد اختبار الشيفرة البرمجية كما ذكرنا سابقًا ممارسةً معقدة، ويستخدم الناس الاختبارات بطرق ومصطلحات مختلفة، إضافةً إلى تنظيمها، إلا أن مجتمع مبرمجي لغة رست ينظر إلى الاختبارات بكونها تنتمي إلى أحد التصنيفين الرئيسين: اختبارات الوحدة unit tests واختبارات التكامل integration tests. تعدّ اختبارات الوحدة اختبارات صغيرة ومحددة على وحدة معيّنة منعزلة ويُمكن أن تختبر الواجهات الخاصة private interfaces، بينما تكون اختبارات التكامل خارجية كليًا لمكتبتك وتستخدم شيفرتك البرمجية بالطريقة ذاتها التي تستخدم فيها شيفرة برمجية خارجية اعتيادية شيفرتك البرمجية باستخدام الواجهات العامة public interface وتتضمن غالبًا أكثر من وحدة ضمن الاختبار الواحد.

كتابة نوعَي الاختبارات مهمٌ للتأكد من أن أجزاء في مكتبتك تنجز ما هو مطلوب منها بغض النظر عن باقي الأجزاء.

11.3.1 اختبارات الوحدة

الهدف من اختبارات الوحدة هو اختبار كل وحدة من شيفرة برمجية بمعزل عن الشيفرة البرمجية المتبقية، وذلك لتشخيص النقطة التي لا تعمل فيها الشيفرة البرمجية بصورةٍ صحيحة وبدقة، لذلك ستضع اختبارات الوحدة في المجلد "src" في كل ملف مع الشيفرة البرمجة التي تختبرها، ويقتضي الاصطلاح بإنشاء وحدة تدعى `tests` في كل ملف لاحتواء دوال الاختبار وتوصيف الوحدة باستخدام `cfg(test)`.

ا. وحدة الاختبارات وتوصيف #[cfg(test)]

يخبر توصيف `#[cfg(test)]` على وحدة الاختبارات رست بأنه يجب تصريف وتشغيل شيفرة الاختبار البرمجية فقط عند تنفيذ `cargo test` وليس عند تنفيذ `cargo build`، مما يختصر وقتًا من عملية التصريف عندما تريد فقط بناء المكتبة وتوفير المساحة التي سيشغلها الملف المُصرَّف الناتج وذلك لأن الاختبارات غير مُضمّنة به. توضع اختبارات التكامل في مجلد مختلف ولا تستخدم التوصيف `#[cfg(test)]`، إلا أنك بحاجة لاستخدام `#[cfg(test)]` لتحديد أنها لا يجب أن تُضمَّن في الملف المصرَّف الناتج لأن اختبارات الوحدة موجودة في ملف الشيفرة البرمجية ذاته.

تذكر أن كارجو Cargo ولّد الشيفرة البرمجية التالية لنا عندما ولدنا المشروع `adder` الجديد سابقًا:

اسم الملف: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
```

```
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

تُولّد هـذه الشـيفرة البرمجيـة تلقائيًـا على هيئـة وحـدة اختبـار. تمثّـل السـمة `cfg` اختصـارًا لكلمـة الضـبط configuration، إذ تُعلم رست أن العنصر الآتي يجب أن يُضمَّن فقط في خيار ضـبط معيّن، وفي هـذه الحالـة فإن خيار الضبط `test` الموجود في رست لتصريف وتنفيذ الاختبارات. يصرّف كارجو شـيفرة الاختبـار البرمجيـة فقط في حال تنفيذ الاختبارات بفعاليـة باسـتخدام `cargo test`، وهـذا يتضـمّن أي دوال مسـاعدة يمكن أن تكون داخل هذه الوحدة، إضافةً للدوال الموصّفة باستخدام `#[test]`.

ب. اختبار الدوال الخاصة

هناك اختلافٌ بين المبرمجين بخصوص الاختبارات وبالأخص اختبـار الـدوال الخاصـة، إذ يعتقـد البعض أن الدوال الخاصة يجب أن تُختبر مباشرةً، بينما لا يتفق البعض الآخر مع ذلك، وتجعل لغات البرمجة اختبار الـدوال الخاصة صعبًا أو مستحيلًا، وبغض النظر عمّا تعتقد بخصوص هذا الشأن، تسمح قـوانين خصوصـية رسـت لـك باختبار الدوال الخاصة. ألقِ نظرةً على الشيفرة 12 التي تحتوي على الشيفرة الخاصة `internal_adder`.

اسم الملف: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    internal_adder(a, 2)
}

fn internal_adder(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2));
    }
}
```

[الشيفرة 12: اختبار دالة خاصة]

لاحظ أن الدالة `internal_adder` ليست دالة عامة (لا تحتوي على `pub`). الاختبارات هي شيفرة برمجية مكتوبة بلغة رست وبالتالي تمثل وحدة `tests` وحدة اعتيادية، وكما ناقشنا سابقًا يمكن العناصر في الوحدات الابن استخدام العناصر الموجودة في الوحدات الأب، وفي هذا الاختبار نُضيف كل عناصر الوحدات الأب الخاصة بالوحدة `test` إلى النطاق بكتابة `use super::*`، بحيث يمكننا استدعاء `internal_adder` فيما بعد. إذا لم تكن مقتنعًا بأن الدوال الخاصة يجب أن تُختَبر فلن تجبرك رست على ذلك.

11.3.2 اختبارات التكامل Integration Tests

اختبارات التكامل Integration Tests في رست خارجية external كليًا بالنسبة لمكتبتك، وتستخدم هذه الاختبارات مكتبتك بالطريقة ذاتها لأي شيفرة برمجية، مما يعني أنها يمكن أن تستدعي دوال تشكل جزءًا من الواجهة البرمجية العامة Public API للمكتبة. الهدف من هذا النوع من الاختبارات هو اختبار ما إذا كانت أجزاء من مكتبتك تعمل مع بعضها بعضًا بصورةٍ صحيحة، إذ أن بعض الأجزاء من الشيفرة البرمجية قد تعمل بصورةٍ صحيحة لوحدها ولكن تواجه بعض المشاكل عند تكاملها مع أجزاء أخرى، لذا يُغطّي هذا النوع من الاختبارات الشيفرة البرمجية المتكاملة. نحتاج لإنشاء اختبارات التكامل أولًا إلى مجلد tests.

ا. مجلد tests

نُنشئ مجلد "tests" في المستوى الأعلى لمجلد مشروعنا بجانب "src"، إذ يتعرّف كارجو على المجلد والاختبارات التي بداخله، ويمكننا إنشاء ملفات اختبار قدر ما شئنا، وسيصرّف كارجو كل ملف اختبار بدوره بكونه وحدة مصرّفة crate منفصلة.

لننشئ اختبار تكامل باستخدام الشيفرة البرمجية الموجودة في الشيفرة 12 الموجودة في الملف src/lib.rs، نبدأ أولًا بإنشاء مجلد tests ونُنشئ ملفًا جديدًا نسميه tests/integration_test.rs. يجب أن يبدو هيكل المجلد كما يلي:

```
adder
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    └── integration_test.rs
```

أدخل الشيفرة البرمجية في الشيفرة 13 إلى الملف tests/integration_test.rs:

اسم الملف: tests/integration_test.rs

```
use adder;
```

```
#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

[الشيفرة 13: اختبار تكامل لدالة في الوحدة المصرَّفة `adder`]

يمثل كل ملف في المجلد "tests" وحدة مصرفة منعزلة، لذا يجب علينا إضافة مكتبتنا إلى نطاق وحدة الاختبار المصرَّفة، ونُضيف لهذا السبب `use adder` في بداية الشيفرة البرمجية وهو ما لم نحتاجه سابقًا عند استخدامنا لاختبارات الوحدة.

ليس علينا توصّف الشيفرة البرمجية في tests/integration_test.rs باستخدام `#[cfg(test)]`، إذ أنّ كارجو تُعامل المجلد "tests" على نحوٍ خاص وتُصرِّف جميع الملفات في هذا المجلد عند تنفيذ الأمر `cargo test`. ننفّذ `cargo test` فنحصل على التالي:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 1.31s
     Running unittests src/lib.rs (target/debug/deps/adder-
1082c4b063a8fbe6)

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

     Running tests/integration_test.rs
(target/debug/deps/integration_test-1082c4b063a8fbe6)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

   Doc-tests adder
```

```
running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

تتضمن أقسام الخرج الثلاثة اختبارات الوحدة والتكامل والتوثيق، لاحظ أنه إذا فشل أي اختبار في قسم مـا، لن يُنفَّذ القسم التالي. على سبيل المثال، إذا فشل اختبار وحدة فهذا يعني أنه لن يكـون هنـاك أي خـرج لاختبـار التكامل واختبار التوثيق لأن هذه الاختبارت ستُنفَّذ فقط في حال نجاح جميع اختبارات الوحدة.

القسم الأول هو لاختبارات الوحدة وهو مشابه لما رأيناه سابقًا، إذ يُخصّص كل سطر لاختبار وحدة ما (هناك اختبـار واحـد يسـمى `internal` أضـفناه سـابقًا في الشـيفرة 12) بالإضـافة إلى سـطر الملخص لنتـائج اختبارات الوحدة.

يبدأ قسـم اختبـارات التكامـل بالسـطر `Running tests/integration_test.rs`، ومن ثم نلاحـظ سـطرًا لكـل دالـة اختبـار في اختبـار التكامـل ذلـك مـع سـطر ملخص لنتـائج اختبـار التكامـل قبـل بـدء القسم `Doc-tests adder`.

يحتوي كل اختبار تكامل على قسمه الخـاص، لـذا سنحصـل على المزيـد من الأقسـام إن أضـفنا مزيـدًا من الملفات في المجلد "tests".

يمكننا تنفيذ دالة اختبار معيّنة بتحديد اسم دالة الاختبار مثل وسيط للأمـر `cargo test`، ولتنفيـذ جميـع الاختبارات في ملف اختبـار تكامـل معيّن نسـتخدم الوسـيط `--test` في الأمـر `cargo test` متبوعًـا باسـم الملف كما يلي:

```
$ cargo test --test integration_test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.64s
     Running tests/integration_test.rs
(target/debug/deps/integration_test-82e7799c1bc62298)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

ينفّذ هذا الأمر الاختبارات الموجودة في ملف "tests/integration_test.rs" فقط.

ب. الوحدات الجزئية في اختبارات التكامل

قد تحتاج إنشاء المزيد من الملفات في المجلد "tests" لمساعدتك في تنظيم اختبارات التكامل في حال إضافتك للمزيد منها، على سبيل المثال يمكنك تجميع دوال الاختبار بناءً على الخاصية التي تفحصها، وكما ذكرنا سابقًا: يُصرَّف كل ملف في المجلد "tests" بمفرده على أنه وحدة مصرّفة، وهو أمر مفيد لإنشاء نطاقات متفرقة عن بعضها بعضًا لمحاكاة الطريقة التي يستخدم فيها المستخدمون وحدتك المصرفة، إلا أن هذا يعني أن الملفات في مجلد "tests" لن تشارك السلوك ذاته الخاص بالملفات في المجلد "src" كما تعلمت سابقًا بخصوص فصل الشيفرة البرمجية إلى وحدات وملفات.

يُلاحظ السلوك بملفات المجلد "tests" بوضوح عندما يكون لديك مجموعةً من الدوال المساعدة تريد استخدامها في ملفات اختبار تكامل مختلفة وتحاول أن تتبع الخطوات الخاصة بفصل الوحدات إلى ملفات مختلفة كما ناقشنا سابقًا لاستخلاصها إلى وحدة مشتركة. على سبيل المثال، إذا أنشأنا "tests/common.rs" ووضعنا دالةً اسمها `setup` في الملف، يمكننا إضافة شيفرة برمجية إلى `setup` لاستدعائها من دوال اختبار مختلفة في ملفات اختبار متعددة.

اسم الملف: tests/common.rs

```
pub fn setup() {
    // شيفرة ضبط برمجية مخصصة لاختبارات مكتبتك
}
```

عند تشغيل الاختبارات مجددًا سنرى قسمًا جديدًا في خرج الاختبار للملف "common.rs"، على الرغم من أننا لا نستدعي الدالة `setup` من أي مكان كما أن هذا الملف لا يحتوي على أي دوال اختبار:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.89s
     Running unittests src/lib.rs (target/debug/deps/adder-
92948b65e88960b4)

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

     Running tests/common.rs (target/debug/deps/common-
92948b65e88960b4)
```

```
running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

     Running tests/integration_test.rs
(target/debug/deps/integration_test-92948b65e88960b4)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

العثور على `common` في نتائج الاختبار مع `running 0 tests` ليس ما أردنا رؤيته، إذا أننا أردنا مشاركة جزء من شيفرة برمجية مع ملفات اختبار التكامل الأخرى.

لتجنب الحصـــول على `common` في خـــرج الاختبـــار، نُنشـــئ "tests/common/mod.rs" بـــدلًا من "tests/common.rs"، بحيث يبدو هيكل مجلد المشروع كما يلي:

```
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    ├── common
    │   └── mod.rs
    └── integration_test.rs
```

هذا هو اصطلاح التسمية القديم في رست وقد ذكرناه سـابقًا في الفصـل 7، إذ تخـبر تسـمية الملـف بهـذا الاسم رست بعدم التعامل مع الوحدة `common` على أنها ملف اختبار تكامل، وبالتـالي لن يظهـر هـذا القسـم في

خرج الاختبار بعد أن ننقل شيفرة الدالة البرمجية `setup` إلى "tests/common/mod.rs" ونحذف الملف "tests/common.rs". لا تُصرَّف الملفات الموجودة في المجلدات الفرعية في المجلد tests مثل وحدات مصرّفة متفرقة أو تحتوي على أقسام متفرقة في خرج الاختبار.

يمكننا استخدام أي من ملفات اختبار التكامل مثل وحدة بعد إنشاء الملف "tests/common/mod.rs"، إليك مثالًا على استدعاء الدالة `setup` من الاختبار `it_adds_two` في "tests/integration_test.rs":

اسم الملف: tests/integration_tests.rs

```
use adder;

mod common;

#[test]
fn it_adds_two() {
    common::setup();
    assert_eq!(4, adder::add_two(2));
}
```

لاحظ أن التصريح `mod common;` مماثلٌ لتصريح الوحدة التي شرحنا عنها سابقًا في الشيفرة 21 من الفصل 7. يمكننا بعد ذلك استدعاء الدالة `common::setup()` من دالة الاختبار.

ج. اختبارات التكامل للوحدات الثنائية المصرفة

لا يمكننا إنشاء اختبارات تكامل في المجلد "tests" وإضافة الدوال المعرفة في الملف "src/main.rs" إلى النطاق باستخدام تعليمة `use` إذا كان مشروعنا يمثّل وحدة ثنائية مصرفة binary crate تحتوي على ملف "src/main.rs" فقط ولا تحتوي على ملف "src/lib.rs"، إذ أن وحدات المكتبة المصرفة وحدها قادرة على كشف الدوال التي يمكن للوحدات المصرّفة الأخرى استخدامها؛ لأن الهدف من الوحدات الثنائية المصرفة هو تنفيذها بصورةٍ مستقلة.

هذا واحدٌ من الأسباب لكون مشاريع رست التي تدعم ملفات ثنائية تأتي بملف "src/main.rs"، الذي يستدعي المنطق البرمجي الموجود في الملف src/lib.rs. يمكن لاختبارات التكامل باستخدام هذا التنظيم بأن تختبر صندوق المكتبة المصرف باستخدام `use` لجعل الدوال المهمة متاحة، فإذا عملت الوظيفة الأساسية، عنى ذلك أن الأجزاء الصغيرة من الشيفرة البرمجية في الملف src/main.rs ستعمل أيضًا ويجب اختبار هذه الأجزاء الصغيرة.

11.4 خاتمة

تزوّدك مزايا الاختبار في رست بطريقة لتحديد الكيفية التي تعمل بها الشيفرة البرمجية للتأكد من أنها مستمرة بالعمل كما تتوقع، حتى لو أجريت بعض التغييرات. تستخدم اختبارات الوحدة أجزاء مختلفة من المكتبة بصورةٍ منفصلة ويمكن اختبار تفاصيل التطبيق الخاصة. تتفقد اختبارات التكامل العديد من الأجزاء في المكتبة لرؤية إذا كانت تعمل جيدًا مع بعضها البعض وتستخدم واجهة المكتبة البرمجية العامة لتجربة الشيفرة البرمجية بالطريقة ذاتها التي تستخدم بها الشيفرة البرمجية الخارجية. على الرغم من أن نظام أنواع رست وقواعد الملكية تساعد في التقليل من الأخطاء البرمجية إلا أن الاختبارات مهمة للتقليل من الأخطاء المنطقية التي تتعلق بسلوك الشيفرة البرمجية بذات نفسها.

دعنا نجمع المعرفة التي تعلمناها في هذا الفصل وفي الفصل الذي سبقه لنعمل على مشروع.

خمسات

لبيع وشراء الخدمات المصغرة

أكبر سوق عربي لبيع وشراء الخدمات المصغرة
اعرض خدماتك أو احصل على ما تريد بأسعار تبدأ من $5 فقط

تصفح الخدمات

12. التعامل مع الدخل والخرج: كتابة برنامج سطر أوامر Command Line

يمثّل هذا الفصل تطبيقًا لجميع المهارات التي تعلمتها حتى الآن، ونظرةً عمليةً إلى المزيد من المزايا الموجودة في المكتبة القياسية Standard Library. سنبني سويًّا أداة سطر أوامر command line tool تتفاعل مع ملف وتُجري عمليات الدخل والخرج باستخدام سطر الأوامر للتمرُّن على بعض مفاهيم رست التي تعلمتها حتى هذه اللحظة.

تجعل السرعة والأمان والخرج الثنائي الوحيد ودعم مختلف المنصات cross-platform من رست لغةً مثالية لإنشاء أدوات السطر البرمجي، لذا سيكون مشروعنا هو إصدار خاص من أداة بحث سطر الأوامر الكلاسيكية "grep" (اختصارًا للبحث العام باستخدام التعابير النمطية والطباعة **g**lobally search a **r**egular **e**xpression and **p**rint). يبحث "grep" في حالات الاستخدام الأسهل على سلسلة نصية string محددة داخل ملف معين، ولتحقيق ذلك يأخذ "grep" مسار الملف وسلسلة نصية مثل وسطاء له، ثم يقرأ الملف ويجد الأسطر التي تحتوي على وسيط السلسلة النصية داخل الملف ويطبع هذه الأسطر.

سنوضّح لك كيف ستستخدم أداة سطر الأوامر لخصائص سطر الأوامر وهو ما تستخدمه العديد من أدوات سطر الأوامر الأخرى، إذ سنقرأ قيمة متغير البيئة environment variable للسماح للمستخدم بضبط سلوك أداتنا، كما أننا سنطبع رسالة خطأ إلى مجرى الخطأ القياسي الخاص بالطرفية standard error console stream -أو اختصارًا stderr- بدلًا من الخرج القياسي standard output -أو اختصارًا stdout. لذلك، يمكن للمستخدم مثلًا إعادة توجيه الخرج الناجح إلى ملف بحيث يظل قادرًا على رؤية رسالة الخطأ على الشاشة في الوقت ذاته.

أنشأ عضو من مجتمع رست يدعى آندرو غالانت Andrew Gallant إصدارًا سريعًا ومليئًا بالمزايا من grep وأطلق عليه تسمية ripgrep. سيكون إصدارنا الذي سننشئه في هذا الفصل أكثر بساطةً إلا أن هذا الفصل ستمنحك بعض الأساسيات التي تحتاج إليها لتفهم المشاريع العملية الواقعية مثل ripgrep.

سيجمع مشروع grep الخاص بنا عددًا من المفاهيم التي تعلمناها سابقًا:

- تنظيم الشيفرة البرمجية (استخدام ما تعلمناه بخصوص الوحدات modules في الفصل 7)
- استخدام الأشعة vectors والسلاسل النصية (في الفصل 8)
- التعامل مع الأخطاء (في الفصل 9)
- استخدام السمات traits ودورات الحياة lifetimes عند الحاجة (في الفصل 10)
- كتابة الاختبارات (في الفصل 11)

سنتكلم أيضًا بإيجاز عن المغلّفات closures والمكرّرات iterators وسمة الكائنات trait objects، إلا أننا سنتكلم عن هذه المفاهيم بالتفصيل لاحقًا.

12.1 الحصول على الوسطاء من سطر الأوامر

دعنا نُنشئ مشروعًا جديدًا باستخدام `cargo new` كما اعتدنا، سنسمّي مشروعنا باسم "minigrep" للتمييز بينه وبين الأداة `grep` التي قد تكون موجودةً على نظامك مسبقًا.

```
$ cargo new minigrep
     Created binary (application) `minigrep` project
$ cd minigrep
```

المهمة الأولى هي بجعل `minigrep` يقبل وسيطين من سطر الأوامر، ألا وهما مسار الملف وسلسلة نصية للبحث عنها، أي أننا نريد أن نكون قادرين على تنفيذ برنامجنا باستخدام `cargo run` متبوعًا بشرطتين للدلالة على أن الوسطاء التي ستتبعها تنتمي للبرنامج الذي نريد أن ننفذه وليس لكارجو cargo، سيكون لدينا وسيطان أولهما سلسلة نصية للبحث عنها وثانيهما مسار الملف الذي نريد البحث بداخله على النحو التالي:

```
$ cargo run -- searchstring example-filename.txt
```

لا يستطيع البرنامج المولَّد باستخدام `cargo new` حاليًا معالجة الوسطاء التي نمرّرها له، ويمكن أن تساعدك بعض المكتبات الموجودة على crates.io بكتابة برامج تقبل وسطاء سطر الأوامر، إلا أننا سنطبّق هذا الأمر بأنفسنا بهدف التعلُّم.

12.1.1 قراءة قيم الوسطاء

سنحتاج لاستخدام الدالة `std::env::args` الموجودة في مكتبة رست القياسية لتمكين `minigrep` من قراءة قيم وسطاء سطر الأوامر التي نمرّرها إليه، إذ تُعيد هذه الدالة مكرّرًا iterator إلى وسطاء سطر الأوامر الممرّرة إلى `minigrep`. سنغطّي المكررات لاحقًا، إلا أنه من الكافي الآن معرفتك معلومتين بخصوص المكررات: تنتج المكررات مجموعةً من القيم، ويمكننا استدعاء التابع `collect` على مكرّر لتحويله إلى تجميعة collection مثل الأشعة التي تحتوي جميع العناصر التي تنتجها المكررات.

تسمح الشيفرة 1 لبرنامجك `minigrep` بقراءة أي وسطاء سطر أوامر تمررها إليه، ثمّ تجمّع القيم في شعاع.

اسم الملف: src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    dbg!(args);
}
```

[الشيفرة 1: تجميع وسطاء سطر الأوامر في شعاع وطباعتها]

نُضيف أوّلًا الوحدة `std::env` إلى النطاق scope باستخدام تعليمة `use`، بحيث يمكننا استخدام الدالة `args` المحتواة داخلها. لاحظ أن الدالة `std::env::args` متداخلة nested مع مستويين من الوحدات. سنختار إحضار الوحدة الأب إلى النطاق بدلًا من الدالة كما ناقشنا سابقًا في الحالات التي تكون فيها الدالة المطلوبة متداخلة مع أكثر من وحدة واحدة، ويمكننا بذلك استخدام الدوال الأخرى من `std::env`، كما أن هذه الطريقة أقل غموضًا من إضافة `use std::env::args`، ثم استدعاء الدالة بكتابة `args` فقط لأن `args` قد يُنظر إليها بكونها دالة معرّفة بالوحدة الحالية بصورةٍ خاطئة.

> ستهلع `std::env::args` إذا احتوى أي وسيط على يونيكود غير صالح، وإذا احتاج البرنامج لقبول الوسطاء التي تحتوي على يونيكود غير صالح، فعليك استخدام `std::env::args_os` بدلًا منها، إذ تعيد هذه الدالة مكرّرًا ينتج قيم `OsString` بدلًا من قيم `String`، وقد اخترنا استخدام `std::env::args` هنا لبساطتها، ونظرًا لاختلاف قيم `OsString` بحسب المنصة وهي أكثر تعقيدًا في التعامل معها مقارنةً بقيم `String`.

نستدعي `env::args` في السطر الأول من `main` ومن ثم نستخدم `collect` مباشرةً لتحويل المكرّر إلى شعاع يحتوي على جميع القيم الموجودة في المكرّر، ويمكننا استخدام `collect` هنا لإنشاء عدة أنواع من التجميعات، لذا يجب أن نشير صراحةً لنوع `args` لتحديد أننا نريد شعاع من السلاسل النصية، وعلى الرغم من

أن تحديد الأنواع في رست نادر، إلا أن `collect` دالة يجب أن تحدد فيها النوع عادةً، لأن رست لا تستطيع استنتاج نوع التجميعة التي تريدها.

أخيرًا، نطبع الشعاع باستخدام ماكرو تنقيح الأخطاء debug macro. دعنا نجرّب تنفيذ الشيفرة البرمجية أولًا دون استخدام أي وسطاء ومن ثم باستخدام وسيطين:

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.61s
     Running `target/debug/minigrep`
[src/main.rs:5] args = [
    "target/debug/minigrep",
]
$ cargo run -- needle haystack
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 1.57s
     Running `target/debug/minigrep needle haystack`
[src/main.rs:5] args = [
    "target/debug/minigrep",
    "needle",
    "haystack",
]
```

لاحظ أن القيمة الأولى في الشعاع هي `"target/debug/minigrep"` وهو اسم ملفنا التنفيذي، وهذا يطابق سلوك لائحة الوسطاء في لغة سي C بالسماح للبرامج باستخدام الاسم الذي كان السبب في بدء تنفيذها، ومن الملائم عادةً الحصول على اسم البرنامج في حال أردت طباعته ضمن رسائل، أو تعديل سلوك البرنامج المبني على اسم سطر الأوامر البديل command line alias الذي نستخدمه لبدء تشغيل البرنامج، وسنتجاهله الآن ونحفظ فقط أول وسيطين نستخدمهما.

12.1.2 حفظ قيم الوسطاء في متغيرات

يستطيع البرنامج حاليًا الحصول على القيم المحدّدة مثل وسطاء سطر أوامر، أما الآن فنحن بحاجة لحفظ قيم الوسيطين في متغيرين بحيث يمكننا استخدام المتغيرين ضمن بقية البرنامج، وهو ما فعلناه في الشيفرة 2.

اسم الملف: src/main.rs

```
use std::env;
```

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let query = &args[1];
    let file_path = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", file_path);
}
```

[الشيفرة 2: إنشاء متغيرين لتخزين وسيط السلسلة النصية ووسيط مسار الملف]

يحتلّ اسم البرنامج القيمة الأولى في الشعاع عند `args[0]` كما رأينا سابقًا عندما طبعنا الشعاع، لذا نبدأ من الدليل "1" إذ أن الوسيط الأول للبرنامج `minigrep` هو السلسلة النصية التي سنبحث عنها، لذا نضع مرجعًا reference على أول وسيط في المتغير `query`، بينما يمثل الوسيط الثاني مسار الملف، لذا نضع مرجعًا عليه في المتغير `file_path`.

نطبع مؤقتًا قيم المتغيرَين لنتأكد من أن الشيفرة البرمجية تعمل وفق ما هو متوقع. دعنا ننفذ البرنامج مجددًا بالوسيطين `test` و `sample.txt`:

```
$ cargo run -- test sample.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep test sample.txt`
Searching for test
In file sample.txt
```

عظيم، يعمل برنامجنا بنجاح، إذ تُمرّر قيم الوسطاء التي نحتاجها وتُحفظ في المتغيرات المناسبة. سنضيف لاحقًا شيفرةً برمجيةً للتعامل مع الأخطاء في حالات استخدام خاطئة محتملة، مثل الحالة التي لا يدخل فيها المستخدم أي وسطاء، والتي سنتجاهلها الآن ونبدأ بالعمل على إضافة شيفرة برمجية لقراء الملف بدلًا من ذلك.

12.2 قراءة ملف

سنضيف الآن إمكانية قراءة الملف المحدد في الوسيط `file_path`. نحتاج أولًا لملف تجريبي لتجربة البرنامج باستخدامه، وسنستخدم ملفًا يحتوي على نصٍ قصير يحتوي على عدّة أسطر مع كلمات مكرّرة. تحتوي

الشـيفرة 3 على قصـيدة لإيميلي ديكـنز Emily Dickinson وهـو مـا سنسـتخدمه هنـا. أنشـئ ملفًـا يـدعى "poem.txt" في مستوى جذر المشروع وأدخل قصيدة "أنا لا أحد! من أنت؟ I'm Nobody! Who are you?".

اسم الملف: poem.txt

```
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

[الشيفرة 3: قصيدة إيميلي ديكنز تمثّل ملف تجريبي مناسب]

بعد تهيئتك للنص، عدّل الملف "src/main.rs" وضِف شيفرة برمجية لقـراءة الملـف كمـا هـو موضـح في الشيفرة 4.

اسم الملف: src/main.rs

```
use std::env;
use std::fs;

fn main() {
    // --snip--
    println!("In file {}", file_path);

    let contents = fs::read_to_string(file_path)
        .expect("Should have been able to read the file");

    println!("With text:\n{contents}");
}
```

[الشيفرة 4: قراءة محتويات الملف المحدّد وفق الوسيط الثاني]

أضفنا أولًا جزءًا متعلقًا بالبرنامج من المكتبة القياسية باستخدام تعليمـة `use` إلى النطـاق، لأننـا نحتـاج إلى `std::fs` إلى التعامل مع الملفات.

تأخـذ التعليمـة `fs::read_to_string` الجديـدة في الدالـة `main` القيمـة `file_path`، ثم تفتح ذلـك الملف وتُعيد قيمةً من النوع `std::io::Result<String>` تمثّل محتويات الملف.

أضفنا مجددًا تعليمة `println!` مؤقتة تطبع قيمة `contents` بعـد قـراءة الملـف، حـتى نتأكـد من عمـل البرنامج بصورةٍ صحيحة.

لننفّذ هذه الشيفرة البرمجية باستخدام أي سلسلة نصية تمثّل وسيط سـطر أوامـر أول (لأننـا لم نطبـق جـزء البحث عن السلسلة النصية بعد) والملف "poem.txt" بمثابة وسيط ثاني:

```
$ cargo run -- the poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

عظيم، تقرأ الآن الشيفرة البرمجية محتويات الملف ثم تطبعها، إلا أن الشـيفرة البرمجيـة تحتـوي على بعض الثغرات، إذ تحتوي الدالة `main` الآن على عدّة مسؤوليات، ومن الأفضل عمومًا استخدام دالة واحـدة لمسـؤولية واحدة للحصول على دوال أسهل بالتعامل وأوضح، والمشكلة الثانية هي أننا لم نتعامـل مـع الأخطـاء كمـا ينبغي لنا، إلا أن البرنامج ما زال صغيرًا وبالتالي لا تشكل هذه المشاكل تهديدًا كبيرًا، لكنهـا ستصـبح صـعبة الحـل مـع زيادة حجم البرنامج، فمن الأفضل إعادة بناء التعليمات البرمجية refactor بمرحلة مبكرة من تطوير البرنـامج لأن إعادة بناء التعليمات البرمجية سيكون أسهل بكثير من كميات قليلة من الشيفرات البرمجية، لذا دعنا نفعل ذلك تاليًا في القسم التالي.

12.3 إعادة بناء التعليمات البرمجية لتحسين النمطية Modularity والتعامل مع الأخطاء

بدأنا في القسم السابق بناء المشروع وسنكمل العملية في هذا الفصل حيث سنصلح أربع مشكلات خاصة بهيكل البرنامج وكيفية تعامله مع الأخطاء المحتملة لتحسين برنامجنا. تتمثل المشكلة الأولى في أن للدالة `main` مهمتان: المرور على لائحة الوسطاء وقراءة الملفات. سيزداد عدد المهام المتفرقة التي تنجزها الدالة `main` مع نموّ حجم برنامجنا، وتصبح الدالة التي تحتوي على الكثير من المهام صعبة الفهم والاختبار والتعديل دون المخاطرة بتعطيل بعض خصائصها، ومن الأفضل فصل المهام عن بعضها بعضًا بحيث تكون كل دالة مسؤولةً عن مهمةٍ واحدة.

تمتدّ المشكلة إلى مشكلة أخرى ثانية: على الرغم من أن `query` و `file_path` تمثلان متغيرات بيئة لبرنامجنا إلا أن متغيرات مثل `contents` تُستخدم في برنامجنا لتنفيذ المنطق، ومع زيادة حجم الدالة `main` سيزيد عدد المتغيرات التي سنحتاج إضافتها إلى النطاق مما سيصعّب مهمة متابعة قيمة كل منها، ومن الأفضل تجميع متغيرات الضبط في هيكل واحدة لجعل الهدف منها واضح.

المشكلة الثالثة هي أننا استخدمنا `expect` لطباعة رسالة خطأ عندما تفشل عملية قراءة الملف، إلا أن رسالة الخطأ تقتصر على طباعة "Should have been able to read the file"، وقد تفشل قراءة ملف ما لعدّة أسباب؛ إذ يمكن أن يكون الملف مفقودًا؛ أو أنك لا تمتلك الأذونات المناسبة لفتحه، وحاليًا فنحن نعرض رسالة الخطأ ذاتها بغض النظر عن سبب الخطأ، وهو أمرٌ لن يمنح المستخدم أي معلومات مفيدة.

رابعًا، استخدمنا `expect` بصورةٍ متكررة لنتعامل مع الأخطاء المختلفة وإذا نفّذ المستخدم البرنامج دون تحديد عددٍ كافٍ من الوسطاء فسيحصل على الخطأ "index out of bounds" من رست، وهو خطأ لا يشرح بدقة سبب المشكلة. يُفضَّل هنا أن يكون التعامل مع الأخطاء موجودًا في مكان واحد بحيث يعلم المبرمج الذي يعمل على تطوير البرنامج مستقبلًا المكان الذي يجب التوجه إليه في حال أراد تغيير منطق التعامل مع الأخطاء، كما سيسهّل وجود الشيفرة البرمجية التي تتعامل مع الأخطاء في مكان واحد عملية طباعة رسائل خطأ معبّرة للمستخدم.

دعنا نصلح هذه المشاكل الأربع بإعادة بناء التعليمات البرمجية.

12.3.1 فصل المهام في المشاريع التنفيذية

مشكلة تنظيم المسؤوليات المختلفة بالنسبة للدالة `main` هي مشكلة شائعة في الكثير من المشاريع التنفيذية binary projects، ونتيجةً لذلك طوّر مجتمع رست توجيهات عامة لفصل المهام المختلفة الموجودة في البرنامج التنفيذي عندما تصبح الدالة `main` كبيرة، وتتلخص هذه العملية بالخطوات التالية:

- تجزئة برنامجك إلى "main.rs" و "lib.rs" ونقل منطق البرنامج إلى "lib.rs".

- يمكن أن يبقى منطق الحصول على الوسطاء من سطر الأوامر في "main.rs" طالما هو قصير.
- عندما يصبح منطق الحصول على الوسطاء من سطر الأوامر معقّدًا صدِّره من "main.rs" إلى "lib.rs".

يجب أن تكون المهام التي يجب أن تبقى في `main` بعد هذه العملية محصورةً بما يلي:

- استدعاء منطق الحصول على وسطاء سطر الأوامر باستخدام قيم الوسطاء.
- إعداد أي ضبط لازم.
- استدعاء الدالة `run` في "lib.rs".
- التعامل مع الخطأ إذا أعادت الدالة `run` خطأ.

يحلّ هذا النمط كل ما يتعلق بفصل المهام، إذ يتعامل "main.rs" بكل شيء يخص تشغيل البرنامج، بينما يتعامل "lib.rs" مع منطق المهمة المطروحة. بما أنك لا تستطيع اختبار الدالة `main` مباشرةً، سيساعدك هذا الهيكل في اختبار كل منطق برنامجك بنقله إلى دوال موجودة في "lib.rs"، وستكون الشيفرة البرمجية المتبقية في "main.rs" قصيرة وسيكون التحقق من صحتها بالنظر إليها ببساطة كافيًا. دعنا نعيد كتابة التعليمات البرمجية باستخدام هذه الخطوات.

12.3.2 استخلاص الشيفرة البرمجية التي تحصل على الوسطاء

سنستخرج خاصية الحصول على الوسطاء إلى دالة تستدعيها `main` لتحضير نقل منطق سطر الأوامر إلى src/lib.rs. توضح الشيفرة 5 بدايةً جديدةً من الدالة `main` تستدعي دالةً جديدة تدعى `parse_config` وسنعرّفها حاليًا في src/main.rs.

اسم الملف: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let (query, file_path) = parse_config(&args);

    // --snip--
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let query = &args[1];
    let file_path = &args[2];
```

```
    (query, file_path)
}
```

[الشيفرة 5: استخراج الدالة `parse_config` من `main`]

ما زلنا نجمع وسطاء سطر الأوامر في شعاع، إلا أننا نمرّر الشعاع كاملًا إلى الدالة `parse_config` بـدلًا من إسناد القيمة في الدليل "1" إلى المتغير `query` والقيمة في الدليل "2" إلى المتغير `file_path` داخل الدالـة `main`، إذ تحتوي الدالة `parse_config` على المنطق الذي يحدّد أي القيمتين سيُخزَّن في أي المتغيّـرين ومن ثم تعـديل القيم إلى الدالـة `main`، إلا أننـا مـا زلنـا نُنشـئ المتغـيرين `query` و `file_path` في `main`، لكن لا تحمل `main` مسؤولية تحديد أي وسطاء سطر الأوامر تنتمي إلى أي المتغيرات.

قد تبدو هذه الإضافة مبالغةً شديدةً في برنامجنا البسيط هذا، إلا أننا نعيد بناء التعليمات البرمجية بخطوات صغيرة وتدريجية. نفّذ هذا البرنامج بعد تطبيق التعديلات لتتأكد من أن الحصول على الوسطاء ما زال يعمـل. من المحبّذ التحقق من تنفيذ البرنامج بعد كل تعديل بحيث تستطيع معرفة سبب المشكلة فورًا إذا حصلت.

ا. تجميع قيم الضبط

يمكننا اتخاذ خطوة بسيطة لتحسين الدالة `parse_config` أكثر، إذ أننا نعيد حاليًا صف tuple على الـرغم من أننا نجزّء هذا الصف مباشرةً إلى أجزاء متفرقة من جديد، وهذا يشير إلى أننا لا نطبّق الفكرة صحيحًا.

كون `config` جزءًا من `parse_config` هو مؤشر آخر لوجود احتمالية تحسين، إذ يشير ذلك أن القيمتين التي نُعيدهما مترابطتان وهما جزء من قيمة ضبط واحدة، إلا أننا لا ننقل هذا المعنى بهيكـل البيانـات، كمـا أننـا نجمع القيمتين في صف. دعنا نضع القيمتين بدلًا من ذلك في هيكل واحد ونمنح كلًا من حقـول الهيكـل اسـمًا معبرًا. ستكون الشـيفرة البرمجيـة بعـد ذلـك أسـهل فهمًـا للمطـورين الـذين سـيعملون على الشـيفرة البرمجيـة مستقبلًا، إذ سيوضّح ذلك كيف ترتبط القيم المختلفة مع بعضها بعضًا وهدف كل واحدة منها.

توضح الشيفرة 6 التحسينات التي أجريناها على الدالة `parse_config`.

اسم الملف: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = parse_config(&args);

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);
```

```
    let contents = fs::read_to_string(config.file_path)
        .expect("Should have been able to read the file");

    // --snip--
}

struct Config {
    query: String,
    file_path: String,
}

fn parse_config(args: &[String]) -> Config {
    let query = args[1].clone();
    let file_path = args[2].clone();

    Config { query, file_path }
}
```

[الشيفرة 6: إعادة بناء التعليمات البرمجية في الدالة `parse_config` لإعادة نسخة instance من الهيكل `Config`]

أَضفنا هيكلًا يدعى `Config` وعرّفناه، بحيث يحتوي على حقلين `query` و `file_path`. تشير بصمة signature الدالة `parse_config` الآن أنها تُعيد قيمةً من النوع `Config`، إلا أننا اعتدنا إعادة شرائح السلسلة النصية string slice التي تمثل مرجعًا للنوع `String` في `args`، لذلك نعرّف `Config` بحيث يحتوي على قيمتين مملوكتين owned من النوع `String`. المتغير `args` في `main` هو المالك لقيم الوسطاء ويسمح للدالة `parse_config` باستعارتها فقط، مما يعني أننا سنخرق قواعد الاستعارة في رست إذا حاول `Config` أخذ ملكية القيم من `args`.

هناك عدة طرق نستطيع فيها إدارة بيانات `String`، إلا أن أسهل الطرق غير فعال وهو باستدعاء التابع `clone` على القيم، مما سيعطينا نسخةً من البيانات بحيث تستطيع `Config` امتلاكها وهو أمرٌ يستغرق وقتًا ويشغل ذاكرةً أكبر مقارنةً باستخدام مرجع لبيانات السلسلة النصية، إلا أن نسخ البيانات يجعل من شيفرتنا البرمجية واضحةً لأنه ليس علينا وقتها إدارة دورات حياة المراجع، وفي هذه الحالة تُعد مقايضة الفعالية بالأداء بصورةٍ طفيفة مقابل البساطة أمرًا مقبولًا.

حدّثنا `main` بحيث تضع نسخة من الهيكل `Config` مُعادة بواسطة الدالة `parse_config` إلى متغير يدعى `config` وحدّثنا الشيفرة البرمجية التي استخدمت سابقًا المتغيرات `query` و `file_path` بصورةٍ منفصلة، إذ نستخدم الآن الحقول الموجودة في `Config` بدلًا من ذلك.

أصبحت شيفرتنا البرمجية الآن توضح بصورةٍ أفضل أن القيمتين `query` و `file_path` مترابطتان وأن الهدف منهما هو ضبط كيفية عمل البرنامج. أي شيفرة برمجية تستخدم هاتين القيمتين ستعثر عليهما في نسخة `config` في الحقول المسمّاة بحسب الهدف منهما.

حول سلبيات استخدام clone

يميل مبرمجو لغة رست لتفادي استخدام `clone` لتصحيح مشاكل الملكية بسبب الوقت الذي يستغرقه تنفيذها. ستتعلم لاحقًا كيفية استخدام توابع ذات كفاءة في حالات مشابهة لهذه، إلا أن نسخ بعض السلاسل النصية حاليًا أمرٌ مقبول لأنك تنسخ هذه القيم مرةً واحدةً فقط ومسار الملف والكلمة التي تبحث عنها ليستا بالحجم الكبير. من الأفضل وجود برنامج لا يعمل بفعالية مئةً بالمئة من محاولة زيادة فعالية الشيفرة البرمجية بصورةٍ مفرطة في محاولتك الأولى، إذ سيصبح الأمر أسهل بالنسبة لك مع اكتسابك للخبرة في رست، بحيث تستطيع كتابة حلول برمجية أكثر فاعلية ومن المقبول الآن استدعاء `clone`.

12.3.3 إنشاء باني للهيكل Config

استخلصنا بحلول هذه اللحظة المنطق المسؤول عن الحصول على قيم وسطاء سطر الأوامر من الدالة `main` ووضعناه في الدالة `parse_config`، وساعدنا هذا في رؤية كيفية ارتباط القيمتين `query` و `file_path` ببعضهما، ثم أضفنا هيكل `Config` لتسمية الهدف من القيمتين `query` و `file_path` ولنكون قادرين على إعادة أسماء القيم مثل حقول هيكل من الدالة `parse_config`.

إذًا، أصبح الآن الهدف من الدالة `parse_config` إنشاء نسخ من الهيكل `Config`، ويمكننا تعديل `parse_config` من دالة اعتيادية إلى دالة تدعى `new` مرتبطة بالهيكل `Config`، وسيؤدي هذا التعديل إلى جعل شيفرتنا البرمجية رسميةً idiomatic أكثر.

يمكننا إنشاء نسخ من الأنواع الموجودة في المكتبة القياسية مثل `String` باستدعاء `String::new`، وكذلك يمكننا بتعديل الدالة `parse_config` إلى دالة `new` مرتبطة مع الهيكل `Config` استدعاء `Config::new` للحصول على نسخ من `Config`. توضح الشيفرة 7 التعديلات التي نحتاج لإجرائها.

اسم الملف: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args);

    // --snip--
}
```

```
// --snip--

impl Config {
    fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let file_path = args[2].clone();

        Config { query, file_path }
    }
}
```

[الشيفرة 7: تغيير الدالة `parse_config` إلى `Config::new`]

حدّثنا الدالة `main` التي استدعينا فيها `parse_config` سابقًا لتستدعي `Config::new` بـدلًا من ذلـك، وعدّلنا اسم الدالـة `parse_config` إلى `new` ونقلناهـا لتصـبح داخـل كتلـة `impl`، ممـا يربـط الدالـة `new` مـع الهيكل `Config`. جرّب تصريف الشيفرة البرمجية مجددًا لتتأكد من أنها تعمل دون مشاكل.

12.3.4 تحسين التعامل مع الأخطاء

سنعمل الآن على تصحيح التعامل مع الأخطاء. تذكّر أن محاولتنا للوصـول إلى القيم الموجـودة في الشـعاع `args` في الدليل 1 أو الدليل 2 تسببت بهلع البرنامج في حال احتـواء الشـعاع أقـل من 3 عناصـر. جـرّب تنفيـذ البرنامج دون أي وسطاء، من المفترض أن تحصل على رسالة الخطأ التالية:

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep`
thread 'main' panicked at 'index out of bounds: the len is 1 but the
index is 1', src/main.rs:27:21
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

يمثل السطر `index out of bounds: the len is 1 but the index is 1` رسالة خطأ موجهة للمبرمجين، ولن تساعد مستخدم البرنامج لفهم الخطأ. دعنا نصلح ذلك الآن.

ا. تحسين رسالة الخطأ

سنُضيف في الشيفرة 8 اختبارًا في الدالة new يتأكد من أن الشريحة طويلة كفاية قبل محاولة الوصـول إلى الدليل 1 و2، وإذا لم كانت الشريحة طويلة كفاية، سيهلع البرنامج وسيعرض رسالة خطأ أفضل من الرسالة الـتي رأيناها سابقًا.

اسم الملف: src/main.rs

```
    // --snip--
    fn new(args: &[String]) -> Config {
        if args.len() < 3 {
            panic!("not enough arguments");
        }
        // --snip--
```

[الشيفرة 8: إضافة اختبار للتحقق من عدد الوسطاء]

الشيفرة البرمجية مشابهة لما فعلناه في قسم سابق عند كتابة الدالـة Guess::new، إذ اسـتدعينا المـاكرو panic! عندما كان الوسيط value خارج مجال القيم الصالحة، إلا أننا نفحص طـول args إذا كـان على الأقـل بطول 3 بدلًا من فحص مجال القيم، وتتابع بقية الدالة فيما بعد عملهـا بـافتراض أن الشـرط محقـق. إذا احتـوى الشعاع args على أقل من ثلاثة عناصر، سيتحقق الشرط الذي يسـتدعي المـاكرو panic! وبالتـالي سـيتوقف البرنامج مباشرةً.

دعنا نجرّب تنفيذ البرنامج بعد إضافتنا للسطور البرمجية القليلـة هـذه في دالـة new دون أي وسـطاء ونـرى رسالة الخطأ:

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep`
thread 'main' panicked at 'not enough arguments', src/main.rs:26:13
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

الخرج هذا أفضل لأنه يدلّنا على الخطأ بوضوح أكبر، إلا أننا نحصل على معلومات زائدة لسنا بحاجـة لعرضـها للمستخدم في ذات الوقت. لعلّ هذه الطريقـة ليسـت بالطريقـة المثلى؛ إذ أن اسـتدعاء panic! ملائم لعـرض الخطأ في مرحلة كتابة الشيفرة البرمجية للمبرمج وليس في مرحلة استخدام البرنامج للمستخدم (كما ناقشـنا في

فصول سابقة)، نستخدم بدلًا من ذلك طريقة أخرى تعلمناها سابقًا ألا وهي إعادة النوع `Result` الذي يمثّل قيمة نجاح أو فشل.

ب. إعادة النوع Result بدلا من استدعاء !panic

يمكننا إعادة قيمة `Result` التي تحتوي على نسخة من `Config` في حال النجاح وقيمة تصف الخطأ في حالة الفشل. سنغيّر أيضًا اسم الدالة `new` إلى `build`، إذ سيتوقع العديد من المبرمجين أن الدالة `new` لن تفشل أبدًا. يمكننا استخدام النوع `Result` للإشارة إلى مشكلة عندما تتواصل الدالة `Config::build` مع الدالة `main`، ومن ثم يمكننا التعديل على `main` بحيث تحوّل المتغاير `Err` إلى خطأ أوضح للمستخدم دون النص الذي يتضمن `thread 'main'` و `RUST_BACKTRACE` وهو ما ينتج عن استدعاء `panic!`.

توضح الشيفرة 9 التغييرات التي أجريناها لقيمة الدالة المُعادة -التي تحمل الاسم `Config::build` الآن- ومتن الدالة الذي يُعيد قيمة `Result`. لاحظ أن هذه الشيفرة لن تُصرَّف حتى نعدل الدالة `main` أيضًا، وهو ما سنفعله في الشيفرة التي تليها.

اسم الملف: src/main.rs

```
impl Config {
    fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        Ok(Config { query, file_path })
    }
}
```

[الشيفرة 9: إعادة قيمة `Result` من الدالة `Config::build`]

تُعيد الدالة `build` قيمة `Result` بنسخة `Config` في حال النجاح و `&'static str` في حال الفشل، وستكون قيم الأخطاء سلاسل نصية مجرّدة string literals دومًا بدورة حياة `'static`.

أجرينا تعديلين على محتوى الدالة، فبدلًا من استدعاء `panic!` عندما لا يمرّر المستخدم عددًا كافيًا من الوسطاء، أصبحنا نُعيد قيمة `Err`، وغلّفنا قيمة `Config` المُعادة بمتغاير `Ok`. تجعل هذه التغييرات من الدالة متوافقة مع نوع بصمتها الجديد.

تسمح إعادة القيمة `Err` من `Config::build` إلى الدالة `main` بالتعامل مع القيمة `Result` المُعادة من الدالة `build` ومغادرة البرنامج بصورةٍ أفضل في حالة الفشل.

ج. استدعاء الدالة Config::build والتعامل مع الأخطاء

نحتاج لتعديل الدالة `main` بحيث تتعامل مع النوع `Result` المُعاد إليها من الدالة `Config::build` كي نتعامل مع الأخطاء عند حدوثها وطباعة رسالة مفهومة للمستخدم، وهذا التعديل موضّح في الشيفرة 10. كما أننا سنعدّل البرنامج بحيث نخرج من أداة سطر الأوامر برمز خطأ غير صفري عند استدعاء `panic!` إلا أننا سنطبق ذلك يدويًا. رمز الخطأ غير الصفري nonzero exit code هو اصطلاح للإشارة إلى العملية التي استدعت البرنامج الذي تسبب بالخروج من برنامجنا برمز الخطأ.

اسم الملف: src/main.rs

```
use std::process;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    // --snip--
```

[الشيفرة 10: الخروج برمز خطأ إذا فشل بناء `Config`]

استخدمنا في الشيفرة السابقة تابعًا لم نشرحه بالتفصيل بعد، ألا وهو `unwrap_or_else` وهو تابع معرَّف في `Result<T, E>` في المكتبة القياسية. يسمح لنا استخدام التابع `unwrap_or_else` بتعريف بعض طرق التعامل مع الأخطاء المخصصة التي لا تستخدم الماكرو `panic!`. سلوك التابع مماثل للتابع `unwrap` إذا كانت قيمة `Result` هي `Ok`، إذ أنه يعيد القيمة المغلّفة داخل `Ok`، إلا أن التابع يستدعي شيفرةً برمجيةً في مغلِّفه closure إذا كانت القيمة `Err` وهي دالة مجهولة anonymous function نعرّفها ونمرّرها بمثابة وسيط للتابع `unwrap_or_else`.

سنتحدث عن المُغلِّفات بالتفصيل لاحقًا، وكل ما عليك معرفته حاليًا هو أن `unwrap_or_else` ستمرّر القيمة الداخلية للقيمة `Err` -وهي في هذه الحالة السلسلة النصية "not enough arguments" التي

أضفناها في الشيفرة 9- إلى مغلّفنا ضمن الوسيط `err` الموجود بين الخطين الشاقوليين (`|`)، ومن ثم تستطيع الشيفرة البرمجية الموجودة في المغلّف استخدام القيمة الموجودة في `err` عند تنفيذها.

أضفنا سطر `use` جديد لإضافة `process` من المكتبة القياسية إلى النطاق. تُنفَّذ الشيفرة البرمجية الموجودة في المغلّف في حالة الخطأ ضمن سطرين فقط: نطبع قيمة `err` ومن ثم نستدعي `process::exit`. توقف الدالة `process::exit` البرنامج مباشرةً وتُعيد العدد المُمرّر إليها بمثابة رمز حالة خروج. تشابه العملية استخدام `panic!` في الشيفرة 8، إلا أننا لا نحصل على الخرج الإضافي بعد الآن. دعنا نجرّب الأمر:

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/minigrep`
Problem parsing arguments: not enough arguments
```

عظيم، فالرسالة التي نحصل عليها الآن مفهومةً أكثر للمستخدمين.

12.3.5 استخراج المنطق من الدالة main

الآن وبعد انتهائنا من إعادة بناء التعليمات البرمجية الخاصة بالحصول على الوسطاء، دعنا ننتقل إلى منطق البرنامج، إذ علينا أن نستخرج المنطق إلى دالةٍ نسميها `run` كما ذكرنا سابقًا، بحيث تحتوي على الشيفرة البرمجية الموجودة في `main` حاليًا مع استثناء الشيفرة البرمجية المخصصة لضبط البرنامج، أو التعامل مع الأخطاء، وبحلول نهاية المهمة هذه يجب أن تكون الدالة `main` سهلة الفحص والقراءة ومختصرة، وسنكون قادرين على كتابة الاختبارات لجزء المنطق من البرنامج بصورةٍ منفصلة.

توضح الشيفرة 11 الدالة `run` المُستخلصة، وسنبدأ بإجراء تحسينات صغيرة وتدريجية حاليًا لاستخلاص المنطق إلى الدالة. نبدأ بتعريف الدالة في src/main.rs.

اسم الملف: src/main.rs

```
fn main() {
    // --snip--

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    run(config);
}
```

```
fn run(config: Config) {
    let contents = fs::read_to_string(config.file_path)
        .expect("Should have been able to read the file");

    println!("With text:\n{contents}");
}

// --snip--
```

[الشيفرة 11: استخراج الدالة run التي تحتوي على بقية منطق البرنامج]

تحتوي الدالة run الآن على جميع المنطق المتبقي في main بدءًا من قراءة الملف، وتأخذ الدالة run نسخةً من الهيكل Config مثل وسيط.

ا. إعادة الأخطاء من الدالة run

أصبح بإمكاننا -بعد فصل منطق البرنامج في الدالة run- تحسين التعامل مع الأخطاء كما فعلنا بالدالة Config::build في الشيفرة 9. تُعيد الدالة run القيمة Result<T, E> بعد حصول خطأ بدلًا من السماح للبرنامج بالهلع باستدعاء expect، وسيسمح لنا ذلك بدعم المنطق الخاص بالتعامل مع الأخطاء في main بطريقة سهلة الاستخدام. توضح الشيفرة 12 التغييرات الواجب إجرائها ومحتوى الدالة run.

اسم الدالة: src/main.rs

```
use std::error::Error;

// --snip--

fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    println!("With text:\n{contents}");

    Ok(())
}
```

[الشيفرة 12: تعديل الدالة run بحيث تعيد القيمة Result]

أجرينا ثلاثة تعديلات هنا؛ إذ عدّلنا أولًا القيمة المُعادة من الدالة `run` إلى النوع `Result<(), Box<dyn Error>>`، فقد أعادت هذه الدالة سابقًا نوع الوحدة unit type `()`، إلا أننا نُبقي هذا النوع مثل قيمة مُعادة في حالة `Ok`.

استخدمنا كائن السمة `Box<dyn Error>` لنوع الخطأ (وقد أضفنا `std::error::Error` إلى النطاق باستخدام التعليمة `use` في الأعلى). سنغطّي كائنات السمة لاحقًا، ويكفي الآن معرفتك أن `Box<dyn Error>` تعني أن الدالة ستُعيد نوعًا يطبّق السمة `Error`، إلا أن تحديد نوع القيمة المُعادة ليس ضروريًا. يمنحنا ذلك مرونة إعادة قيم الخطأ التي قد تكون من أنواع مختلفة في حالات فشل مختلفة، والكلمة المفتاحية `dyn` هي اختصار للكلمة "ديناميكي dynamic".

يتمثّل التعديل الثاني بإزالة استدعاء `expect` واستبداله بالعامل `?`، الذي شرحناه سابقًا هنا؛ فبدلًا من استخدام الماكرو `panic!` على الخطأ، يُعيد العامل `?` قيمة الخطأ من الدالة الحالية للشيفرة البرمجية المستدعية لكي تتعامل معه.

ثالثًا، تُعيد الدالة `run` الآن قيمة `Ok` في حالة النجاح. صرّحنا عن نوع نجاح الدالة `run` في بصمتها على النحو التالي: `()`، مما يعني أننا بحاجة تغليف قيمة نوع الوحدة في قيمة `Ok`. قد تبدو طريقة الكتابة `Ok(())` هذه غريبة قليلًا، إلا أن استخدام `()` بهذا الشكل هو طريقة اصطلاحية للإشارة إلى أننا نستدعي `run` من أجل تأثيرها الجانبي فقط، إذ أنها لا تُعيد قيمةً نستخدمها.

ستُصرَّف الشيفرة البرمجية السابقة عند تنفيذها، إلا أننا سنحصل على التحذير التالي:

```
$ cargo run the poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
warning: unused `Result` that must be used
  --> src/main.rs:19:5
   |
|     run(config);
   |     ^^^^^^^^^^^^
   |
   = note: `#[warn(unused_must_use)]` on by default
   = note: this `Result` may be an `Err` variant, which should be
handled

warning: `minigrep` (bin "minigrep") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.71s
     Running `target/debug/minigrep the poem.txt`
Searching for the
```

```
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

تخبرنا رست أن شيفرتنا البرمجية تتجاهل قيمة `Result`، وأن قيمة `Result` قد تشير إلى حصول خطأ ما، إلا أننا لا نتحقق من حدوث خطأ، ويذكّرنا المصرّف بأنه من الأفضل لنا كتابة شيفرة برمجية للتعامل مع الأخطاء هنا. دعنا نصحّح هذه المشكلة الآن.

ب. التعامل مع الأخطاء المُعادة من الدالة run في الدالة main

سنتحقق من الأخطاء ونتعامل معها باستخدام طرق مماثلة للطرق التي استخدمناها مع `Config::build` في الشيفرة 10، مع اختلاف بسيط:

اسم الملف: src/main.rs

```rust
fn main() {
    // --snip--

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    if let Err(e) = run(config) {
        println!("Application error: {e}");
        process::exit(1);
    }
}
```

نستخدم `if let` بدلًا من `unwrap_or_else` للتحقق فيما إذا كانت الدالة `run` تُعيد قيمة `Err` ونستدعي `process::exit(1)` إذا كانت هذه الحالة محققة. لا تُعيد الدالة `run` قيمة نحتاج لفك التغليف عنها unwrap باستخدام `unwrap` بالطريقة ذاتها التي تُعيد فيها الدالة `Config::build` نسخةً من `Config`.

نهتم فقط بحالة حصول خطأ والتعرف عليه لأن `run` تُعيد `()` في حالة النجاح، لذا لا نحتاج من `unwrap_or_else` أن تُعيد القيمة المفكوك تغليفها، والتي هي ببساطة `()`.

محتوى تعليمات `if let` ودوال `unwrap_or_else` مماثلة في الحالتين: إذ نطبع الخطأ، ثم نغادر البرنامج.

12.3.6 تجزئة الشيفرة البرمجية إلى وحدة مكتبة مصرفة

يبدو مشروع `minigrep` جيّدًا حتى اللحظة. سنجزّء الآن ملف src/main.rs ونضع جزءًا من الشيفرة البرمجية في ملف "src/lib.rs"، إذ يمكننا بهذه الطريقة اختبار الشيفرة البرمجية مع ملف src/main.rs لا ينجز العديد من المهام.

دعنا ننقل الشيفرة البرمجية غير الموجودة في الدالة `main` من الملف src/main.rs إلى الملف src/lib.rs، والتي تتضمن:

- تعريف الدالة `run`.
- تعليمات `use` المرتبطة بالشيفرة البرمجية التي سننقلها.
- تعريف الهيكل `Config`.
- تعريف دالة `Config::build`.

يجب أن يحتوي الملف src/lib.rs على البصمات الموضحة في الشيفرة 13 (أهملنا محتوى الدوال لاختصار طول الشيفرة). لاحظ أن الشيفرة البرمجية لا تُصرَّف حتى نعدّل الملف src/main.rs وهو ما نفعله في الشيفرة 14.

اسم الملف: src/lib.rs

```
use std::error::Error;
use std::fs;

pub struct Config {
    pub query: String,
    pub file_path: String,
}

impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        // --snip--
```

```
    }
}

pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    // --snip--
}
```

[الشيفرة 13: نقل `Config` و `run` إلى src/lib.rs]

استخدمنا الكلمة المفتاحية `pub` هنا بحرية في كل من الهيكل `Config` وحقوله وتابعه `build` وعلى الدالة `run`. أصبح لدينا وحدة مكتبة مصرَّفة library crate بواجهة برمجية عامة public API يمكننا استخدامها.

نحتاج إضافة الشيفرة البرمجية التي نقلناها إلى الملف src/lib.rs إلى نطاق الوحدة الثنائية المصرفة binary crate في الملف src/main.rs كما هو موضح في الشيفرة 14:

اسم الملف: src/main.rs

```
use std::env;
use std::process;

use minigrep::Config;

fn main() {
    // --snip--
    if let Err(e) = minigrep::run(config) {
        // --snip--
    }
}
```

[الشيفرة 14: استخدام وحدة المكتبة المصرفة `minigrep` في src/main.rs]

أضفنا السطر `use minigrep::Config` لإضافة النوع `Config` من وحدة المكتبة المصرّفة إلى نطاق الوحدة الثنائية المصرّفة، وأسبقنا prefix الدالة `run` باسم الوحدة المصرفة crate. يجب أن تكون وظائف البرنامج مترابطة مع بعضها بعضًا الآن، وأن تعمل بنجاح، لذا نفّذ البرنامج باستخدام `cargo run` وتأكد من أن كل شيء يعمل كما هو مطلوب.

أخيرًا، كان هذا عملًا شاقًا، إلا أننا بدأنا بأساس يضمن لنا النجاح في المستقبل، إذ أصبح التعامل مع الأخطاء الآن سهلًا وقد جعلنا من شيفرتنا البرمجية أكثر معيارية. سنعمل في الملف src/lib.rs بصورةٍ أساسية من الآن وصاعدًا.

دعنا نستفيد من المعيارية الجديدة في برنامجنا بتحقيق شيءٍ كان من الممكن أن يكون صعب التحقيق في الشيفرة البرمجية القادمة، إلا أنه أصبح أسهل بالشيفرة البرمجية الجديدة، ألا وهو كتابة الاختبارات.

12.4 اختبار البرنامج

بدأنا عملية برمجة أداة سطر الأوامر المشابهة لأداة `grep` التي تبحث داخل ملف معيّن عن سلسلة نصية محددة وتضع أساس برنامج سطر الأوامر حيث برمجنا منطق التعامل مع الوسطاء المرّرة في سطر الأوامر، ثم حسناه وطورناه أكثر حيث عملنا على تجزئة برنامجنا إلى وحدات منفصلة لتسهل عملية اختبار البرنامج، ونتطرّق في هذا القسم إلى اختبار البرنامج.

12.4.1 تطوير عمل المكتبة باستخدام التطوير المقاد بالاختبار test-driven

الآن، وبعد استخراجنا لمعظم منطق البرنامج إلى الملف src/lib.rs، يبقى لدينا منطق الحصول على الوسطاء والتعامل مع الأخطاء في src/main.rs، ومن الأسهل كتابة الاختبارات في هذه الحالة، إذ ستركّز الاختبارات على منطق شيفرتنا البرمجية الأساسية. يمكننا استدعاء الدوال مباشرةً باستخدام مختلف الوسطاء arguments والتحقق من القيمة المعادة دون الحاجة لاستدعاء ملفنا التنفيذي من سطر الأوامر.

نُضيف في هذا القسم منطق البحث إلى البرنامج "minigrep" باستخدام التطوير المُقاد بالاختبار test-driven development -أو اختصارًا TDD- باتباع الخطوات التالية:

1. كتابة اختبار يفشل وتنفيذه للتأكد من أنه يفشل فعلًا للسبب الذي تتوقعه.
2. كتابة شيفرة برمجية أو التعديل على شيفرة برمجية موجودة مسبقًا لجعل الاختبار الجديد ينجح.
3. إعادة بناء التعليمات البرمجية المُضافة أو المُعدّلة للتأكد من أن الاختبارات ستنجح دومًا.
4. كرّر الأمر مجدّدًا بدءًا من الخطوة 1.

على الرغم من أن التطوير المُقاد بالاختبار هو طريقة من الطرق العديدة الموجودة لكتابة البرمجيات، إلا أنه من الممكن أن يساهم بتصميم الشيفرة البرمجية، إذ تساعد كتابة الاختبارات قبل كتابة الشيفرة البرمجية التي تجعل من الاختبار ناجحًا في المحافظة على اختبار جميع أجزاء الشيفرة البرمجية بسوّية عالية خلال عملية التطوير.

سنختبر تطبيق الخاصية التي ستبحث عن السلسلة النصية المُدخلة في محتويات الملف وتعطينا قائمةً من الأسطر تحتوي على حالات التطابق، ثمّ سنضيف هذه الخاصية في دالة تدعى `search`.

ا. كتابة اختبار فاشل

دعنا نتخلص من تعليمات `println!` من الملفين "src/lib.rs" و "src/main.rs" التي كنا نستخدمها لتفقُّد سلوك البرنامج ولن نحتاج إليها بعد الآن، ثم نضيف الوحدة `tests` في الملف "src/lib.rs" مع دالة اختبار كما فعلنا في قسم سابق. تحدد دالة الاختبار السلوك الذي نريده من الدالة `search` ألا وهو: ستأخذ الدالة سلسلةً نصيةً محددةً ونصًا تبحث فيه وستُعيد السطور التي تحتوي على تطابق. توضح الشيفرة 15 هذا الاختبار، إلا أنها لن تُصرَّف بنجاح بعد.

اسم الملف: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn one_result() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(vec!["safe, fast, productive."], search(query,
contents));
    }
}
```

[الشيفرة 15: إنشاء اختبار فاشل للدالة `search` التي كنا نتمنى الحصول عليها]

يبحث هذا الاختبار عن السلسلة النصية `"duct"`، إذ يتألف النص الذي نبحث فيه من ثلاثة أسطر ويحتوي واحدٌ منها فقط السلسلة النصية `"duct"` (يخبر الخط المائل العكسي backslash بعد علامتي التنصيص المزدوجتين رست بعدم إضافة محرف سطر جديد في بداية محتوى السلسلة النصية المجردة). نتأكد أن القيمة المُعادة من الدالة `search` تحتوي فقط على السطر الذي نتوقعه.

لا يمكننا تنفيذ هذا الاختبار بعد ورؤيته يفشل لأن الاختبار لا يُصرَّف، والسبب في ذلك هو أن الدالة `search` غير موجودة بعد. وفقًا لمبادئ التطوير المُقاد بالاختبار، علينا إضافة القليل من الشيفرة البرمجية بحيث يمكننا تصريف الاختبار وتنفيذه بإضافة تعريف الدالة `search` التي تُعيد شعاعًا vector فارغًا دومًا كما

هو موضح في الشيفرة 16، ومن ثم يجب أن يُصرَّف الاختبار ويفشل لعدم مطابقة الشعاع الفارغ للشعاع الـذي يحتوي السطر `".safe, fast, productive"`.

اسم الملف: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}
```

[الشيفرة 16: تعريف الدالة `search` باستخدام شيفرة برمجية قصيرة بحيث يُصرَّف الاختبار]

لاحظ أننا بحاجة إلى تعريف دورة حياة lifetime صراحةً تدعى `'a` في بصمة الدالة `search` واستخدام دورة الحياة في الوسيط `contents` والقيمة المُعادة. تذكر أننا ذكرنا في قسم سابق أن معاملات دورة الحياة تحدد أي دورات حياة الوسطاء متصلة بدورة حياة القيمة المُعادة، وفي هذه الحالة فإننا نحدد أن الشـعاع المُعـاد يجب أن يحتوي على شرائح سلسلة نصية string slices تمثـل مرجعًـا لشـرائح الوسـيط `contents` بـدلًا من الوسـيط `query`.

بكلمات أخرى، نخبر رست بأن البيانات المُعادة من الدالة `search` ستعيش طالما تعيش البيانات المُمـرّرة إلى الدالة `search` في الوسيط `contents`. يجب أن تكون الشرائح المُستخدمة مثل مراجـع للبيانـات صـالحة حتى يكون المرجع صالحًا، إذ سيكون التحقق من الأمان خاطئًا لو افترض المصـرف أننـا نُنشـئ شـرائح سلاسـل نصية من `query` بدلًا من `contents`.

نحصل على الخطأ التالي إذا نسينا توصيف دورات الحياة وجرّبنا تصريف هذه الدالة:

```
$ cargo build
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
error[E0106]: missing lifetime specifier
  --> src/lib.rs:28:51
   |
| pub fn search(query: &str, contents: &str) -> Vec<&str> {
   |                      ----           ----         ^ expected
named lifetime parameter
   |
   = help: this function's return type contains a borrowed value, but
the signature does not say whether it is borrowed from `query` or
`contents`
help: consider introducing a named lifetime parameter
   |
```

```
| pub fn search<'a>(query: &'a str, contents: &'a str) -> Vec<&'a str>
{
   |              ++++         ++                 ++              ++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `minigrep` due to previous error
```

لا يمكن لرست معرفة أي الوسيطين نحتاج، لذا يجب أن نصرح عن ذلك مباشـرةً. نعلم أن `contents` هـو الوسيط الذي يجب أن يُربط مع القيمة المُعادة باستخدام دورة الحياة وذلك لأنه الوسيط الذي يحتوي على كامـل محتوى الملف النصي الذي نريد أن نعيد أجزاءً متطابقةً منه.

لا تتطلب لغات البرمجة الأخرى ربط الوسطاء للقيمة المعادة في بصمة الدالة، إلا أنك ستعتاد على ذلك مع الممارسة. ننصحك بمقارنة هذا المثال مع مثال موجود في الفصل 10.

دعنا ننفذ الاختبار:

```
$ cargo test
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished test [unoptimized + debuginfo] target(s) in 0.97s
     Running unittests src/lib.rs (target/debug/deps/minigrep-
9cd200e5fac0fc94)

running 1 test
test tests::one_result ... FAILED

failures:

---- tests::one_result stdout ----
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `["safe, fast, productive."]`,
 right: `[]`', src/lib.rs:44:9
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace



failures:
    tests::one_result
```

```
test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0
filtered out; finished in 0.00s


error: test failed, to rerun pass `--lib`
```

عظيم، فشل الاختبار كما توقعنا. دعنا نجعل الاختبار ينجح الآن.

ب. كتابة شيفرة برمجية لاجتياز الاختبار

يفشل اختبارنا حاليًا لأننا نُعيد دائمًا شعاعًا فارغًا، وعلى برنامجنا اتباع الخطوات التالية لتصحيح ذلك وتطبيق `search`:

- المرور على سطور محتوى الملف.
- التحقق ما إذا كان السطر يحتوي على السلسلة النصية التي نبحث عنها.
- إذا كان هذا الأمر محققًا: نضيف السطر إلى قائمة القيم التي سنعيدها.
- إن لم يكن محققًا: لا نفعل أي شيء.
- نُعيد قائمة الأسطر التي تحتوي على تطابق مع السلسلة النصية التي نبحث عنها.

لنعمل على كل خطوة بالتدريج، بدءًا من المرور على الأسطر على الترتيب.

المرور على الأسطر باستخدام التابع lines

توفر لنا رست تابعًا مفيدًا للتعامل مع السلاسل النصية سطرًا تلو الآخر بصورةٍ سهلة وهو تابع `lines`، ويعمل التابع بالشكل الموضح في الشيفرة 17. انتبه إلى أن الشيفرة 17 لن تُصرَّف بنجاح.

اسم الملف: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        // line أجرِ بعض العمليات على
    }
}
```

[الشيفرة 17: المرور على أسطر الوسيط `contents`]

يُعيد التابع `lines` مكرّرًا iterator، وسنتحدث عن المكررات فيما بعد؛ تذكّر أنك رأيت هذه الطريقة باستعمال المكررات في الشيفرة 5 من الفصل 3 عندما استخدمنا حلقة `for` مع مكرر لتنفيذ شيفرة برمجية على كل عنصر من عناصر التجميعة collection.

البحث عن الاستعلام في كل سطر

الآن نبحث فيما إذا كان السطر يحتوي على السلسلة النصية المحددة بالاستعلام، وتحتوي السلاسل النصية لحسن الحظ على تابع مفيد يدعى `contains` يفعل هذا نيابةً عنّا. أضف استدعاءً للتابع `contains` في الدالـة `search` كما هو موضح في الشيفرة 18. لاحظ أن هذه الشيفرة البرمجية لن تُصرَّف بعد.

اسم الملف: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        if line.contains(query) {
            // do something with line
        }
    }
}
```

[الشيفرة 18: إضافة ميزة البحث عن السلسلة النصية الموجودة في الوسيط `query` داخل السطر]

بدأنا ببناء وظيفة البرنامج الأساسية الآن، ولتصريف البرنامج بنجاح نحن بحاجة لإعادة قيمـة من متن الدالـة كما قلنا أننا سنفعل في بصمة الدالة.

تخزين الأسطر المطابقة

أخيرًا يجب علينا استخدام طريقة لتخزين الأسطر المُطابقة التي نريد أن نُعيدها من الدالة، ونسـتخدم لـذلك شعاعًا متغيّرًا mutable vector قبل الحلقة `for` ونستدعي التابع `push` لتخزين `line` في الشعاع، ونُعيد هذا الشعاع بعد انتهاء الحلقة `for` كما هو موضح في الشيفرة 19.

اسم الملف: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
```

```
}
```

[الشيفرة 19: تخزين الأسطر المتطابقة بحيث نستطيع إعادتهم من الدالة]

الآن يجب أن تُعيد الدالة `search` فقط الأسطر التي تحتـوي على الوسـيط `query` ممـا يعـني أن اختبارنـا سينجح. دعنا ننفّذ الاختبار:

```
$ cargo test
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished test [unoptimized + debuginfo] target(s) in 1.22s
     Running unittests src/lib.rs (target/debug/deps/minigrep-
9cd200e5fac0fc94)

running 1 test
test tests::one_result ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

     Running unittests src/main.rs (target/debug/deps/minigrep-
9cd200e5fac0fc94)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

   Doc-tests minigrep

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

نجح الاختبار، لذا فالدالة تعمل.

يجب أن نفكّر بفرص إعادة بناء التعليمات البرمجية المحتملة بحلول هذه النقطة داخل الدالـة `search` مـع المحافظة على نجـاح الاختبـار للحفـاظ على الهـدف من الدالـة. ليسـت الشـيفرة البرمجيـة الموجـودة في الدالـة

search سيئة، لكنها لا تستغلّ خصائص المكرّرات المفيدة، وسنعود لهذه الشيفرة البرمجية في فصول لاحقة عندما نتحدث عن المكررات بتعمق أكبر لمعرفة التحسينات الممكنة.

استخدام الدالة search في الدالة run

الآن وبعد عمل الدالة search وتجربتها بنجاح، نحتاج إلى استدعاء search من الدالة run، وتمرير قيمة config.query و contents التي تقرأهما run من الملف إلى الدالة search. تطبع الدالة run كل سطر مُعاد من الدالة search:

اسم الملف: src/lib.rs

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    for line in search(&config.query, &contents) {
        println!("{line}");
    }

    Ok(())
}
```

ما زلنا نستخدم الحلقة for لإعادة كل سطر من search وطباعته.

يجب أن يعمل كامل البرنامج الآن. دعنا نجرّبه أولًا بكلمة "الضفدع frog" التي ينبغي أن تُعيد سطرًا واحدًا بالتحديد من قصيدة ايميلي ديكنسون Emily Dickinson:

```
$ cargo run -- frog poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.38s
     Running `target/debug/minigrep frog poem.txt`
How public, like a frog
```

عظيم. دعنا نجرب كلمةً يُفترض أنها موجودة في عدة أسطر مثل "body":

```
$ cargo run -- body poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep body poem.txt`
I'm nobody! Who are you?
```

```
Are you nobody, too?
How dreary to be somebody!
```

وأخيرًا، دعنا نتأكد من أننا لن نحصل على أي سطر عندما تكون الكلمة غير موجودة ضمن أي سطر في القصيدة مثل الكلمة "monomorphization":

```
$ cargo run -- monomorphization poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep monomorphization poem.txt`
```

ممتاز، بنينا إصدارنا الخاص المصغّر من أداة البحث الكلاسيكية `grep`، وتعلمنا الكثير عن هيكلة التطبيقات، كما أننا تعلمنا بعض الأشياء بخصوص دخل وخرج الملفات ودورات الحياة والاختبار والحصول على الوسطاء من سطر الأوامر.

سنوضّح في الفصل التالي كيفية العمل مع متغيرات البيئة في ختام هذا المشروع، إضافةً لكيفية طباعة الأخطاء إلى مجرى الأخطاء القياسي، وهما أمران مهمّان في برامج سطر الأوامر.

12.5 التعامل مع متغيرات البيئة وطباعة الأخطاء

بدأنا عملية برمجة أداة سطر الأوامر المشابهة لأداة `grep` التي تبحث داخل ملف معيّن عن سلسلة نصية محددة في القسم 12.1 وضع أساس برنامج سطر الأوامر بلغة رست حيث برمجنا منطق التعامل مع الوسطاء المرّرة في سطر الأوامر، ثم حسناه وطورناه أكثر في القسم التالي 12.3 حيث عملنا على تجزئة برنامجنا إلى وحدات منفصلة لتسهل عملية اختبار البرنامج، ثم اختبرنا البرنامج عبر كتابة اختبارات له، وسنوضّح أخيرًا في هذا القسم كيفية العمل مع متغيرات البيئة environment variables، إضافةً لكيفية طباعة الأخطاء إلى مجرى الأخطاء القياسي، وهما أمران مهمّان في برامج سطر الأوامر.

12.5.1 التعامل مع متغيرات البيئة

سنحسّن على برنامج "minigrep" باستخدام ميزة إضافية، ألا وهي خيار استخدام البحث بنمط عدم حساسية حالة الحروف case-insensitive (سواءٌ كانت أحرف صغيرة أو كبيرة) بحيث يمكن للمستخدم تفعيل هذا النمط أو تعطيله باستخدام متغيرات البيئة. يمكننا جعل هذه الميزة خيارًا لسطر الأوامر إلا أن المستخدم سيكون بحاجةٍ لكتابة هذا الخيار في سطر الأوامر في كل مرة يريد استخدام البرنامج، وباستخدام متغيرات البيئة نجعل ضبط هذا الخيار لمرة واحدة بحيث تكون كل عمليات البحث غير حساسة لحالة الأحرف في جلسة الطرفية تلك.

ا. كتابة اختبار يفشل لدالة search لميزة عدم حساسية حالة الأحرف

نُضيف أولًا دالة `search_case_insensitive` الـتي تُسـتدعى عنـدما يكـون لمتغـير البيئـة قيمـةً مـا، وسنستمر باتباع عملية التطوير المُقاد بالاختبار هنا، وبالتالي ستكون الخطوة الأولى هي كتابـة اختبـار يفشـل؛ إذ سنضيف اختبارًا جديدًا للدالة الجديدة `search_case_insensitive` وسـنعيد تسـمية الاختبـار القـديم من اسـمه السـابق `one_result` إلى `case_sensitive` لتوضـيح الفـرق بين الاختبـارين كمـا هـو موضـح في الشيفرة 20.

اسم الملف: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn case_sensitive() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Duct tape.";

        assert_eq!(vec!["safe, fast, productive."], search(query,
contents));
    }

    #[test]
    fn case_insensitive() {
        let query = "rUsT";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Trust me.";

        assert_eq!(
```

```
            vec!["Rust:", "Trust me."],
            search_case_insensitive(query, contents)
        );
    }
}
```

[الشيفرة 20: إضافة اختبار جديد يفشل لدالة عدم حساسية حالة الحرف التي سنضيفها لاحقًا]

لاحظ أننا أضفنا اختبار `contents` القديم أيضًا، وأضفنا سطرًا جديدًا للنص "Duct tape." باستخدام حرف D كبير، والذي يجب ألا يطابق استعلام السلسلة النصية "duct" عندما نبحث في حالة حساسية حالة الأحرف. يساعد تغيير الاختبار القديم بهذه الطريقة في التأكد من أننا لن نعطّل خاصية البحث في حالة حساسية الأحرف (وهي الحالة التي طبقناها أولًا، والتي تعمل بنجاح للوقت الحالي). يجب أن ينجح هذا الاختبار الآن ويجب أن يستمر بالنجاح بينما نعمل على خاصية عدم حساسية حالة الأحرف.

يستخدم الاختبار الجديد للبحث بخاصية عدم حساسية حالة الأحرف "rUsT" مثل كلمة بحث، لذلك سنُضيف في الدالة `search_case_insensitive` الكلمة "rUsT" والتي يجب أن تطابق "Rust:" بحرف R كبير وأن تطابق السطر "Trust me." أيضًا رغم أن للنتيجتين حالة أحرف مختلفة عن الكلمة التي استخدمناها. هذا هو اختبارنا الذي سيفشل، وستفشل عملية تصريفه لأننا لم نعرّف بعد الدالة `search_case_insensitive`. ضِف هيكلًا للدالة بحيث تُعيد شعاعًا فارغًا بطريقة مشابهة لما فعلناه في دالة `search` في الشيفرة 16 حتى نستطيع تصريف الاختبار ورؤية أنه يفشل فعلًا.

ب. تنفيذ دالة search_case_insensitive

ستكون الدالة `search_case_insensitive` الموضحة في الشيفرة 21 مماثلة تقريبًا للدالة `search`، والفارق الوحيد هنا هو أننا سنحوّل حال الأحرف للكلمة التي نبحث عنها إلى أحرف صغيرة (الوسيط `query`) إضافةً إلى كل سطر `line`، بحيث تكون حالة الأحرف متماثلة عند المقارنة بينهما بغضّ النظر عن حالة الأحرف الأصلية.

اسم الملف: src/lib.rs

```
pub fn search_case_insensitive<'a>(
    query: &str,
    contents: &'a str,
) -> Vec<&'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();
```

```
    for line in contents.lines() {
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }

    results
}
```

[الشيفرة 21: تعريف الدالة `search_case_insensitive` بحيث تحوّل أحرف الكلمة التي نبحث عنها مع السطر إلى أحرف صغيرة قبل مقارنتهما]

نحوّل أولًا أحرف السلسلة النصية `query` إلى أحرف صغيرة ونخزّنها في متغير يحمل الاسم ذاته، ولتحقيق ذلك نستدعي `to_lowercase` على السلسلة النصية بحيث تكون النتيجة واحدة بغضّ النظر عن حالة الأحرف المدخلة `"rust"` أو `"RUST"` أو `"Rust"` أو `"rUsT"` وسنعامل السلسلة النصية المدخلة وكأنها `"rust"` لإهمال حالة الأحرف، سيعمل التابع `to_lowercase` على محارف يونيكود Unicode الأساسية إلا أن عمله لن يكون صحيحًا مئة بالمئة. إن كنّا نبرمج تطبيقًا واقعيًا فعلينا أن نقوم بالمزيد من العمل بخصوص هذه النقطة، إلا أننا نناقش في هذا القسم متغيرات البيئة وليس يونيكود، لذا لن نتطرق لذلك الآن.

لاحظ أن `query` من النوع `String` الآن وليس شريحة سلسلة نصية لأن استدعاء التابع `to_lowercase` يُنشئ بيانات جديدة عوضًا عن استخدام مرجع للبيانات الموجودة مسبقًا. لنفرض بأن الكلمة هي `"rUsT"` كمثال: لا تحتوي شريحة السلسلة النصية على حرف `u` أو `t` صغير لنستخدمه لذا علينا حجز مساحة جديدة لنوع `String` يحتوي على `"rust"`، وعندما نمرّر `query` كوسيط إلى التابع `contains` فنحن بحاجة لإضافة الرمز `&` لأن شارة `contains` معرفة بحيث تأخذ شريحة سلسلة نصية.

نضيف من ثمّ استدعاءً للتابع `to_lowercase` لكل `line` لتحويل أحرفه إلى أحرف صغيرة، وبذلك نكون حولنا كل من أحرف `line` و`query` إلى أحرف صغيرة وسنجد حالات التطابق بغض النظر عن حالة الأحرف في السلسلتين الأصليتين.

دعنا نرى إذا كان تطبيقنا سيجتاز الاختبار:

```
$ cargo test
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished test [unoptimized + debuginfo] target(s) in 1.33s
     Running unittests src/lib.rs (target/debug/deps/minigrep-
9cd200e5fac0fc94)
```

```
running 2 tests
test tests::case_insensitive ... ok
test tests::case_sensitive ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

     Running unittests src/main.rs (target/debug/deps/minigrep-
9cd200e5fac0fc94)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

   Doc-tests minigrep

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

عظيم، اجتزنـا الاختبـار. دعنـا نسـتدعي الآن الدالـة `search_case_insensitive` من الدالـة `run`، وسنُضيف أولًا خيار الضبط إلى الهيكل `Config` للتبديل بين البحث الحساس وغير الحساس لحالة الأحـرف، إلا أن إضافة هذا الحقل ستتسبب بأخطاء عند التصريف لأننا لم نسند هذا الحقل إلى أي مكان بعد:

اسم الملف: src/lib.rs

```
pub struct Config {
    pub query: String,
    pub file_path: String,
    pub ignore_case: bool,
}
```

أضفنا الحقل `igonre_case` الذي يخزن متحول بوليـاني Boolean، وسـنحتاج الدالـة `run` للتتحقـق من قيمـة `ignore_case` لتحديـد اسـتدعاء أيّ من دالـتي البحث: `search` أو `search_case_insensitive` كما هو موضح في الشيفرة 22. لن تُصرَّف هذه الشيفرة بنجاح بعد.

اسم الملف: src/lib.rs

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    let results = if config.ignore_case {
        search_case_insensitive(&config.query, &contents)
    } else {
        search(&config.query, &contents)
    };

    for line in results {
        println!("{line}");
    }

    Ok(())
}
```

[الشيفرة 22: استدعاء الدالة `search` أو الدالة `search_case_insensitive` بحسب القيمة الموجودة في `config.ignore_case`]

أخيرًا، نحن بحاجة إلى فحص متغير البيئة. الدوال الخاصة بالتعامل مع متغيرات البيئة موجـودة في الوحـدة `env` في المكتبة القياسية، لذا نضيف الوحدة إلى النطاق أعلى الملف src/lib.rs. نستخدم الدالة `var` من الوحـدة `env` لفحص القيمـة المضـبوطة في متغـير البيئـة ذو الاسـم `IGNORE_CASE` كمـا هـو موضـح في الشيفرة 23.

اسم الملف: src/lib.rs

```
use std::env;
// --snip--

impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();
```

```
        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
```

[الشيفرة 23: التحقق من القيمة المضبوطة في متغير البيئة ذو الاسم `IGNORE_CASE`]

نُنشئ هنا متغيرًا جديدًا يدعى `ignore_case` ونُسند قيمته باستدعاء الدالة `env::var` ونمرّر اسم متغير البيئة `IGNORE_CASE` إليها. تُعيد الدالة `env::var` قيمةً من النوع `Result` تحتوي على متغاير variant يدعى `Ok` يحتوي على قيمة متغير البيئة إذا كان متغير البيئة مضبوطًا إلى قيمة معينة وإلا فهي تعيد قيمة المتغاير `Err`.

نستخدم التابع `is_ok` على القيمة `Result` للتحقق فيما إذا كان متغير البيئة مضبوطًا إلى قيمة معينة أم لا؛ فإذا كان مضبوطًا إلي قيمة فهذا يعني أنه علينا استخدام البحث بتجاهل حالة الأحرف؛ وإذا لم يكن مضبوطًا إلى قيمة معينة، فهذا يعني أن `is_ok` ستُعيد القيمة `false` وسينفذ البرنامج الدالة التي تجري البحث الحساس لحالة الأحرف. لا نهتم بقيمة متغير البيئة بل نهتم فقط فيما إذا كانت موجودة أو لا، ولذلك فنحن نستخدم التابع `is_ok` بدلًا من استخدام `unwrap` أو `expect` أو أيًا من التوابع الأخرى التي استخدمناها مع `Result` سابقًا.

نمرر القيمة في المتغير `ignore_case` إلى نسخة `Config` بحيث يمكن للدالة `run` أن تقرأ هذه القيمة وتقرّر استدعاء الدالة `search_case_insensitive` أو `search` كما طبقنا سابقًا في الشيفرة 22.

دعنا نجرّب البرنامج، ولننفذ أولًا البرنامج دون ضبط متغير البيئة وباستخدام الكلمة `to`، التي يجب أن تمنحنا جميع نتائج المطابقة للكلمة "to" بأحرف صغيرة فقط:

```
$ cargo run -- to poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
```

يبدو أن البرنامج يعمل بنجاح. دعنا نجرّب الآن تنفيذ البرنـامج مـع ضـبط IGNORE_CASE إلى 1 باسـتخدام الكلمة ذاتها to.

```
$ IGNORE_CASE=1 cargo run -- to poem.txt
```

إذا كنت تستخدم PowerShell، فعليك ضبط متغير البيئة، ثم تنفيذ البرنامج على أنهما أمرين منفصلين:

```
PS> $Env:IGNORE_CASE=1; cargo run -- to poem.txt
```

سيجعل ذلك متغير البيئة IGNORE_CASE مستمرًا طـوال جلسـة الصـدفة shell. ويمكن إزالـة القيمـة عن طريق الأمر Remove-Item:

```
PS> Remove-Item Env:IGNORE_CASE
```

يجب أن نحصل على الأسطر التي تحتوي على الكلمة "to" بغض النظر عن حالة الأحرف:

```
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

عظيم، حصلنا على الكلمة "To" ضمن كلمات أخـرى. يمكن لبرنـامج minigrep الآن البحث عن الكلمـات بغض النظر عن حالة الأحرف عن طريق متغير بيئة، ويمكنـك الآن التحكم بخيـارات البرنـامج عن طريـق وسـطاء سطر الأوامر، أو عن طريق متغيرات البيئة.

تسمح بعض البرامج بوسطاء سطر الأوامر ومتغيرات البيئة في ذات الوقت للخيار نفسه، وفي هذه الحالات يقرّر البرنامج أسبقية أحـد الخيـارين (وسـيط سـطر الأوامـر أو متغـير البيئـة). تمـرّن بنفسـك عن طريـق التحكم بحساسية حالة الأحرف عن طريق وسيط سطر أوامر أو متغير بيئة في الوقت ذاته، وحدّد أسـبقية أحـد الخيـارين حسب تفضيلك إذا تعارض الخياران مع بعضهما.

تحتوي الوحدة std::env على العديد من الخصائص الأخرى المفيدة للتعامـل مـع متغـيرات البيئـة، اقـرأ توثيق الوحدة للاطّلاع على الخيارات المتاحة.

12.5.2 كتابة رسائل الخطأ إلى مجرى الخطأ القياسي بدلا من مجرى الخرج القياسي

نكتب رسائل الأخطاء حاليًا إلى الطرفية باستخدام الماكرو println!، وفي معظم الطرفيات هناك نـوعين من الخرج: خرج قياسي stdout للمعلومات العامة وخطأ قياسي stderr لرسائل الخطأ، ويساعد التمييز بين

النوعين المستخدمين بتوجيه خرج نجاح البرنامج إلى ملف مع المحافظة على ظهور رسائل الخطأ على شاشة الطرفية.

الماكرو `println!` قادرٌ فقط على الطباعة إلى الخرج القياسي، لذا علينا استخدام شيء مختلف لطباعة الأخطاء إلى مجرى الأخطاء القياسي standard error stream.

ا. التحقق من مكان كتابة الأخطاء

دعنا أولًا نلاحظ كيفية طباعة المحتوى في برنامج `minigrep` حاليًا إلى الخرج القياسي، متضمنًا ذلك رسائل الأخطاء التي نريد كتابتها إلى مجرى الأخطاء القياسي بدلًا من ذلك، وسنحقّق ذلك بإعادة توجيه مجرى الخرج القياسي إلى ملف والتسبب بخطأ عمدًا، بينما سنُبقي على مجرى الأخطاء القياسي ولن نعيد توجيهه، وبالتالي سيُعرض محتوى مجرى الأخطاء القياسي على الشاشة مباشرةً.

من المتوقع لبرامج سطر الأوامر أن ترسل رسائل الخطأ إلى مجرى الأخطاء القياسي بحيث يمكننا رؤية الأخطاء على الشاشة حتى لو كنّا نعيد توجيه مجرى الخرج القياسي إلى ملف ما. لا يسلك برنامجنا حاليًا سلوكًا جيدًا، إذ أننا على وشك رؤية أن رسائل الخطأ تُخزّن في الملف.

لتوضيح هذا السلوك سننفذ البرنامج باستخدام `>` ومسار الملف output.txt وهو الملف الذي نريد إعادة توجيه مجرى الخرج القياسية إليه. لن نمرّر أي وسطاء عند التنفيذ، وهو ما سيتسبب بخطأ:

```
$ cargo run > output.txt
```

يخبر الرمز `>` الصدفة بكتابة محتويات الخرج القياسي إلى الملف output.txt بدلًا من الشاشة. لم نحصل على أي رسالة خطأ على الرغم من توقعنا لها بالظهور على الشاشة مما يعني أن الرسالة قد كُتبت إلى الملف. إليك محتوى الملف output.txt:

```
Problem parsing arguments: not enough arguments
```

نعم، تُطبع رسالة الخطأ إلى الخرج القياسي كما توقعنا ومن الأفضل لنا طباعة رسائل الخطأ إلى مجرى الأخطاء القياسي بدلًا من ذلك بحيث يحتوي الملف على البيانات الناتجة عن التنفيذ الناجح، دعنا نحقّق ذلك.

ب. طباعة الأخطاء إلى مجرى الأخطاء القياسي

سنستخدم الشيفرة البرمجية في الشيفرة 24 لتعديل طريقة طباعة الأخطاء. لحسن الحظ، الشيفرة البرمجية المتعلقة بطباعة رسائل الخطأ موجودة في دالة واحدة ألا وهي `main` بفضل عملية إعادة بناء التعليمات البرمجية التي أنجزناها سابقًا. تقدّم لنا المكتبة القياسية الماكرو `eprintln!` الذي يطبع إلى مجرى الأخطاء القياسي، لذا دعنا نعدّل من السطرين الذين نستخدم فيهما الماكرو `println!` ونستخدم `eprintln!` بدلًا من ذلك.

اسم الملف: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    if let Err(e) = minigrep::run(config) {
        eprintln!("Application error: {e}");
        process::exit(1);
    }
}
```

[الشيفرة 24: كتابة رسائل الخطأ إلى مجرى الأخطاء القياسية بدلًا من مجرى الخرج القياسي باستخدام `eprintln!`]

دعنا ننفذ البرنامج مجددًا بالطريقة ذاتها دون وسطاء وبإعادة توجيه الخرج القياسي إلى ملف باستخدام `>`:

```
$ cargo run > output.txt
Problem parsing arguments: not enough arguments
```

نستطيع رؤية الخطأ على الشاشة الآن، ولا يحتوي الملف output.txt أي بيانات وهو السلوك الذي نتوقعه من برامج سطر الأوامر.

دعنا ننفذ البرنامج مجددًا باستخدام الوسطاء لتنفيذ البرنامج دون أخطاء، وبتوجيه الخرج القياسي إلى ملف أيضًا كما يلي:

```
$ cargo run -- to poem.txt > output.txt
```

لن نستطيع رؤية أي خرج على الطرفية، وسيحتوي الملف output.txt على نتائجنا:

اسم الملف: output.txt

```
Are you nobody, too?
How dreary to be somebody!
```

يوضح هذا الأمر أننا نستخدم مجرى الخرج القياسي للخرج في حالة النجاح، بينما نستخدم مجرى الأخطاء القياسي في حالة الفشل.

12.6 خاتمة

لخّص هذا الفصل بمشروعه الكثير من المفاهيم المهمة التي تعلمناها لحد اللحظة كما أننا تكلمنا عن كيفية إجراء عمليات الدخل والخرج في رست، وذلك باستخدام وسطاء سطر الأوامر والملفات ومتغيرات البيئة والماكرو `eprintln!` لطباعة الأخطاء، ويجب أن تكون الآن مستعدًا لكتابة تطبيقات سطر الأوامر المختلفة. يجب أن تبقى شيفرتك البرمجية منظمةً جيدًا بمساعدة المفاهيم التي تعلمتها في الفصول السابقة وأن تخزن البيانات بفعالية في هياكل بيانات مناسبة وأن تتعامل مع الأخطاء بصورةٍ مناسبة، إضافةً إلى إجراء الاختبارات.

سننظر في الفصول التالية إلى بعض مزايا رست التي تأثرت باللغات الوظيفية، ألا وهي المغلّفات closures والمكررات iterators.

دورة علوم الحاسوب

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

13. ميزات البرمجة الوظيفية: المكررات والمنغلقات

استُوحي تصميم رست من تصميم العديد من لغات البرمجة الأخرى والتقنيات، ولعلّ التأثير الأكثر وضــوحًا على اللغة هو تأثير البرمجة الوظيفية functional programming، إذ تحتوي البرمجـة بأسـلوب وظيفي غالبًـا على استخدام الدوال مثل قيم عبر تمريرها مثل وسطاء وإعادتها من دوال أخرى وإسنادها إلى متغيرات لتنفيذها لاحقًا، وهلمّ جرًّا.

لن نناقش في هذا الفصل ماهية البرمجة الوظيفية، بل سنناقش بعض مزايـا لغـة رسـت المشـابهة لمزايـا موجودة في الكثير من اللغات التي يُشار إليها بلغات برمجة وظيفية.

سنغطّي النقاط التالية:

- المغلّفات closures، وهي بنى construct مشابهة للدوال، يمكنك تخزينها في متغير.
- المكررات iterators، وهي طريقة لمعالجة سلسلة من العناصر.
- كيفية استخدام المغلفات والمكررات لتحسين مشروع الدخل والخرج الذي أنجزناه سابقًا.
- أداء المغلّفات والمكرّرات (تنبيه: إنهما أسر ع ممّا تعتقد).

تكلمنا عن بعض مزايا رست مثـل مطابقـة النمـط pattern matching والمعـدّدات enums وهي مزايـا مستوحاة من لغات ذو أسلوب وظيفي. سنخصّص هذا الفصل للمغلفات والمكررات لأنهما جـزء مهم من كتابـة شيفرات رست البرمجية بصورةٍ اصطلاحية وسريعة.

13.1 المغلفات closures

تمثّل المغلّفات في لغة رست دوالًا مجهولة anonymous functions يمكنك حفظها في متغير أو تمريرها مثل وسيط إلى دالة أخرى، ويمكنك إنشاء مغلف في مكان ما، ثم استدعاءه من مكان آخر ليُقَّذ بحسب سياق المكان، وعلى عكس الدوال فالمغلفات يمكنها الوصول إلى القيم الموجودة في النطاق المعرفة بها، وسنوضح كيف تسمح لنا مزايا المغلفات بإعادة استخدام شيفرتنا البرمجية وتخصيص سلوكها.

13.1.1 الحصول على المعلومات من البيئة باستخدام المغلفات

سنفحص أولًا كيفية استخدام المغلفات للحصول على القيم من البيئة التي عرّفناها فيها لاستخدام لاحق. إليك حالة استخدام ممكنة: لدينا شركة لبيع القمصان ونمنح شخصًا ما على قائمة مراسلة البريد الإلكتروني قميصًا حصريًا بين الحين والآخر مثل ترويج لشركتنا، ويمكن أن يُضيف الأشخاص على قائمة المراسلة لونهم المفضل إلى ملفهم بصورةٍ اختيارية، وإذا حدّد الشخص الذي سيحصل على قميص مجاني لونه المفضل فإنه يحصل على هذا اللون تحديدًا وإلا فإنه يحصل على اللون المتواجد بكثرة في المخزن.

هناك عدة طرق لتطبيق ذلك، إذ يمكننا على سبيل المثال استخدام معدّد enum يدعى `ShirtColor` يحتوي على متغايرين variants هما `Red` و `Blue` (حددنا لونين فقط للبساطة). نمثّل مخزن الشركة باستخدام الهيكل `Inventory` الذي يحتوي على حقل يدعى `shirts` يحتوي على النوع `Vec<ShirtColor>`، الذي يمثّل لون القميص الموجود حاليًا في المخزن. يحصل التابع `giveaway` المُعرّف في `Inventory` على لون القميص المفضّل الاختياري للمستخدم من المستخدمين الرابحين القميص مجانًا ويُعيد لون القميص الذي سيحصل عليه المستخدم. التطبيق لكل ما سبق ذكره موضح في الشيفرة 1:

اسم الملف: src/main.rss

```
#[derive(Debug, PartialEq, Copy, Clone)]
enum ShirtColor {
    Red,
    Blue,
}

struct Inventory {
    shirts: Vec<ShirtColor>,
}

impl Inventory {
    fn giveaway(&self, user_preference: Option<ShirtColor>) ->
ShirtColor {
```

```
        user_preference.unwrap_or_else(|| self.most_stocked())
    }

    fn most_stocked(&self) -> ShirtColor {
        let mut num_red = 0;
        let mut num_blue = 0;

        for color in &self.shirts {
            match color {
                ShirtColor::Red => num_red += 1,
                ShirtColor::Blue => num_blue += 1,
            }
        }
        if num_red > num_blue {
            ShirtColor::Red
        } else {
            ShirtColor::Blue
        }
    }
}

fn main() {
    let store = Inventory {
        shirts: vec![ShirtColor::Blue, ShirtColor::Red,
ShirtColor::Blue],
    };

    let user_pref1 = Some(ShirtColor::Red);
    let giveaway1 = store.giveaway(user_pref1);
    println!(
        "The user with preference {:?} gets {:?}",
        user_pref1, giveaway1
    );

    let user_pref2 = None;
    let giveaway2 = store.giveaway(user_pref2);
    println!(
```

```
            "The user with preference {:?} gets {:?}",
            user_pref2, giveaway2
        );
    }
```

[الشيفرة 1: شيفرة توزيع القمصان للمستخدمين]

يحتوي المتغير `store` المعرف في الدالة `main` على قميصين أحدهما باللون الأزرق والآخر باللون الأحمر للتوزيع ضمن حملة التسويق هذه. نستدعي التابع `giveaway` للمستخدم الذي يفضّل القميص الأحمر وللمستخدم الذي ليس لديه أي تفضيل معين.

يمكن تطبيق الشيفرة البرمجية بمختلف الطرق، إلا أننا نركز هنا على استخدام المغلفات، لذا فقد التزمنا بالمفاهيم التي تعلمتها مسبقًا باستثناء ما بداخل التابع `giveaway` الذي يستخدم مغلّفًا. نحصل على تفضيل المستخدم مثل معامل من النوع `Option<ShirtColor>` في التابع `giveaway` ونستدعي التابع `unwrap_or_else` على `user_preference`. التابع `unwrap_or_else` على النوع `Option<T>` مُعرّف في المكتبة القياسية ويأخذ وسيطًا واحدًا ألا وهو مغلف دون أي وسطاء يعيد القيمة `T` (النوع ذاته المُخزن في المتغاير `Some` داخل النوع `Option<T>`، وفي هذه الحالة `ShirtColor`). إذا كان النوع `Option<T>` هو المتغاير `Some`، سيُعيد التابع `unwrap_or_else` القيمة الموجودة داخل `Some`، وإذا كان المتغاير داخل النوع `Option<T>` هو `None`، سيستدعي التابع المغلف ويُعيد القيمة المُعادة من المغلف.

نحدد تعبير المغلف بالشكل `|| self.most_stocked()` مثل وسيط للتابع `unwrap_or_else`. لا يأخذ هذا المغلف أي معاملات، إذ نضع المعاملات بين الخطين العموديين إذا احتوى المغلف على معاملات. يستدعي متن المغلف التابع `self.most_stocked()`، ونعرّف هنا المغلف بحيث يُقيّم تطبيق `unwrap_or_else` المغلف لاحقًا إذا احتجنا للنتيجة.

يطبع تنفيذ الشيفرة البرمجية السابقة الخرج التالي:

```
$ cargo run
   Compiling shirt-company v0.1.0 (file:///projects/shirt-company)
    Finished dev [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/shirt-company`
The user with preference Some(Red) gets Red
The user with preference None gets Blue
```

الجانب المثير للاهتمام هنا هو أننا نمرر مغلفًا يستدعي `self.most_stocked()` على نسخة `Inventory` الحالية. لا تحتاج المكتبة القياسية لمعرفة أي شيء بخصوص النوعين `Inventory` أو `ShirtColor` الذين عرّفناهما، أو المنطق الذي نريد استخدامه في هذه الحالة، إذ أن المغلف يحصل على

مرجع ثابت immutable reference إلى `self` الخاصة بنسخة `Inventory`، ثم يمرره إلى الشيفرة البرمجية التي كتبناها داخل التابع `unwrap_or_else`. إذا قارنّا هذه العملية بالتوابع، فالتوابع غير قادرة على الحصول على هذه المعلومات من البيئة بالطريقة ذاتها.

13.1.2 استنتاج نوع المغلف وتوصيفه

هناك المزيد من الاختلافات بين الدوال والمغلفات، إذ لا تتطلب المغلفات منك عادةً توصيف أنواع المعاملات، أو نوع القيمة المُعادة مثلما تتطلب الدوال `fn` ذلك، والسبب في ضرورة تحديد الأنواع في الدوال هو لأن الأنواع جزءٌ من واجهة صريحة مكشوفة لمستخدميك وتعريف الواجهة بصورةٍ دقيقة مهمٌ للتأكد من أن الجميع متفقٌ على أنواع القيم التي تستخدمها الدالة وتعيدها، بينما لا تُستخدم المغلّفات في الواجهات المكشوفة بهذه الطريقة، إذ أنها تُخزّن في متغيرات دون تسميتها وكشفها لمستخدمي مكتبتك.

تكون المغلفات عادةً قصيرة وترتبط بسياق معين ضيق بدلًا من حالة عامة اعتباطية، ويمكن للمصرّف في هذا السياق المحدود استنتاج أنواع المعاملات والقيمة المعادة بصورةٍ مشابهة لاستنتاجه لمعظم أنواع المتغيرات، وهناك حالات نادرة يحتاج فيها المصرف وجود توصيف للأنواع في المغلفات أيضًا.

يمكننا إضافة توصيف النوع إذا أردنا لزيادة دقة ووضوح المغلف كما هو الحال عند تعريف المتغيرات، إلا أن هذه الطريقة تتطلب كتابةً أطول غير ضرورية. يبدو توصيف الأنواع في المغلفات مثل التعريف الموجود في الشيفرة 2، ونعرّف في هذا المثال مغلّفًا في النقطة التي نمرّر فيها وسيطًا كما فعلنا في الشيفرة 1.

اسم الملف: src/main.rs

```
    let expensive_closure = |num: u32| -> u32 {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    };
```

[الشيفرة 2: إضافة توصيف اختياري لأنواع المعاملات والقيمة المُعادة في المغلف]

تبدو طريقة كتابة المغلفات مشابهة لطريقة كتابة الدوال بعد إضافة توصيف النوع، إذ نعرّف هنا دالةً تجمع 1 إلى المعامل ومغلّفًا بالسلوك ذاته للمقارنة بين الاثنين، ويوضح ذلك كيف أن طريقة كتابة المغلفات مشابهة لطريقة كتابة الدوال باستثناء استخدام الرمز | وكمية الصياغة الاختيارية:

```
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
```

```
let add_one_v4 = |x|               x + 1  ;
```

يوضح السطر الأول تعريف دالة، بينما يوضح السطر الثاني تعريف مغلف موصّف بالكامل، ثمّ نزيل في السطر الثالث التوصيف من تعريف المغلف، ونزيل في السطر الرابع الأقواس الاختيارية لأن محتوى المغلف يتألف من تعبير واحد، وجميع التعاريف السابقة تعاريف صالحة الاستخدام تمنحنا السلوك ذاته عند استدعائها. يتطلب السطران `add_one_v3` و `add_one_v4` تقييم قيمة المغلف حتى يجري تصريفهما، لأن الأنواع يجب أن تُستنتج من خلال استخدامهما وهذا الأمر مشابه لحاجة السطر `let v = Vec::new();` لتوصيف النوع أو إضافة قيم من نوع ما إلى `Vec`، بحيث تستطيع رست استنتاج النوع.

يستنتج المصرف نوعًا واحدًا ثابتًا لكل من المعاملات في حال تعريف المغلفات، إضافةً إلى النوع المُعاد منها. على سبيل المثال، توضح الشيفرة 3 تعريف مغلف قصير يُعيد القيمة التي يتلقاها مثل معاملٍ له، وهذا المغلف ليس مفيدًا جدًا إلا أن هدفه توضيحي. لاحظ أننا لم نضِف أي توصيف للنوع في التعريف وبالتالي يمكننا استدعاء المغلف باستخدام أي نوع، وهو ما فعلناه في الشيفرة 3، إذ استدعينا المغلّف في المرة الأولى باستخدام النوع `String`، إلا أننا حصلنا على خطأ عند استدعائنا للمغلف `example_closure` باستخدام قيمة عدد صحيح integer.

اسم الملف: src/main.rs

```
    let example_closure = |x| x;

    let s = example_closure(String::from("hello"));
    let n = example_closure(5);
```

[الشيفرة 3: محاولة استدعاء مغلف يُستنتج نوعه باستخدام نوعين مختلفين]

يعرض لنا المصرّف الخطأ التالي:

```
$ cargo run
   Compiling closure-example v0.1.0 (file:///projects/closure-example)
error[E0308]: mismatched types
 --> src/main.rs:5:29
  |
|     let n = example_closure(5);
  |             --------------- ^- help: try using a conversion
method: `.to_string()`
  |             |               |
  |             |               expected struct `String`, found
integer
```

```
  |             arguments to this function are incorrect
  |
note: closure parameter defined here
 --> src/main.rs:2:28
  |
|     let example_closure = |x| x;
  |                           ^

For more information about this error, try `rustc --explain E0308`.
error: could not compile `closure-example` due to previous error
```

استنبط المصرّف نوع x المُمرّر بكونه سلسلة نصية String عندما رأى أن أول استخدام للمغلف example_closure كان باستخدام قيمة String، كما استنبط القيمة المعادة بالنوع String. تُقيّد هذه الأنواع في المغلف example_closure من تلك النقطة فصاعدًا وسنحصل على خطأ، إذا حاولنا استخدام نوع مختلف بعد ذلك في المغلف ذاته.

13.1.3 الحصول على المراجع أو نقل الملكية

يمكن أن تحصل المغلفات على القيم من البيئة بثلاث طرق وهي مرتبطة بالطرق الثلاث التي تستطيع فيها الدالة الحصول على القيم على أنها معاملاتها: الاستعارة الثابتة أو الاستعارة المتغيّرة أو أخذ الملكية، ويحدد المغلف واحدًا من هذه الطرق الثلاث حسب القيم الموجودة في متن الدالة.

نعرّف في الشيفرة 4 مغلفًا يحصل على مرجع ثابت لشعاع يدعى list لأنه يحتاج فقط للمراجع الثابتة لطباعة القيمة:

اسم الملف: src/main.rs

```
fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    let only_borrows = || println!("From closure: {:?}", list);

    println!("Before calling closure: {:?}", list);
    only_borrows();
    println!("After calling closure: {:?}", list);
}
```

[الشيفرة 4: تعريف مغلف واستدعاءه، بحيث يحصل هذا المغلف على مرجع ثابت]

يوضح هذا المثال أيضًا أنه من الممكن للمتغير أن يرتبط بتعريف مغلف، ويمكننا لاحقًا استدعاء المغلف باستخدام اسم المتغير والقوسين وكأن اسم المتغير يمثل اسم دالة.

يمكن الوصول للشعاع `list` من الشيفرة البرمجية قبل تعريف المغلف وبعد تعريفه، شرط أن يكون قبل استدعاء المغلف وبعد استدعاء المغلف، وذلك لأنه من الممكن وجود عدّة مراجع ثابتة للشعاع `list` في ذات الوقت، وتُصرَّف الشيفرة البرمجية بنجاح وتُنفَّذ وتطبع التالي:

```
$ cargo run
   Compiling closure-example v0.1.0 (file:///projects/closure-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/closure-example`
Before defining closure: [1, 2, 3]
Before calling closure: [1, 2, 3]
From closure: [1, 2, 3]
After calling closure: [1, 2, 3]
```

نعدّل في الشيفرة 5 متن المغلف بحيث نضيف عنصرًا إلى الشعاع `list`، وبالتالي يحصل المغلف الآن على مرجع متغيّر:

اسم الملف: src/main.rs

```
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    let mut borrows_mutably = || list.push(7);

    borrows_mutably();
    println!("After calling closure: {:?}", list);
}
```

[الشيفرة 5: تعريف واستدعاء مغلف يحتوي على مرجع متغيّر]

تُصرَّف الشيفرة البرمجية بنجاح وتُنفَّذ ونحصل على الخرج التالي:

```
$ cargo run
   Compiling closure-example v0.1.0 (file:///projects/closure-example)
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/closure-example`
Before defining closure: [1, 2, 3]
After calling closure: [1, 2, 3, 7]
```

لاحظ أننا لا نستخدم `println!` بين تعريف المغلف `borrows_mutably` واستدعائه، إذ يحصل المغلف على مرجع متغيّر للشعاع `list` عند تعريفه، ولا نستخدم المغلف مجددًا بعد استدعائه لذا تنتهي الاستعارة المتغيرة عندها. لا يُسمح بطباعة مرجع متغيّر بين تعريف المغلف واستدعائه، إذ لا يُسمح بوجود أي عمليات استعارة أخرى عند وجود استعارة متغيّرة. حاول إضافة استدعاء للماكرو `println!` لترى رسالة الخطأ التي تظهر لك.

إذا أردت إجبار المغلف على أخذ ملكية القيم التي يستخدمها في البيئة على الرغم من أن متن المغلف لا يتطلب أخذ الملكية، فيمكنك استخدام الكلمة المفتاحية `move` قبل قائمة المعاملات.

هذه الطريقة مفيدة خصوصًا عند تمرير مغلف لخيط thread جديد لنقل البيانات، بحيث تُمتلك بواسطة الخيط الجديد. سنناقش الخيوط بالتفصيل وسبب استخدامنا لها لاحقًا، لكن دعنا للوقت الحالي نستكشف عملية إضافة خيط جديد باستخدام مغلّف يحتاج الكلمة المفتاحية `move`. توضح الشيفرة 6 إصدارًا عن الشيفرة 4، إلا أننا نطبع الشعاع هنا في خيط جديد بدلًا من الخيط الرئيسي:

اسم الملف: src/main.rs

```
use std::thread;

fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    thread::spawn(move || println!("From thread: {:?}", list))
        .join()
        .unwrap();
}
```

[الشيفرة 6: استخدام `move` لإجبار خيط المغلف على أخذ ملكية `list`]

نُنشئ خيطًا جديدًا وذلك بمنح الخيط مغلف ليعمل به مثل وسيط، ويطبع متن المغلف القائمة. يحصل المغلف في الشيفرة 4 على `list` باستخدام مرجع ثابت لأنها تُعد أدنى درجات الوصول المطلوبة لطباعة محتويات `list`، إلا أننا في هذا المثال نحدد أن `list` يجب أن تُنقل إلى المغلف بإضافة الكلمة المفتاحية

`move` في بداية تعريف المغلّف على الرغم من أن متن المغلف يتطلب فقط مرجعًا ثابتًا. يمكن للخيط الجديد أن ينتهي من التنفيذ قبل انتهاء الخيط الرئيسي من التنفيذ أو أن ينتهي الخيط الرئيسي أولًا، وإذا احتفظ الخيط الرئيسي بملكية `list` وانتهى من التنفيذ قبل الخيط الجديد وأسقط `list` فهذا يعني أن المرجع الثابت في الخيط الجديد سيكون غير صالحًا، ولذلك يتطلب المصرف نقل `list` إلى المغلف في الخيط الجديد بحيث يبقى المرجع صالحًا داخله. جرّب إزالة الكلمة المفتاحية `move` أو استخدام `list` في الخيط الرئيسي بعد تعريف المغلف لرؤية خطأ المصرف الذي يظهر لك.

13.1.4 نقل القيم خارج المغلفات وسمات Fn

تعرّف الشيفرة البرمجية الموجودة داخل مغلف ما الأمر الذي سيحصل للمراجع أو القيم بعد أن يُقيّم المغلف (بالتالي التعريف على الشيء الذي نُقل **خارج** المغلف إذا كان موجودًا) وذلك بعد أن يحصل المغلف على المرجع أو ملكية قيمة ما من البيئة مكان تعريفه (بالتالي التعريف على الشيء الذي نُقل **إلى** المغلف إذا كان موجودًا). يمكن لمتن المغلف أن يفعل أيًا من الأشياء التالية: نقل قيمة خارج المغلّف، أو التعديل على قيمة داخل المغلف، أو عدم نقل القيمة وعدم تعديلها، أو عدم الحصول على أي قيمة من البيئة في المقام الأول.

تؤثر الطريقة التي يحصل بها المغلف على القيم ويتعامل معها من البيئة على السمات التي يطبقها المغلف والسمات traits هي الطريقة التي تستطيع فيها كل من الدوال والهياكل تحديد نوع المغلفات الممكن استخدامها. تطبّق المغلفات تلقائيًا سمةً أو سمتين أو ثلاث من سمات `Fn` التالية على نحوٍ تراكمي بحسب تعامل متن المغلف للقيم:

1. السمة `FnOnce`: تُطبَّق على جميع المغلفات الممكن استدعاؤها مرةً واحدة. تُطبّق جميع المغلفات هذه السمة على الأقل لأنه يمكن لجميع المغلفات أن تُستدعى، وسيطبق المغلف الذي ينقل القيم خارج متنه السمة `FnOnce` فقط دون أي سمات `Fn` أخرى لأنه يُمكن استدعاءه مرةً واحدةً فقط.
2. السمة `FnMut`: تُطبَّق على جميع المغلفات التي لا تنقل القيم خارج متنها، إلا أنها من الممكن أن تعدّل على هذه القيم، ويمكن استدعاء هذه المغلفات أكثر من مرة واحدة.
3. السمة `Fn`: تُطبَّق على المغلفات التي لا تنقل القيم خارج متنها ولا تعدل على القيم، إضافةً إلى المغلفات التي لا تحصل على أي قيم من البيئة. يمكن لهذه المغلفات أن تُستدعى أكثر من مرة واحدة دون التعديل على بيئتها وهو أمرٌ مهم في حالة استدعاء مغلف عدة مرات على نحوٍ متعاقب.

دعنا ننظر إلى تعريف التابع `unwrap_or_else` على النوع `Option<T>` الذي استخدمناه في الشيفرة 1:

```
impl<T> Option<T> {
    pub fn unwrap_or_else<F>(self, f: F) -> T
    where
        F: FnOnce() -> T
```

```
        {
            match self {
                Some(x) => x,
                None => f(),
            }
        }
    }
```

تذكّر أن `T` هو نوع معمم generic type يمثل نوع القيمة في المتغاير `Some` للنوع `Option`، وهذا النوع `T` يمثل أيضًا نوع القيمة المعادة من الدالة `unwrap_or_else`، إذ ستحصل الشيفرة البرمجية التي تستدعي `unwrap_or_else` على النوع `Option<String>` على النوع `String` على سبيل المثال.

لاحظ تاليًا أن الدالة `unwrap_or_else` تحتوي على معامل نوع معمم إضافي يُدعى `F` والنوع `F` هنا هو نوع المعامل ذو الاسم `f` وهو المغلف الذي نمرره للدالة `unwrap_or_else` عند استدعائها.

حد السمة trait bound المحدد على النوع المعمم `E` هو `FnOnce() -> T`، مما يعني أن النوع `F` يجب أن يكون قابلًا للاستدعاء مرةً واحدة وألّا يأخذ أي وسطاء وأن يعيد قيمةً من النوع `T`. يوضح استخدام `FnOnce` في حد السمة القيد: أن `unwrap_or_else` ستستدعي `f` مرةً واحدةً على الأكثر. يمكنك من رؤية متن الدالة `unwrap_or_else` معرفة أن `f` لن تُستدعى إذا كان المتغاير `Some` موجودًا في `Option`، بينما ستُستدعى `f` مرةً واحدة إذا وُجد المتغاير `None` في `Option`. تقبل الدالة `unwrap_or_else` أنواعًا مختلفة من المغلفات بصورةٍ مرنة، لأن جميع المغلفات تطبّق السمة `FnOnce`.

> يمكن أن تطبق الدوال سمات `Fn` الثلاث أيضًا. يمكننا استخدام اسم الدالة بدلًا من مغلف عندما نريد شيئًا يطبق واحدةً من سمات `Fn` إذا كان ما نريد فعله لا يتطلب الحصول على قيمة من البيئة. يمكننا على سبيل المثال، استدعاء `unwrap_or_else(Vec::new)` على قيمة `Option<Vec<T>>` للحصول على شعاع فارغ جديد إذا كانت القيمة هي `None`.

دعنا ننظر الآن إلى تابع المكتبة القياسية `sort_by_key` المعرف على الشرائح slices لرؤية الاختلاف بينه وبين `unwrap_or_else` وسبب استخدام `sort_by_key` للسمة `FnMut` بدلًا من السمة `FnOnce` لحد السمة. يحصل المغلف على وسيط واحد على هيئة مرجع للعنصر الحالي في الشريحة ويُعيد قيمةً من النوع `K` يمكن ترتيبها، وهذه الدالة مفيدة عندما تريد ترتيب شرحة بسمة attribute معينة لكل عنصر. لدينا في الشيفرة 7 قائمة من نسخ `instances` من الهيكل `Rectangle` ونستخدم `sort_by_key` لترتيبها حسب سمة `width` من الأصغر إلى الأكبر:

اسم الملف: src/main.rs

```
#[derive(Debug)]
```

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    list.sort_by_key(|r| r.width);
    println!("{:#?}", list);
}
```

[الشيفرة 7: استخدام `sort_by_key` لترتيب المستطيلات بحسب عرضها]

نحصل على الخرج التالي مما سبق:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.41s
     Running `target/debug/rectangles`
[
    Rectangle {
        width: 3,
        height: 5,
    },
    Rectangle {
        width: 7,
        height: 12,
    },
    Rectangle {
        width: 10,
        height: 1,
    },
```

```
]
```

السبب في كون `sort_by_key` معرفًا ليأخذ مغلقًا يطبق السمة `FnMut` هو استدعاء الدالة للمغلـف عـدة مرات: مرةً واحدةً لكل عنصر في الشريحة. لا يحصل المغلف `|r| r.width` على أي قيمة من البيئة أو يعـدل عليها أو ينقلها لذا فهو يحقق شروط حد السمة هذه.

توضح الشيفرة 8 مثالًا لمغلف على النقيض، إذ يطبق هذا المغلف السمة `FnOnce` فقط لأنـه ينقـل قيمـة خارج البيئة، ولن يسمح لنا المصرّف باستخدام هذا المغلف مع `sort_by_key`:

اسم الملف: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    let mut sort_operations = vec![];
    let value = String::from("by key called");

    list.sort_by_key(|r| {
        sort_operations.push(value);
        r.width
    });
    println!("{:#?}", list);
}
```

[الشيفرة 8: محاولة استخدام مغلف يطبق السمة `FnOnce` فقط مع التابع `sort_by_key`]

يمثّل ما سبق طريقةً معقدة (لا تعمل بنجاح) لمحاولة عدّ المرات التي يُستدعى بها التـابع `sort_by_key` عند ترتيب `list`، وتحاول الشيفرة البرمجية تحقيق ذلك بإضافة `value` -ألا وهي قيمة من النوع `String` من

بيئة المغلف- إلى الشعاع sort_operations. يحصل المغلف على القيمة value، ثم ينقلها خارج المغلـف بنقل ملكيتها إلى الشعاع sort_operations، ويمكن أن يُستدعى هـذا المغلـف مـرةً واحـدةً إلا أن محاولـة استدعائه للمـرة الثانيـة لن تعمـل لأن value لن يكـون في البيئـة ليُضـاف إلى الشـعاع sort_operations مجددًا، وبالتالي يطبق هذا المغلف السمة FnOnce فقط، وعندما نحاول تصـريف الشـيفرة البرمجيـة السـابقة، سنحصل على خطأ مفاده أن value لا يمكن نقلها خارج المغلف لأن المغلف يجب أن يطبّق السمة FnMut:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
error[E0507]: cannot move out of `value`, a captured variable in an
`FnMut` closure
  --> src/main.rs:18:30
   |
|     let value = String::from("by key called");
   |         ----- captured outer variable
|
|     list.sort_by_key(|r| {
   |                      --- captured by this `FnMut` closure
|         sort_operations.push(value);
   |                              ^^^^^ move occurs because `value`
has type `String`, which does not implement the `Copy` trait

For more information about this error, try `rustc --explain E0507`.
error: could not compile `rectangles` due to previous error
```

يشير الخطأ إلى السطر الذي ننقل فيه القيمة value خارج البيئة داخل متن المغلف، ولتصحيح هذا الخطـأ علينا تعديل متن المغلف بحيث لا ينقل القيم خارج البيئة. نحافظ على وجود عدّاد في البيئة ونزيد قيمته داخـل المغلف بحيث نستطيع عدّ المرات التي يُستدعى فيها التابع sort_by_key. يعمل المغلف في الشيفرة 9 مع sort_by_key لأنه يحصـل فقـط على المرجـع المتغيّـر الخـاص بالعـداد num_sort_operations ويمكن بالتالي استدعاؤه أكثر من مرة واحدة:

اسم الملف: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}
```

```
fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    let mut num_sort_operations = 0;
    list.sort_by_key(|r| {
        num_sort_operations += 1;
        r.width
    });
    println!("{:#?}, sorted in {num_sort_operations} operations",
list);
}
```

[الشيفرة 9: استخدام مغلف يطبق السمة `FnMut` مع التابع `sort_by_key` دون الحصول على أخطاء]

سمات `Fn` مهمة عند تعريف أو استخدام الدوال أو الأنواع التي تستخدم المغلفات. سنناقش تاليًا المكررات iterators، إذ أن العديد من توابع المكررات تأخذ المغلفات مثل وسطاء، لذا تذكّر التفاصيل المتعلقة بالمغلّفات عند قراءة القسم التالي

13.2 معالجة سلسلة من العناصر باستخدام المكررات iterators

يسمح لك نمط المكرّر iterator pattern بإنجاز مهمة ما على سلسلة من العناصر بصورةٍ متتالية، والمكرّر مسؤول عن المنطق الخاص بالمرور على كل عنصر وتحديد مكان انتهاء السلسلة، إذ ليس من الواجب عليك إعادة تطبيق هذا المنطق بنفسك عند استخدام المكررات.

المكررات في رست كسولة، بمعنى أنها لا تمتلك أي تأثير حتى تستدعي أنت التابع الذي يستخدم المكرر، فعلى سبيل المثال نُنشئ الشيفرة 10 مكرّرًا على العناصر الموجودة في الشعاع `v1` باستدعاء التابع `iter` المعرّف على `Vec<T>`، ولا تفعل تلك الشيفرة البرمجية أي شيء مفيد.

```
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();
```

[الشيفرة 10: إنشاء مكرر]

نخـزّن المكـرر في المتغـير `v1_iter`، ويمكننـا اسـتخدامه بطـرق عـدّة بعـد إنشـائه. على سـبيل المثـال، استخدمنا حلقة `for` في الشيفرة 5 من الفصل الثالث للمرور على مصفوفة وذلـك لتنفيـذ شـيفرة برمجيـة على كل من عناصرها، وفي الحقيقة كان المكرّر موجودًا ضمنيًا في تلـك الشـيفرة لتحقيـق ذلـك إلا أننـا لم نتكلم عن ذلك سابقًا.

نفصل في الشيفرة 11 إنشاء المكرر من استخدامه في الحلقة `for`، فعندما تُستدعى الحلقة `for` باستخدام المكرر `v1_iter`، يُستخدم كل عنصر في المكرر مرةً تلو الأخرى في كل دورة للحلقة، وهذا يطبـع كـل قيمـة من قيم العناصر.

```
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    for val in v1_iter {
        println!("Got: {}", val);
    }
```

[الشيفرة 11: استخدام مكرر في حلقة `for`]

يمكنك كتابة شيفرة برمجية تفعل الأمر نفسه في لغـات البرمجـة الـتي لا تسـتخدم المكـررات في مكتبتهـا القياسية وذلك عن طريق البدء بالمتغير من الدليل 0 واستخدام قيمة المتغير يمثابة دليلٍ للشعاع للحصول على قيمة ومن ثم زيادة قيمة المتغير في الحلقة حتى تصل قيمته لعدد العناصر الكلية في الشعاع.

تتعامل المكررات مع المنطق البرمجي نيابـةً عنـك ممـا يقلـل من الشـيفرات البرمجيـة المتكـررة الـتي من الممكن أن تتسبب بأخطاء، وتعطيك المكررات مرونةً أكبر في التعامـل مـع المنطـق ذاتـه في الكثـير من أنـواع السلاسل وليس فقط هياكل البيانات data structures التي يمكنك الوصول إلى قيمها باستخدام دليـل مثـل الشعاع. دعنا نلقي نظرةً إلى كيفية تحقيق المكررات لكل هذا.

13.2.1 سمة Iterator وتابع next

تطبّق جميـع المكـررات السـمة `Iterator` المُعرّفـة في المكتبـة القياسـية، ويبـدو تعريـف السـمة شـيء مماثل لهذا:

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
```

```
    // methods with default implementations elided
}
```

لاحظ أن هذا التعريف يستخدم صيغتان جديدتان، هما: `type Item` و `Self::Item` اللتان تُعرِّفان نوعًا مترابطًا associated type مع هذه السمة. سنتحدث عن الأنواع المترابطة بالتفصيل لاحقًا، ويكفي للآن معرفتك أن هذه الشيفرة البرمجية تقول أن تطبيق السمة `Iterator` يتطلب منك تعريف نوع `Item` أيضًا وهذا النوع مُستخدم مثل نوع مُعاد من التابع `next`، وبكلمات أخرى، سيكون النوع `Item` هو النوع المُعاد من المكرّر.

تتطلب السمة `Iterator` ممن يطبّقها فقط أن يعرّف تابعًا واحدًا هو `next`، الذي يُعيد عنصرًا واحدًا من المكرر كل مرة ضمن `Some` وعندما تنتهي العناصر (ينتهي التكرار)، يُعيد `None`.

يمكننا استدعاء التابع `next` على المكررات مباشرةً، وتوضح الشيفرة 12 القيم المُعادة من الاستدعاءات المتعاقبة على المكرّر باستخدام `next` وهو المكرر الموجود في الشعاع.

اسم الملف: src/lib.rs

```
    #[test]
    fn iterator_demonstration() {
        let v1 = vec![1, 2, 3];

        let mut v1_iter = v1.iter();

        assert_eq!(v1_iter.next(), Some(&1));
        assert_eq!(v1_iter.next(), Some(&2));
        assert_eq!(v1_iter.next(), Some(&3));
        assert_eq!(v1_iter.next(), None);
    }
```

[الشيفرة 12: استدعاء التابع `next` على المكرر]

لاحظ أننا احتجنا لإنشاء `v1_iter` متغيّر mutable، إذ يغيّر استدعاء التابع `next` على مكرر الحالة الداخلية التي يستخدمها المكرر لتتبع مكانه ضمن السلسلة، وبكلمات أخرى، **تستهلك consumes** الشيفرة البرمجية المكرّر، إذ يستهلك كل استدعاء للتابع `next` عنصرًا واحدًا من المكرر. لم يكن هناك أي حاجة لإنشاء `v1_iter` متغيّر عندما استخدمنا حلقة `for` لأن الحلقة أخذت ملكية ownership المكرر `v1_iter` وجعلته متغيّرًا ضمنيًا.

لاحظ أيضًا أن القيم التي نحصل عليها من استدعاءات التابع `next` هي مراجع ثابتة immutable references للقيم الموجودة في الشعاع، ويعطينا التابع `iter` مكررًا على المراجع الثابتة. إذا أردنا إنشاء مكرر يأخذ ملكية `v1` ويُعيد القيم المملوكة فيمكننا استدعاء `into_iter` بدلًا من `iter`، ويمكننا بصورةٍ مماثلة استدعاء `iter_mut` بدلًا من `iter` إذا أردنا المرور على مراجع متغيّرة.

13.2.2 توابع تستهلك المكرر

للسمة `Iterator` العديد من التوابع في تطبيقها الافتراضي والموجودة في المكتبة القياسية، ويمكنك الاطّلاع على هذه التوابع عن طريق النظر إلى توثيق واجهة المكتبة القياسية البرمجية للسمة `Iterator`. تستدعي بعض هذه التوابع التابع `next` في تعريفها وهو السبب في ضرورة تطبيق التابع `next` عند تطبيق السمة `Iterator`.

تُسمّى التوابع التي تستدعي التابع `next` بالمحوّلات المُستهلكة consuming adaptors لأن استدعائها يستهلك المكرر. يُعد تابع `sum` مثال على هذه التوابع، فهو يأخذ ملكية المكرر ويمرّ على عناصره بصورةٍ متتالية مع استدعاء `next` مما يتسبب باستهلاك المكرر، وبينما يمرّ على العناصر فهو يجمع كل عنصر إلى قيمة كلية، ثم يعيد القيمة الكلية في النهاية. تحتوي الشيفرة 13 على اختبار يوضح استخدام التابع `sum`:

اسم الملف: src/lib.rs

```
#[test]
fn iterator_sum() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}
```

[الشيفرة 13: استدعاء التابع `sum` للحصول على القيمة الكلية لجميع العناصر في المكرر]

لا يُسمح لنا باستخدام `v1_iter` بعد استدعاء `sum` لأن `sum` يأخذ ملكية المكرّر الذي نستدعي `sum` عليه.

13.2.3 التوابع التي تنشئ مكررات أخرى

محولات المكرر iterator adaptors هي توابع مُعرّفة على السمة `Iterator` ولا تستهلك المكرر، بل تُنشئ مكررات مختلفة بدلًا من ذلك عن طريق تغيير بعض خصائص المكرر الأصل.

توضح الشيفرة 14 مثالًا عن استدعاء تابع محول مكرر `map` وهو تابع يأخذ مغلّفًا closure ويستدعيه على كل من العناصر بصورةٍ متتالية. يُعيد التابع `map` مكررًا جديدًا يُنشئ عناصرًا مُعدّلٌ عليها، ويُنشئ المغلف في هذه الحالة مكررًا جديدًا يزيد قيمة عناصره بمقدار 1 لكل عنصر في الشعاع:

اسم الملف: src/main.rs

```
    let v1: Vec<i32> = vec![1, 2, 3];

    v1.iter().map(|x| x + 1);
```

[الشيفرة 14: استدعاء محول المكرر `map` لإنشاء مكرر جديد]

إلا أن الشيفرة البرمجية السابقة تعطينا إنذارًا:

```
$ cargo run
   Compiling iterators v0.1.0 (file:///projects/iterators)
warning: unused `Map` that must be used
 --> src/main.rs:4:5
  |
|     v1.iter().map(|x| x + 1);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_must_use)]` on by default
  = note: iterators are lazy and do nothing unless consumed

warning: `iterators` (bin "iterators") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.47s
     Running `target/debug/iterators`
```

لا تفعل الشيفرة 14 أي شيء، إذ أن المغلف الذي حددناه لا يُستدعى أبدًا، ويذكرنا الإنذار بسبب ذلك: إذ أن محولات المكرر كسولة ونحتاج لاستهلاك المكرر هنا.

نستخدم التابع `collect` لتصحيح هذا الإنذار واستهلاك المكرر وهو ما استخدمناه في (الشيفرة 1 من الفصل 12) باستخدام `env::args`، إذ يستهلك هذا التابع المكرر ويجمّع القيم الناتجة إلى نوع بيانات تجميعة collection data type.

نجمع في الشيفرة 15 النتائج من عملية المرور على المكرر والمُعادة من استدعاء التابع `map` على الشعاع، وسيحتوي هذا الشعاع في نهاية المطاف على جميع عناصره الموجودة مع زيادة على قيمة كل منها بمقدار 1.

اسم الملف: src/main.rs

```
    let v1: Vec<i32> = vec![1, 2, 3];

    let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

    assert_eq!(v2, vec![2, 3, 4]);
```

[الشيفرة 15: استدعاء التابع `map` لإنشاء مكرر جديد ومن ثم استدعاء التابع `collect` لاستهلاك المكرر الجديد وإنشاء شعاع]

يمكننا تحديد أي عملية نريد إنجازها على كل عنصر بالنظر إلى أن `map` تتلقى مغلقًا بمثابة وسيط لها. هذا مثال عظيم عن كيفية إنجاز مهام معقدة بطريقة مقروءة، ولأن جميع المكررات كسولة، فهذا يعني أنك بحاجة استدعاء واحد من توابع المحولات المستهلكة للحصول على نتائج استدعاءات محولات المكرر.

13.2.4 استخدام المغلقات التي تحصل على القيم من بيئتها

تأخذ الكثير من محولات المكرر المغلقات مثل وسطاء لها، وستحصل هذه المغلقات التي تُحدد مثل وسطاء لمحولات المكرر على قيم من بيئتها غالبًا.

نستخدم في هذا المثال التابع `filter` الذي يأخذ مغلّقًا، ويحصل المغلف على عنصر من المكرر ويُعيد `bool`، وتُضمّن هذه القيمة في التكرار المُنشئ بواسطة `filter` إذا أعاد المغلف القيمة `true`، وإذا أعاد المكرر القيمة `false` فلن تُضمَّن القيمة.

نستخدم في الشيفرة 16 التابع `filter` بمغلّف يحصل على المتغير `shoe_size` من بيئته للمرور على تجميعة من نسخ instances الهيكل `Shoe`، وسيُعيد فقط الأحذية ذات مقاس محدد.

اسم الملف: src/lib.rs

```
#[derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

fn shoes_in_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    shoes.into_iter().filter(|s| s.size == shoe_size).collect()
}
```

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn filters_by_size() {
        let shoes = vec![
            Shoe {
                size: 10,
                style: String::from("sneaker"),
            },
            Shoe {
                size: 13,
                style: String::from("sandal"),
            },
            Shoe {
                size: 10,
                style: String::from("boot"),
            },
        ];

        let in_my_size = shoes_in_size(shoes, 10);

        assert_eq!(
            in_my_size,
            vec![
                Shoe {
                    size: 10,
                    style: String::from("sneaker")
                },
                Shoe {
                    size: 10,
                    style: String::from("boot")
                },
            ]
        );
```

```
        }
    }
```

[الشيفرة 16: استخدام التابع `filter` مع مغلف يحصل على القيمة `shoe_size`]

تأخذ الدالة `shoes_in_size` ملكية شعاع الأحذية وقياس الحذاء مثل معاملات، وتُعيد شعاعًا يحتوي على الأحذية بالمقاس المحدد.

نستدعي `into_iter` في متن الدالة `shoes_in_size` لإنشاء مكرر يأخذ ملكية الشعاع، ثم نستدعي `filter` لتحويل المكرّر إلى مكرر جديد يحتوي فقط على عناصر يُعيد منها المغلّف القيمة `true`.

يحصل المغلف على المعامل `shoe_size` من البيئة ويقارنه مع قيمة كل مقاس حذاء مع إبقاء الأحذية التي يتطابق مقاسها، وأخيرًا يجمع استدعاء `collect` القيم المُعادة بواسطة محول المكرر إلى شعاع يُعاد من الدالة.

يوضح هذا الاختبار أنه عندما نستدعي `shoes_in_size`، فنحن نحصل فقط على الأحذية التي تتطابق مقاساتها مع القيمة التي حددناها.

13.3 استخدام المكررات Iterators في تطبيق سطر الأوامر

يمكننا الآن تحسين مشروع سطر الأوامر الذي نفذناه إلى الآن، إذ سنستخدمها لجعل شيفرتنا البرمجية أكثر وضوحًا واختصارًا. دعنا نلقي نظرةً على طريقة تطبيق المكررات في مشروعنا والتي ستجعل منه إصدارًا محسنًا خاصةً على الدالتين `Config::build` و `search`.

13.3.1 إزالة clone باستخدام المكرر

أضفنا في الشيفرة 6 (من الفصل 12) شيفرةً برمجية تأخذ شريحةً slice من القيم ذات النوع `String` وأنشأنا بها نسخةً instance من الهيكل `Config` بالمرور على الشريحة ونسخ القيم والسماح بالهيكل `Config` بامتلاك هذه القيم. سنُعيد كتابة التطبيق ذاته الخاص بالدالة `Config::build` في الشيفرة 17 كما كانت في الشيفرة 23 (الفصل 12):

اسم الملف: src/lib.rs

```
impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }
```

```
        let query = args[1].clone();
        let file_path = args[2].clone();

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
```

[الشيفرة 17: إعادة بناء الدالة `Config::build` من الشيفرة 23 (الفصل 12)]

ذكرنا وقتها أنه ليس علينا القلق بخصوص استدعاءات `clone` غير الفعالة لأننا سنزيلها في المستقبل. حسنًا، أتى الوقت الآن.

نحتاج `clone` هنا لوجود شريحة بعناصر `String` في المعامل `args`، إلا أن الدالة `build` لا تمتلك `args`، ولإعادة ملكية نسخة `Config`، اضطررنا لنسخ القيم من الحقول `query` و `file_path` الموجودة في الهيكل `Config` بحيث تمتلك نسخة `Config` القيم الموجودة في حقوله.

أصبح بإمكاننا -بمعرفتنا الجديدة بخصوص المكررات- التعديل على الدالة `build` لأخذ ملكية مكرر مثل وسيط لها بدلًا من استعارة شريحة، وسنستخدم وظائف المكرر بدلًا من الشيفرة البرمجية التي تتحقق من طول السلسلة النصية وتمرّ على عناصرها في مواقع محددة، وسيوضِّح ذلك استخدام `Config::build` لأن المكرر يستطيع الحصول على هذه القيم.

بعد أخذ الدالة `Config::build` لملكية المكرر وتوقف استخدامها لعمليات الفهرسة indexing التي تستعير القيم، أصبح بإمكاننا نقل قيم `String` من المكرر إلى الهيكل `Config` بدلًا من استدعاء `clone` وإنشاء تخصيص allocation جديد.

١. إزالة المكرر المعاد مباشرة

افتح ملف src/main.rs الخاص بمشروع الدخل والخرج، والذي يجب أن يبدو مماثلًا لما يلي:

اسم الملف: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    // --snip--
}
```

سنعدّل أولًا بداية الدالة `main` التي كانت موجودة في الشيفرة 24 من الفصل 12 لتكون الشيفرة 18 التي تستخدم مكررًا، إلا أنها لن تُصرَّف بنجاح إلا بعد تعديلنا للدالة `Config::build` أيضًا.

اسم الملف: src/main.rs

```
fn main() {
    let config = Config::build(env::args()).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    // --snip--
}
```

[الشيفرة 18: تمرير القيمة المعادة من `env::args` إلى `Config::build`]

تُعيد الدالة `env::args` مكررًا، إذ يمكننا تمرير ملكية المكرّر المُعاد من `env::args` إلى `Config::build` الآن مباشرةً بدلًا من تجميع قيم المكرر في شعاع ثم تمرير شريحة إلى الدالة `Config::build`.

نحتاج إلى تحديث تعريف الدالة `Config::build` في ملف src/lib.rs الخاص بمشروعنا. دعنا نغير بصمة الدالة `Config::build` لتبدو على نحوٍ مماثل للشيفرة 19، إلا أن هذا لن يُصرَّف بنجاح لأننا بحاجة لتحديث متن الدالة أيضًا.

اسم الملف: src/lib.rs

```
impl Config {
    pub fn build(
        mut args: impl Iterator<Item = String>,
    ) -> Result<Config, &'static str> {
        // --snip--
```

[الشيفرة 19: تحديث بصمة الدالة `Config::build` بحيث تتوقع تمرير مكرر]

يوضّح توثيق المكتبة القياسية بالنسبة للدالة `env::args` بأن نوع المكرر المُعاد هـو `std::env::Args` وأن هذا النوع يطبّق السمة `Iterator` ويُعيد قيمًا من النوع `String`.

حدّثنا بصمة الدالة `Config::build` بحيث يحتوي المعامل `args` على نـوع معمم generic type بحـدّ السمة `impl Iterator<Item = String>` بدلًا من trait bound `impl &[String]`. ناقشنا الصيغة `impl Trait` سابقًا وهي تعني أن `args` يمكن أن يكون أي نـوع يطبّـق النـوع `Iterator` ويُعيـد عناصـر من النـوع `String`.

يمكننا إضافة الكلمة المفتاحية `mut` إلى توصيف المعامل `args` لجعلـه متغيّـرًا mutable بـالنظر إلى أننـا نأخذ ملكية `args` وسنعدّل `args` بالمرور ضمنه.

ب. استخدام توابع السمة Iterator بدلا من الفهرسة

سنصحح متن الدالـة `Config::build`، فنحن نعلم أنـه بإمكاننـا اسـتدعاء التـابع `next` على `args` لأنهـا تطبّق السمة `Iterator`. نحدّث في الشيفرة 20 الشيفرة البرمجيـة الـتي كـانت موجـودة في الشـيفرة 23 من الفصل 12 بحيث نستخدم التابع `next`:

اسم الملف: src/lib.rs

```
impl Config {
    pub fn build(
        mut args: impl Iterator<Item = String>,
    ) -> Result<Config, &'static str> {
        args.next();

        let query = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a query string"),
        };
```

```
        let file_path = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a file path"),
        };

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
```

[الشيفرة 20: تعديل محتوى الدالة `Config::build` بحيث نستخدم توابع المكرر]

تذكّر أن أول قيمة من القيم المُعادة من الدالة `env::args` هي اسم البرنامج، لذا نريد تجاهلها والحصول على القيمة التي تليها، ولذلك نستدعي `next` أولًا دون فعل أي شيء بالقيمة المُعادة، ثم نستدعي `next` مرةً أخرى للحصول على القيمة التي نريد وضعها في حقل `query` من الهيكل `Config`. إذا أعادت `next` القيمة `Some`، سنستخدم `match` لاستخلاص القيمة، أما إذا أعادت `None`، فهذا يعني عدم وجود وسطاء كافية من المستخدم وعندها نُعيد من الدالة القيمة `Err` مبكرًا، ونفعل ذلك مجددًا للتعامل مع القيمة `file_path`.

13.3.2 جعل الشيفرة البرمجية أكثر وضوحا باستخدام محولات المكرر

يمكننا أيضًا استغلال ميزة من مزايا المكررات في الدالة `search` ضمن مشروعنا، وهي موضّحة في الشيفرة 21 والشيفرة 19 (من الفصل 12):

اسم الملف: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
```

```
            }
        }

        results
    }
```

[الشيفرة 21: تطبيق الدالة search من الشيفرة 19 (الفصل 12)]

يمكننا كتابة هذه الشيفرة البرمجية بطريقة مختصرة باستخدام توابع محولات المكرر، إذ يسمح لنا ذلك أيضًا بوجود شعاع وسيط متغيّر نسميه results. يفضِّل أسلوب لغات البرمجة العمليّة تقليل كمية الحالات المتغيّرة لجعل الشيفرة البرمجية أكثر وضوحًا، إذ قد تفتح لنا إزالة الحالة المتغيّرة فرصةً لجعل عمليات البحث تُنفَّذ بصورةٍ متزامنة، لأنه لن يكون علينا حينها إدارة الوصول المتتالي إلى الشعاع results. توضح الشيفرة 22 هذا التغيير:

اسم الملف: src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    contents
        .lines()
        .filter(|line| line.contains(query))
        .collect()
}
```

[الشيفرة 22: استخدام توابع محول المكرر في تطبيق الدالة search]

تذكر أن الهدف من الدالة search هو إعادة جميع الأسطر الموجودة في contents التي تحتوي على query. تستخدم هذه الشيفرة البرمجية -بصورةٍ مشابهة لمثال filter في الشيفرة 16- محوّل filter للمحافظة على السطور التي يُعيد فيها التعبير line.contains(query) القيمة true. يمكننا تجميع الأسطر الناتجة في شعاع آخر باستخدام collect.

هذه الطريقة أبسط بكثير. جرّب تنفيذ التعديلات ذاتها بحيث تستخدم توابع المكرر في الدالة search_case_insensitive بصورةٍ مشابهة.

13.4 الاختيار بين الحلقات Loops والمكررات Iterators

تعرفنا في القسم السابق على المكررات Iterators وكيفية استخدامها عمليًا ولعل السؤال الذي خطر ببالك الآن بعد قراءتهما هو: أيّ طرق التطبيق ينبغي عليك استعمالها عند برمجتك لبرنامجك ولماذا؟ التطبيق الأصلي في الشيفرة 21، أم الإصدار باستخدام المكررات في الشيفرة 22؟ يفضِّل معظم مبرمجي لغة رست استخدام

المكررات، وعلى الرغم من أن هذه الطريقة أصعب فهمًا في البداية إلا أنك ستفضلها بعد الاعتياد عليها وعلى محولات المكررات المختلفة، إذ ستركّز عندها شيفرتك البرمجية على الهدف العام للحلقة بدلًا من إضاعة الوقت في ضبط الأجزاء المختلفة من الحلقات وإنشاء أشعة جديدة. تخلّصنا هذه الطريقة من الكثير من الشيفرات البرمجية الاعتيادية بحيث يكون من الأسهل رؤية المقصد الأساسي من الشيفرة البرمجية على نحوٍ فريد لكل استخدام، مثل ترشيح كل عنصر في المكرر بحسب شرط ما.

لكن هل الطريقتان متساويتان حقًا؟ يقول الافتراض المنطقي: الحلقة التي يكون تطبيقها على مستوى منخفض أسرع. دعنا نتناقش بالتفاصيل.

13.4.1 المقارنة بين أداء الحلقات والمكررات

عليك معرفة أيّ التطبيقين أسرع، الحلقات أم المكررات؟ وذلك لتحديد متى يجب عليك استخدام أحد التطبيقين؛ هل إصدار الدالة `search` باستخدام حلقة `for` أسرع أم إصدار المكررات؟

ننفّذ اختبار أداء بكتابة كامل محتويات رواية "مغامرات شيرلوك هولمز The Adventures of Sherlock Holmes" لكاتبها سير آرثر كونان دويل Sir Arthur Conan Doyle إلى `String` والبحث عن الكلمة "the" في المحتويات. إليك نتائج اختبار الأداء على كلا الإصدارين لدالة `search` باستخدام حلقة `for`، وباستخدام المكرّرات:

```
test bench_search_for  ... bench:  19,620,300 ns/iter (+/- 915,700)
test bench_search_iter ... bench:  19,234,900 ns/iter (+/- 657,200)
```

كان إصدار المكررات أسرع قليلًا. لن نشرح الشيفرة البرمجية الخاصة باختبار الأداء هنا، لأن الهدف من الفصل ليس برهنة أن الإصدارين متساويين بالأداء بل هو لملاحظة أداء كل طريقة بالنسبة للأخرى.

ينبغي عليك التحقق من نصوص متفاوتة الطول للوسيط `contents` للحصول على اختبار أداء أكثر وضوحًا، واستخدام كلمات مختلفة من أطوال متفاوتة للوسيط `query` وتجربة مختلف أنواع الحالات. ما نريد الوصول إليه هنا هو التالي: على الرغم من أن المكررات تستخدم تطبيقًا مجرّدًا عالي المستوى، إلا أنها تُصرَّف إلى شيفرة برمجية مماثلة لشيفرة تطبيق منخفض المستوى كتبتها بنفسك، إذ أن المكررات هي من الطرق المجرّدة عديمة الحمل zero-cost abstraction في رست، وهذا يعني أن استخدام التجريد لن يؤثر على وقت التشغيل. هذا الأمر مماثل لكيفية تعريف بيارن ستروستروب Bjarne Stroustrup مصمّم لغة سي بلس بلس C++ ومطبّقها لمبدأ انعدام الحمل غير المباشر zero-overhead في كتابه "أساسيات سي بلس بلس Foundations of C++" (2012):

> تتبع تطبيقات لغة سي بلس بلس عمومًا مبدأ انعدام الحمل غير المباشر: إذ أنك لا تدفع عمّا لا تستخدمه، إضافةً إلى ذلك: لا يمكن للطريقة التي تتبعها التعامل مع الشيفرة البرمجية بطريقة أفضل من ذلك.

الشيفرة البرمجية التالية هي مثال آخر مأخوذ من برنامج فك ترميز صوت، إذ تستخدم خوارزمية فك الترميز عملية التوقع الخطي الرياضي لتوقّع القيم المستقبلية بناءً على دالة خطية تأخذ العينات السابقة. تستخدم هذه الشيفرة البرمجية سلسلة مكررات لإنجاز بعض العمليات الرياضية على ثلاثة متغيرات موجودة في النطاق scope، هي: `buffer` شريحة البيانات، ومصفوفة من 12 عنصر `coefficients` وأي مقدار إزاحة بالبيانات مُخزَّن في `qlb_shift`، وقد صرّحنا عن هذه المتغيرات داخل المثال دون إسناد أي قيمة لها. على الرغم من أن هذه الشيفرة البرمجية ليست مفيدة خارج سياقها إلا أنها مثال مختصر وواقعي على كيفية ترجمة رست للأفكار والتطبيقات عالية المستوى إلى مستوى منخفض.

```
let buffer: &mut [i32];
let coefficients: [i64; 12];
let qlp_shift: i16;

for i in 12..buffer.len() {
    let prediction = coefficients.iter()
                                 .zip(&buffer[i - 12..i])
                                 .map(|(&c, &s)| c * s as i64)
                                 .sum::<i64>() >> qlp_shift;
    let delta = buffer[i];
    buffer[i] = prediction as i32 + delta;
}
```

تمرّ الشيفرة البرمجية على القيم الاثنا عشر الموجودة في `coefficients` وتستخدم التابع `zip` لاقتران القيم الاثنا عشر الحالية مع القيم الاثني العشر السابقة الموجودة في `buffer` وذلك لحساب قيمة التوقع `prediction`، ثم نضرب قيمة كل زوج على حدى ونجمع النتيجة ونُزيح shift البتات bits في نتيجة الجمع بمقدار `qlb_shift` بت إلى اليمين.

تركّز العمليات الحسابية في تطبيقات مثل فك ترميز الصوت على الأداء في المقام الأول. نُنشئ هنا مكررًا باستخدام محوّلَين adaptor ومن ثم نستهلك القيمة. ما هي شيفرة أسيمبلي Assembly التي ستُصرَّف إليها شيفرة رست هذه؟ تُصرَّف حتى الوقت الحالي إلى نفس شيفرة أسيمبلي التي قد تكتبها بنفسك. ليس هناك أي حلقة للمكرر الخاص بالقيم `coefficients`، إذ تعرف رست أن هناك 12 تكرارًا فقط، لذا فهي تنشر الحلقة؛ وعملية النشر unrolling هي عملية لتحسين الأداء، إذ يُزال فيها الحمل الإضافي غير المباشر الخاص بالشيفرة البرمجية المُتحكمة بالحلقة وتُولَّد شيفرة برمجية متكررة بدلًا من ذلك للشيفرة البرمجية في كل تكرار من الحلقة.

تُخزَّن جميع المعاملات في المسجلات registers، مما يعني أن الوصول إلى القيم عملية سريعة، ولا يوجد هناك أي تحقق للقيود على مصفوفة الوصول array access خلال وقت التشغيل. ترفع جميع هذه الخطوات

التي تفعلها رست لتحسين الأداء من فعالية أداء الشيفرة البرمجية الناتجة كثيرًا. الآن وبعد معرفتك لهذا، يمكنك استخدام المكررات والمغلفات دون أي خوف، إذ يجعلان من شيفرتك البرمجية ذات مستوى تطبيقي أعلى دون التفريط بسرعة الأداء عند وقت التشغيل.

13.5 خاتمة

المغلفات والمكررات هي مزايا موجودة في رست ومستوحاة عن أفكار لغات البرمجة الوظيفية، وتساهم كل من المغلفات والمكررات بقدرة رست على التعبير عن أفكار عالية المستوى بأداء شيفرة برمجية مكتوبة بمستوى منخفض، إذ لا يؤثر تطبيقهما في شيفرتك البرمجية على أداء البرنامج وقت التشغيل، وهذا هو جزء من أهداف لغة رست ألا وهو السعي لتقديم تجريد دون ثمن إضافي.

الآن وبعد أن طوّرنا مشروع الدخل والخرج الخاص بنا وجعلناه أكثر تعبيرًا ووضوحًا، حان الوقت لننظر إلى بعض من مزايا كارجو Cargo التي ستساعدنا بمشاركة مشروعنا مع العالم.

دورة تطوير التطبيقات
باستخدام لغة بايثون

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

14. نظرة مفصلة عن كارجو Cargo

استخدمنا حتى الآن أبسط ميزات كارجو لبناء وتنفيذ واختبار الشيفرة البرمجية، ولكن بإمكاننا فعل أكـثر من ذلك. نناقش في هذا الفصل بعض الميزات الأخرى الأكثر تقدماً التي ستسمح لنا بفعل الآتي:

- تعديل مشروعك عن طريق ملفات الإطلاق release profiles.
- نشر مكتباتك على crates.io.
- تنظيم المشاريع الكبيرة باستخدام مساحات العمل.
- تثبيت الثنائيات binaries من crates.io.
- إضافة المزيد على عمل كارجو باستخدام أوامر مخصصة.

يستطيع كارجو فعل أكثر ممّا سنغطيه هنا، لذا ألقِ نظرةً على للتوثيق الخاص به من أجـل شـرح كامـل لكـل خصائصه.

14.1 تخصيص نسخ مشروع مع حسابات الإصدار

حسـابات الإصـدار في رسـت هي حسـابات قابلـة للتعـديل customizable ومعرّفـة مسـبقًا بالعديـد من الإعدادات المختلفة التي تسمح للمبرمج بأن يمتلـك تحكمًـا أكـبر على العديـد من الخيـارات لتصـريف الشـيفرة البرمجية، إذ أن كل حساب مضبوط بصورةٍ مستقلة عن الآخر.

لدى كارجو حسابين أساسيين، هما: حساب `dev` وهو حساب يستخدمه كارجو عندما تنفّذ `cargo build` وحساب `release` يستخدمه كارجو عندما تنفّذ `cargo build --release`. حساب `dev` معرَّف بإعدادات تصلح لعملية التطوير، وحساب `release` مُعرَّف بإعدادات تصلح لإطلاق نسخ جديدة.

قد تكون أسماء الحسابات هذه ضمن خرج نسخ مشروعك مألوفة:

```
$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
$ cargo build --release
    Finished release [optimized] target(s) in 0.0s
```

حسابات `dev` و `release` هي الحسابات التي يستخدمها المصرّف.

يمتلك كارجو إعدادات افتراضية لكلٍ من الحسابات التي تُطبَّق عندما لا تُحدد بوضوح بإضافة أية أقسام `[profile.*]` إلى ملف cargo.toml الخاص بالمشروع، إذ عند إضافة قسم `[profile.*]` لأي حساب تريد التعديل عليه، فأنت تُعيد الكتابة على أي من الإعدادات الفرعية الخاصة بالإعدادات الافتراضية. مثلاً، هذه هي القيم الأساسية لإعدادات `opt-level` لكل من الحسابين `dev` و `release`.

اسم الملف: cargo.toml

```
[profile.dev]
opt-level = 0

[profile.release]
opt-level = 3
```

تتحكم إعدادات `opt-level` بعدد التحسينات التي ستطبقها رست على شيفرتك البرمجية، بمجال من 0 إلى 3، مع الانتباه إلى أن تطبيق أي تحسينات إضافية سيزيد من وقت التصريف، لذا إذا كنت في مرحلة التطوير وتصرّف شيفرتك البرمجية بصورةٍ متكررة، فستحتاج إلى تحسينات أقل لتُصرف أسرع حتى لو كانت الشيفرة البرمجية الناتجة تعمل أبطأ. ستكون القيمة لكلٍ من `opt-level` و `dev` افتراضيًا هي 0، وعندما تكون جاهزاً لإصدار شيفرتك البرمجية، فمن الأفضل أن تقضي وقتاً أكثر بالتصريف إذ أنك ستصرّف الشيفرة البرمجية لمرة واحدة فقط في وضع الإصدار، لكنك ستشغّل البرنامج عدّة مرات، إذًا يُقايض وضع الإصدار وقت التصريف الطويل مقابل شيفرة برمجية تعمل على نحوٍ أسرع، وهذا هو السبب في كون القيمة الأساسية `opt-level` للحساب `release` هي 3.

يمكنك تجاوز القيمة الافتراضية بإضافة قيمة مختلفة لها في Cargo.toml، فإذا أردنا مثلًا أن يكون مستوى التحسين هو 1 في حساب التطوير، يمكننا إضافة هذين السطرين في ملف Cargo.toml الخاص بالمشروع:

اسم الملف: Cargo.toml

```
[profile.dev]
```

```
opt-level =1
```

تعيد هذه الشيفرة تعريف الإعداد الافتراضي 0، وعندما ننفّذ `cargo build`، سيستخدم كارجو الإعدادات الافتراضية لحساب `dev` إضافةً إلى التعديلات التي أجريناها على `opt-level`، ولأننا ضبطنا `opt-level` إلى القيمة 1، سيطبّق كارجو تحسينات أكثر من التحسينات التي يطبقها في الوضع الافتراضي، ولكن ليس أكثر من التحسينات الموجودة في إصدار البناء.

للحصول على لائحة كاملة من خيارات الضبط والقيم الافتراضية لكل حساب راجع توثيق كارجو.

14.2 نشر وحدة مصرفة crate على crates.io

استخدمنا حزم من crates.io مثل اعتماديات dependencies لمشاريعنا، لكن يمكنك أيضًا أن تشارك شيفرتك مع أشخاص آخرين عن طريق نشر حزمتك الخاصة. يوزِّع تسجيل الوحدة المُصرَّفة crates في crates.io الشيفرة المصدرية للحزم الخاصة بك، بحيث تكون الشيفرة المُستضافة مفتوحة المصدر.

لدى رست وكارجو ميزات تجعل حزمتك المنشورة أسهل إيجادًا من قبل الأشخاص واستخدامها وسنتحدث عن بعض هذه المزايا ثم سنشرح كيف تنشر حزمة.

14.2.1 كتابة تعليقات توثيق مفيدة

سيساعد توثيق حزمتك بدقة المستخدمين الآخرين على معرفة كيف ومتى يستخدمونها، لذا يُعد استثمار الوقت في كتابة التوثيق أمرًا مجديًا. ناقشنا سابقًا كيف تعلّق شيفرة رست في الفصل 3 باستخدام خطين مائلين "//". لدى رست أيضًا نوع محدد من التعليقات للتوثيق تعرف بتعليق التوثيق documentation comment والذي سيولّد بدوره توثيق HTML. يعرض الملف المكتوب باستخدام HTML محتويات تعليقات التوثيق لعناصر الواجهة البرمجية API العامة والموجهة للمبرمجين المهتمين بمعرفة كيفية استخدام وحدة مصرفة بغض النظر عن كيفية عملها وراء الكواليس.

تستخدم تعليقات التوثيق ثلاثة خطوط مائلة "///" بدلاً من خطّين، وتدعم صيغة ماركداون Markdown لتنسيق النص، إذ يكفي وضع تعليقات النص مباشرةً قبل العنصر الذي يوَثّق. تُظهر الشيفرة 1 تعليقات التوثيق لدالة `add_one` في وحدة مصرفة تدعى `my_crate`.

اسم الملف: src/lib.rs

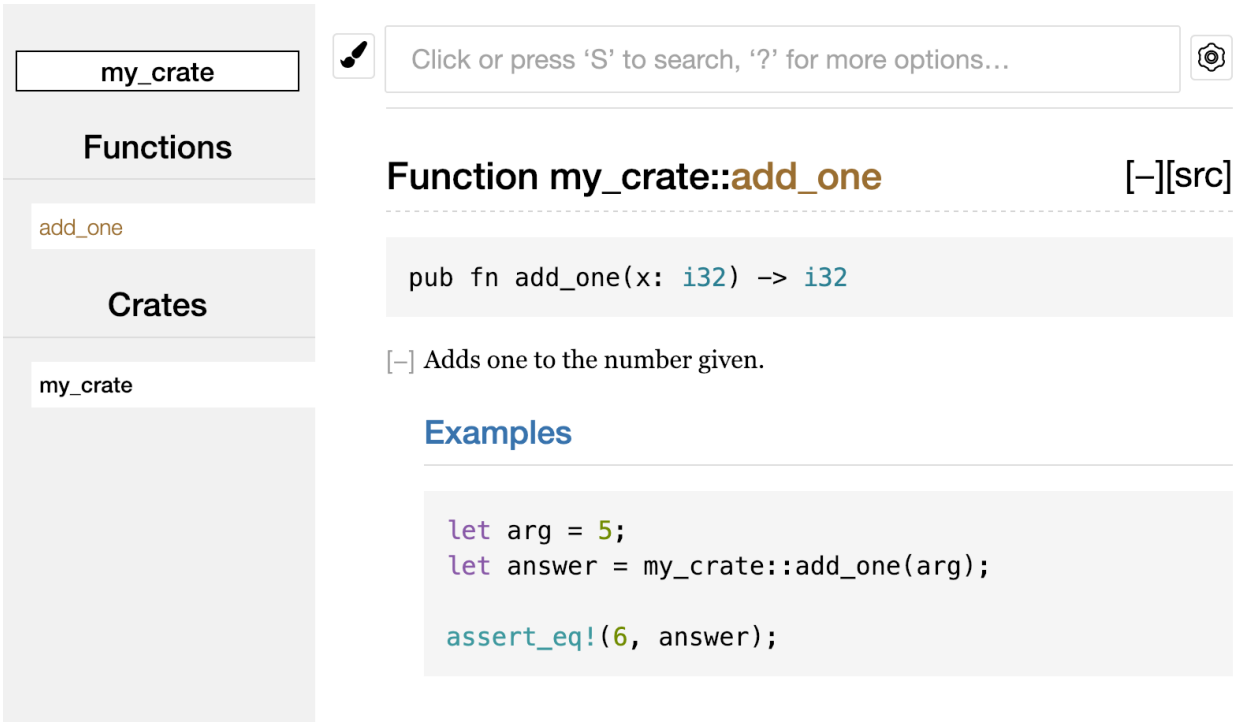
```
/// أضف واحد إلى الرقم المُعطى
///
/// ## أمثلة
///
/// ```
```

```
/// let arg = 5;
/// let answer = my_crate::add_one(arg);
///
/// assert_eq!(6, answer);
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

[الشيفرة 1- تعليق التوثيق لدالة]

نقدم هنا وصفًا عما تفعله دالة `add_one`، ابدأ القسم بعنوان `Examples` ثم ضع شيفرة تعبّر عن كيفية استخدام الدالة `add_one`، ويمكننا توليد توثيق HTML من تعليق التوثيق هذا عن طريق تنفيذ `cargo doc`، إذ يشغّل هذا الأمر أداة `rustdoc` الموزّعة مع رست وتضع توثيق HTML المولّد في المجلد "target/doc".

سيبني تنفيذ `cargo doc --open` ملف HTML للتوثيق الحالي لوحدتك المصرفة (وأيضًا توثيق كل ما يعتمد على وحدتك المصرفة) ويفتح النتيجة في متصفح ويب للسهولة. انتقل إلى الدالة `add_one` وسترى كيف يتحول النص في تعليقات التوثيق، كما يظهر في الشكل 1:



الشكل 7: توثيق HTML لدالة add_one

ا. الأقسام شائعة الاستخدام

استخدمنا عنوان ماركداون `## Examples` في الشيفرة 1 لإنشاء قسم في ملف HTML بالعنوان "Examples" وهذه بعض الأجزاء الشائعة الأخرى التي يستخدمها مؤلفو الوحدات المصرفة في توثيقهم:

- **الهلع Panics**: السيناريوهات التي تتوقف بها الدوال المُضمنة بالتوثيق، ويجب على مستخدمي الدالة الذين لا يريدون لبرامجهم أن تتوقف ألا يستدعوا الدالة في هذه الحالات.
- **الأخطاء Errors**: إذا أعادت الدالة القيمة `Result` واصفةً أنواع الأخطاء التي قد حصلت للشيفرة البرمجية المُستدعاة والظروف التي قد تسببت بحدوث الأخطاء التي تعيدها، تكون هذه المعلومات مفيدة للمستخدمين ويمكنهم بهذا أن يكتبوا شيفرة تستطيع التعامل مع الأنواع المختلفة من الأخطاء بعدة طرق.
- **الأمان Safety**: إذا استُدعيَت الدالة `unsafe` (نناقش عدم الأمان Unsafety لاحقًا)، يجب أن يكون هناك قسمٌ يشرح سبب عدم أمان الدالة ويغطي الأنواع اللا متغايرة invariants التي تتوقعها الدالة من المستخدمين.

لا تحتاج معظم تعليقات التوثيق لكل هذه الأقسام، لكن هذه لائحة جيدة تذكرك بالجوانب التي سيهتم مستخدمو شيفرتك البرمجية بمعرفتها.

ب. استخدام تعليقات التوثيق مثل اختبارات

يساعد إضافة كُتَل من الشيفرات البرمجية على أنها مثال ضمن تعليقات التوثيق فهم كيفية استخدام مكتبتك، ولفعل هذا الأمر مزايا إضافية: إذ أن تنفيذ `cargo test` سينفذ بدوره أمثلة الشيفرة البرمجية في توثيقك مثل اختبارات. لا شيء أفضل من توثيق يحتوي على أمثلة، لكن لا شيء أسوأ من أمثلة لا تعمل لأن الشيفرة البرمجية قد تغيرت منذ وقت كتابة التوثيق. نحصل على قسم في نتائج الاختبارات إذا نفّذنا الأمر `cargo test` مع توثيق دالة `add_one` من الشيفرة 1 على النحو التالي:

```
       Doc-tests my_crate

running 1 test
test src/lib.rs - add_one (line 5) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.27s
```

الآن إذا غيرنا إما الدالة أو المثال بحيث يهلع الماكرو `assert_eq!` في المثال وينفّذ `cargo test` مرةً أخرى، سنرى أن اختبارات التوثيق تلاحظ أن شيفرة المثال والشيفرة البرمجية لا يتزامنان مع بعضهما بعضًا.

ج. تعليق العناصر المحتواة

يضيف أسلوب التعليق "//!" التوثيق إلى العنصر الذي يحتوي على التعليقات بدلاً من العناصر التي تلي التعليقات، ونستخدم عادةً تعليقات المستند هذه داخل ملف الوحدة المصرفة الجذر (src/lib.rs اصطلاحًا)، أو داخل وحدة ما لتوثيق الوحدة المصرفة، أو الوحدة ككُل. على سبيل المثال، لإضافة التوثيق التي يصف الغرض من الوحدة المصرّفة `my_crate` التي تحتوي على الدالة `add_one`، نضيف تعليقات التوثيق التي تبدأ بـ "//!" إلى بداية الملف src/lib.rs، كما هو موضح في الشيفرة 2:

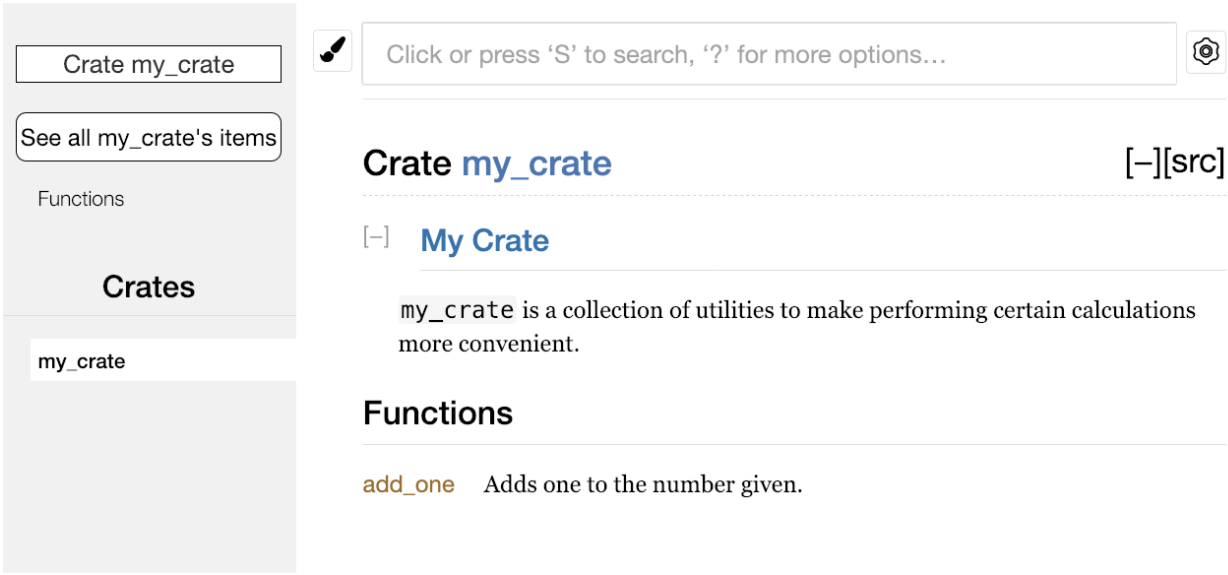
اسم الملف: src/lib.rs

```
//! ## My Crate
//!
//! `my_crate` is a collection of utilities to make performing certain
//! calculations more convenient.

/// Adds one to the number given.
// --snip--
```

[الشيفرة 2: توثيق الوحدة المصرفة `my_crate` ككل]

لاحظ عدم وجود أي شيفرة برمجية بعد آخر سطر يبدأ بـ "//!"، وذلك لأننا بدأنا التعليقات بـ "//!" بدلاً من "///". نوثّق العنصر الذي يحتوي على هذا التعليق بدلاً من العنصر الذي يتبع هذا التعليق، وفي هذه الحالة، يكون هذا العنصر هو ملف src/lib.rs، وهو الوحدة المصرفة الجذر، إذ تصف هذه التعليقات كامل الوحدة المصرفة.

عندما ننفّذ `cargo doc --open`، تظهر هذه التعليقات على الصفحة الأولى من توثيق `my_crate` أعلى قائمة العناصر العامة في الوحدة المصرفة، كما هو موضح في الشكل 2:



الشكل 8: التوثيق المولَّد للوحدة المصرّفة my_crate متضمنًا التعليق الذي يصف كل الوحدة المصرفة

تُعد تعليقات التوثيق داخل العناصر مفيدةً لوصف الوحدات المصرفة والوحدات خصوصًا. استخدمها لشرح الغرض العام من الحاوية container لمساعدة المستخدمين على فهم تنظيم الوحدة المصرفة.

14.2.2 تصدير واجهة برمجية عامة Public API ملائمة باستخدام pub use

يُعد هيكل الواجهة البرمجية العامة أحد النقاط المهمة عند نشر وحدة مصرفة، إذ يكون الأشخاص الذين يستخدمون الوحدة المصرفة الخاصة بك أقل دراية منك بهيكلية الوحدة، وقد يواجهون صعوبةً في إيجاد الأجزاء التي يريدون استخدامها إذا كانت الوحدة المصرفة الخاصة بك تحتوي على تسلسل هرمي كبير.

تناولنا سابقًا كيفية جعل العناصر عامة باستخدام الكلمة المفتاحية pub، وإضافة العناصر إلى نطاق باستخدام الكلمة المفتاحية use. ومع ذلك، قد لا تكون الهيكلية المنطقية بالنسبة لك أثناء تطوير الوحدة المصرفة مناسبةً للمستخدمين، إذ قد ترغب في تنظيم الهياكل الخاصة بك ضمن تسلسل هرمي يحتوي على مستويات متعددة، ولكن قد يواجه بعض الأشخاص مشاكل بخصوص وجود نوع ما عرّفته في مكان عميق ضمن التسلسل الهرمي وذلك عندما يرغبون باستخدامه، كما قد يسبب الاضطرار إلى إدخال المسار `use my_crate::some_module::another_module::UsefulType;` بدلاً من استخدام `use my_crate::UsefulType;` بعض الإزعاج.

الخبر السار هو أنه إذا لم يكن الهيكل مناسبًا للآخرين لاستخدامه من مكتبة أخرى، فلن تضطر إلى إعادة ترتيب التنظيم الداخلي، إذ يمكنك إعادة تصدير العناصر بدلًا من ذلك لإنشاء هيكل عام مختلف عن هيكلتك الخاصة باستخدام `pub use`. تأخذ عملية إعادة التصدير عنصرًا عامًا من مكان ما وتجعله عامًا في مكان آخر، كما لو جرى تعريفه في موقع آخر عوضًا عن ذلك.

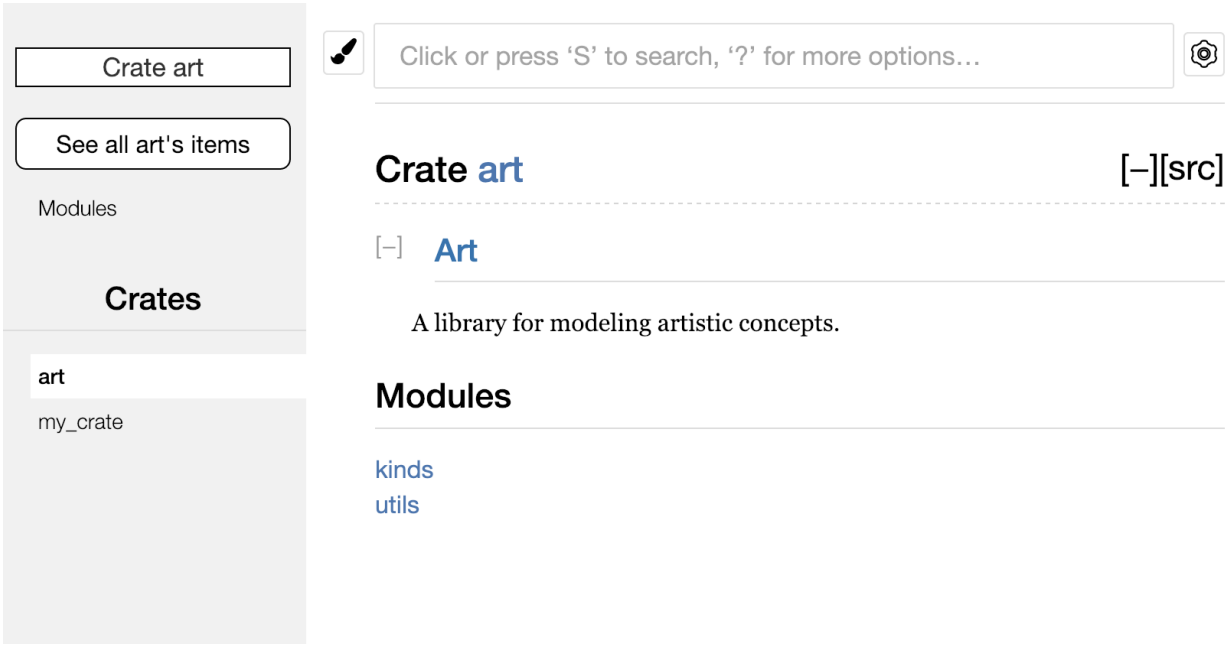
على سبيل المثال، لنفترض أننا أنشأنا مكتبة باسم `art` لنمذجة المفاهيم الفنية، بحيث يوجد داخل هذه المكتبة وحدتان: وحدة `kinds` تحتوي على معدّدَين enums باسم `PrimaryColor` و `SecondaryColor`ووحدة `utils` تحتوي على دالة تدعى `mix`، كما هو موضح في الشيفرة 3:

اسم الملف: src/lib.rs

```
//! ## Art
//!
//! مكتبة لنمذجة المفاهيم الفنية

pub mod kinds {
    ///  RYBالألوان الأساسية طبقًا لنموذج
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    ///  RYBالألوان الثانوية طبقًا لنموذج
    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    use crate::kinds::*;

    /// دمج لونين أساسيين بقيم متساوية لإنشاء لون ثانوي
    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --snip--
    }
}
```

[الشيفرة 3: مكتبة `art` التي تحتوي على عناصر منظمة ضمن الوحدتين `kinds` و `utils`]

يوضح الشكل 3 كيف ستبدو الصفحة الأولى لتوثيق الوحدة المصرفة التي أُنشئت بواسطة `cargo doc`:



الشكل 9: الصفحة الأولى لتوثيق art الذي توضّح الوحدتين kinds و utils

لاحظ أن النوعين `PrimaryColor` و `SecondaryColor` غير مُـدرجين في الصـفحة الأولى وكـذلك دالـة `mix`، إذ يجب علينا النقر على `kinds` و `utils` لرؤيتهما.

ستحتاج وحدة مصرفة أخرى تعتمد على هذه المكتبة إلى عبارات `use`، لتجلب العناصـر الموجـودة في `art` إلى النطاق، وبالتالي تحديد هيكـل الوحـدة المعرّفـة حاليًـا. تُظهـر الشـيفرة 4 مثـالاً على الوحـدة المصـرفة الـتي تستخدم عناصر `PrimaryColor` و `mix` من الوحدة المصرفة `art`:

اسم الملف: src/main.rs

```
use art::kinds::PrimaryColor;
use art::utils::mix;

fn main() {
    let red = PrimaryColor::Red;
    let yellow = PrimaryColor::Yellow;
    mix(red, yellow);
}
```

[الشيفرة 4: وحدة مصرفة تستخدم عناصر الوحدة المصرفة `art` مع تصدير هيكلها الداخلي]

ينبغي على مؤلـف الشـيفرة في الشـيفرة 4 الـتي تسـتخدم الوحـدة المصـرفة `art` أن يعـرّف أن `PrimaryColor` موجود في الوحدة `kinds` وأن `mix` موجودة في الوحدة `utils`. هيكل الوحدة المصرفة `art` مناسب أكثر للمطـورين العـاملين على الوحـدة المصـرفة `art` مقارنـةً بمن سيسـتخدمها، إذ لا يحتـوي الهيكـل

الداخلي على أي معلومات مفيدة لشخص يحاول فهم كيفية استخدام الوحدة المصرفة `art`، بل يتسبب الهيكل الداخلي باللّبس لأن المطورين الذين يستخدمونها يجب أن يعرفوا أيّ المسارات التي يجب عليهم الذهاب إليها كما يجب عليهم تحديد أسماء الوحدات في عبارات `use`.

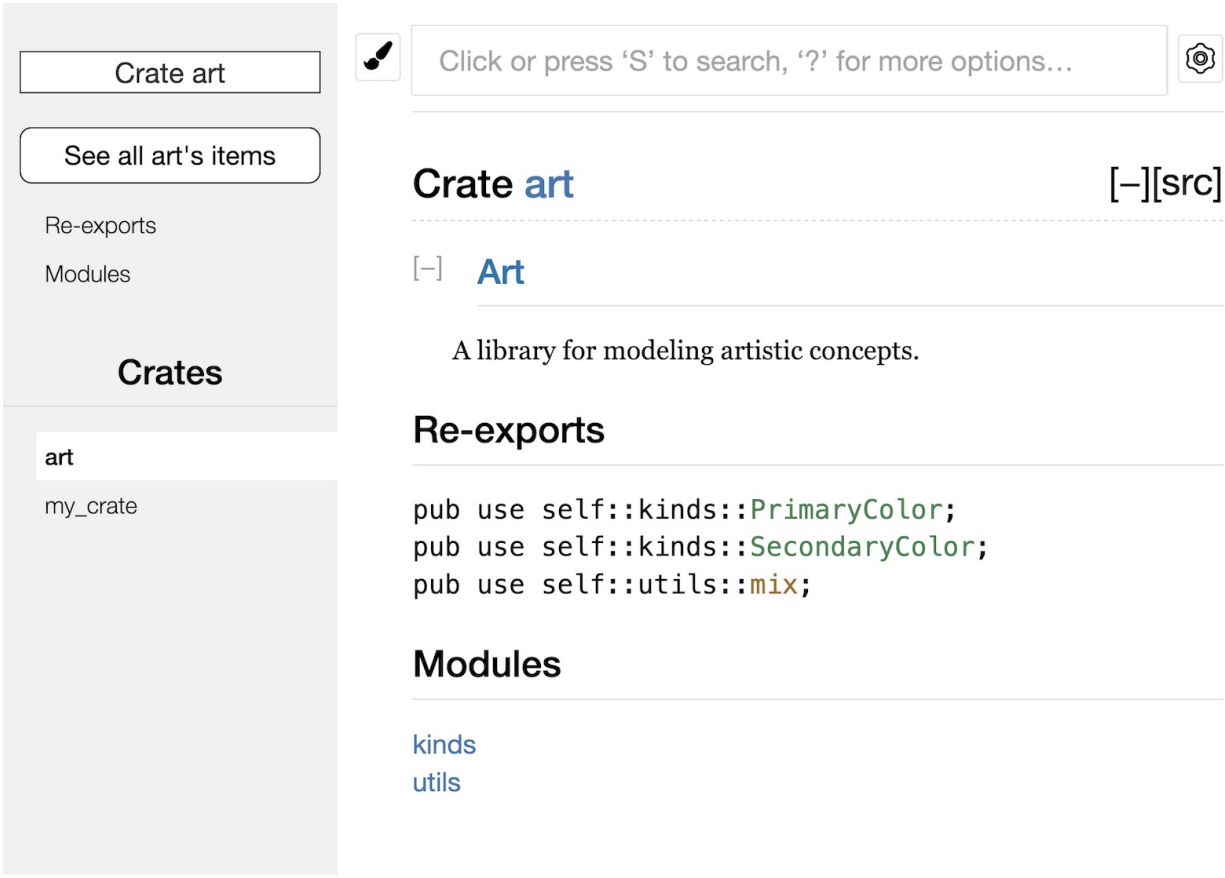
لإزالة التنظيم الداخلي من الواجهة البرمجية العامة يمكننا تعديل شيفرة الوحدة المصرفة `art` في الشيفرة 3 لإضافة تعليمات `pub use` لإعادة تصدير العناصر للمستوى العلوي كما هو موضح في الشيفرة 5:

اسم الملف: src/lib.rs

```
//! ## Art
//!
//! مكتبة لنمذجة المفاهيم الفنية

pub use self::kinds::PrimaryColor;
pub use self::kinds::SecondaryColor;
pub use self::utils::mix;

pub mod kinds {
    // --snip--
}

pub mod utils {
    // --snip--
}
```

[الشيفرة 5: إضافة تعليمات `pub use` لإعادة تصدير العناصر]

سيُدرِج توثيق الواجهة البرمجية التي يولدها الأمر `cargp doc` لهذه الوحدة المصرفة ويعيد تصديرها على الصفحة الأولى كما هو موضح في الشكل 4 جاعلًا النوعَين `PrimaryColor` و `SecondaryColor` ودالة `mix` أسهل للإيجاد.



الشكل 10: الصفحة الأولى لتوثيق art التي تعرض عمليات إعادة التصدير

يمكن لمستخدمي الوحدة المصرفة `art` أن يروا ويستخدموا الهيكلة الداخلية من الشيفرة 3 كما هو موضح في الشيفرة 4 أو يمكنهم استخدام هيكل أكثر سهولة للاستخدام في الشيفرة 5 كما هو موضح في الشيفرة 6:

اسم الملف: src/main.rs

```
use art::mix;
use art::PrimaryColor;

fn main() {
    // --snip--
}
```

[الشيفرة 6: برنامج يستخدم العناصر المعاد تصديرها من الوحدة المصرفة `art`]

يمكن -في الحالات التي يوجد فيها العديد من الوحدات المتداخلة nested modules- أن تحدث عملية إعادة تصدير الأنواع في المستوى العلوي باستخدام `pub use` فرقًا واضحًا على تجربة الأشخاص في استخدام الوحدة المصرّفة. الاستخدام الشائع الآخر للتعليمة `pub use` هو إعادة تصدير تعريفات الاعتمادية في الوحدة المصرفة الحالية لجعل تعريفات تلك الوحدة المصرفة جزءًا من الواجهة البرمجية العامة لوحدتك المصرفة.

يُعد إنشاء بنية واجهة برمجية عامة مفيدة فنًا أكثر من كونه علمًا، ويمكنك تكرار المحاولة حتى تعثر على واجهة برمجية تعمل بصورةٍ أفضل لمستخدميها، ويمنحك اختيار `pub use` مرونةً في كيفية هيكلة وحدتك المصرفة داخليًا وفصل هذه الهيكلة الداخلية عما تقدمه للمستخدمين. ألقِ نظرةً على الشيفرات البرمجية الخاصة ببعض الوحدات المصرفة التي ثبّتتها لمعرفة ما إذا كانت هيكلتها الداخلية مختلفة عن الواجهة البرمجية العامة.

14.2.3 إنشاء حساب Crates.io

قبل أن تتمكن من نشر أي وحدات مصرفة، تحتاج إلى إنشاء حساب على crates.io والحصول على رمز واجهة برمجية مميز API token، ولفعل ذلك، انقر على زر الصفحة الرئيسية على crates.io وسجّل الدخول عبر حساب غيت هب GitHub، إذ يُعد حساب غيت هب أحد المتطلبات حاليًا، ولكن قد يدعم الموقع طرقًا أخرى لإنشاء حساب في المستقبل. بمجرد تسجيل الدخول، اذهب إلى إعدادات حسابك على https://crates.io/me واسترجع مفتاح API. ثم نفّذ الأمر `cargo login` باستخدام مفتاح API الخاص بك، كما يلي:

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

يُعلم هذا الأمر كارجو برمز API الخاص بك وتخزينه محليًا في "~/.cargo/credentials". لاحظ أن هذا الرمز هو سر، فلا تشاركه مع أي شخص آخر، وإذا شاركته مع أي شخص لأي سبب من الأسباب، فيجب عليك إبطاله وإنشاء رمز مميز جديد على crates.io.

14.2.4 إضافة بيانات وصفية لوحدة مصرفة جديدة

لنفترض أن لديك وحدة مصرفة تريد نشرها، ستحتاج قبل النشر إلى إضافة بعض البيانات الوصفية في قسم `[package]` داخل ملف Cargo.toml الخاص بالوحدة المصرفة.

ستحتاج وحدتك المصرفة إلى اسم مميز، إذ يُمكنك تسمية الوحدة المصرفة أثناء عملك على وحدة مصرفة محليًا كما تريد، ومع ذلك، تُخصَّص أسماء الوحدات المصرفة على crates.io على أساس من يأتي أولًا يُخدم أولًا first-come, first-served. بمجرد اختيار اسم لوحدة مصرفة ما، لا يمكن لأي شخص آخر نشر وحدة مصرفة بهذا الاسم. قبل محاولة نشر وحدة مصرفة، ابحث عن الاسم الذي تريد استخدامه، فإذا كان الاسم مستخدمًا، ستحتاج إلى البحث عن اسم آخر وتعديل حقل `name` في ملف Cargo.toml في قسم `[package]` لاستخدام الاسم الجديد للنشر، كما يلي:

اسم الملف: Cargo.toml

```
[package]
name = "guessing_game"
```

حتى إذا اخترت اسمًا مميزًا، عند تنفيذ `cargo publish` لنشر الوحدة المصرفة في هذه المرحلة، ستتلقى تحذيرًا ثم خطأ:

```
$ cargo publish
    Updating crates.io index
warning: manifest has no description, license, license-file,
documentation, homepage or repository.
See https://doc.rust-lang.org/cargo/reference/manifest.html#package-
metadata for more info.
--snip--
error: failed to publish to registry at https://crates.io

Caused by:
  the remote server responded with an error: missing or empty metadata
fields: description, license. Please see
https://doc.rust-lang.org/cargo/reference/manifest.html for how to
upload metadata
```

تحدث هذه الأخطاء بسبب افتقاد بعض المعلومات المهمة؛ إذ أن الوصف والترخيص مطلوبان حتى يعرف الأشخاص ما تفعله الوحدة المصرفة الخاصة بك وتحت أي شروط يمكنهم استخدامها. أضف وصفًا في Cargo.toml بحيث يكون مجرد جملة أو جملتين، لأنه سيظهر مع الوحدة المصرفة الخاصة بك في نتائج البحث، أما بالنسبة لحقل `license`، فأنت بحاجة لمنح قيمة معرّف الترخيص.

تُدرج مؤسسة لينكس لتبادل بيانات حزم البرمجيات Linux Foundation's Software Package Data Exchange -أو اختصارًا SPDX- المعرّفات التي يمكنك استخدامها لهذه القيمة. على سبيل المثال، لتحديد أنك رخّصت وحدتك المصرفة باستخدام ترخيص MIT، أضف معرف `MIT`:

اسم الملف: Cargo.toml

```
[package]
name = "guessing_game"
license = "MIT"
```

إذا أردت استخدام ترخيص غير موجود في SPDX، فأنت بحاجة إلى وضع نص هذا الترخيص في ملف، وتضمين الملف في مشروعك، ثم استخدام `license-file` لتحديد اسم هذا الملف بدلاً من ذلك من استخدام المفتاح `license`.

التوجيه بشأن الترخيص المناسب لمشروعك هو خارج نطاق هذا الكتاب. يرخِّص الكثير من الأشخاص في مجتمع رست مشاريعهم بنفس طريقة رست ألا وهي باستخدام ترخيص مزدوج من "MIT OR Apache-2.0".

تدلّك هذه الممارسة على أنه بإمكانك أيضًا تحديد معرّفات ترخيص متعددة مفصولة بـ OR لتضمين تراخيص متعددة لمشروعك.

باستخدام الاسم المميز والإصدار والوصف والترخيص المضاف، أصبح ملف Cargo.toml الخاص بالمشروع جاهزًا للنشر على النحو التالي:

اسم الملف: Cargo.toml

```
[package]
name = "guessing_game"
version = "0.1.0"
edition = "2021"
description = "A fun game where you guess what number the computer has chosen."
license = "MIT OR Apache-2.0"

[dependencies]
```

يصف توثيق كارجو البيانات الوصفية الأخرى التي يمكنك تحديدها للتأكد من أن الآخرين يمكنهم اكتشاف واستخدام وحدة التصريف الخاصة بك بسهولة أكبر.

14.2.5 النشر على Crates.io

الآن وبعد أن أنشأت حسابًا، وحفظت رمز API، واخترت اسمًا للوحدة المصرفة، وحددت البيانات الوصفية المطلوبة، فأنت جاهزٌ للنشر، إذ يؤدي نشر وحدة مصرفة إلى رفع إصدار معين إلى crates.io ليستخدمه الآخرون.

كن حذرًا، لأن النشر دائم، ولا يمكن الكتابة فوق الإصدار مطلقًا، ولا يمكن حذف الشيفرة البرمجية. يتمثل أحد الأهداف الرئيسة لموقع crates.io بالعمل مثل أرشيف دائم للشيفرة البرمجية بحيث تستمر عمليات إنشاء جميع المشاريع التي تعتمد على الوحدات المصرفة من crates.io في العمل، والسماح بحذف نسخة ما سيجعل تحقيق هذا الهدف مستحيلًا، ومع ذلك، لا يوجد حد لعدد إصدارات الوحدات المصرفة التي يمكنك نشرها.

نفّذ الأمر `cargo publish` مرةً أخرى. يجب أن تنجح الآن:

```
$ cargo publish
    Updating crates.io index
   Packaging guessing_game v0.1.0 (file:///projects/guessing_game)
   Verifying guessing_game v0.1.0 (file:///projects/guessing_game)
```

```
   Compiling guessing_game v0.1.0
(file:///projects/guessing_game/target/package/guessing_game-0.1.0)
    Finished dev [unoptimized + debuginfo] target(s) in 0.19s
   Uploading guessing_game v0.1.0 (file:///projects/guessing_game)
```

تهانينا، فقد شاركت الآن الشيفرة الخاصة بك مع مجتمع رست، ويمكن لأي أحدٍ بسهولة إضافة الوحدة المصرفة الخاصة بك مثل اعتمادية لمشروعه.

14.2.6 نشر نسخة جديدة لوحدة مصرفة موجودة مسبقا

عندما تُجري تغييرات على الوحدة المصرفة الخاصة بك وتكون جاهزًا لطرح إصدار جديد، فإنك تغيّر قيمة `version` المحددة في ملف Cargo.toml الخاص بك وتعيد النشر. استخدم قواعد الإدارة الدلالية لنُسخ البرمجيات Semantic Versioning rules لتحديد رقم الإصدار التالي المناسب بناءً على التغييرات التي أجريتها، ومن ثم نفّذ `cargo publish` لرفع الإصدار الجديد.

14.2.7 تعطيل النسخ من Crates.io باستخدام cargo yank

على الرغم من أنه لا يمكنك إزالة الإصدارات السابقة للوحدة المصرفة، إلا أنه يمكنك منع أي مشاريع مستقبلية من إضافتها مثل اعتمادية جديدة، ويكون هذا مفيدًا عندما يُعطَّل إصدار الوحدة المصرفة لسبب أو لآخر، وفي مثل هذه الحالات، يدعم كارجو سحب yanking إصدار وحدة مصرفة.

يمنع سحب إصدار ما المشاريع الجديدة من الاعتماد على هذا الإصدار مع السماح لجميع المشاريع الحالية التي تعتمد عليه بالاستمرار، إذ يعني السحب أن جميع المشاريع التي تحتوي على Cargo.lock لن تتعطّل، ولن تستخدم أي ملفات Cargo.lock المستقبلية المنشأة الإصدار المسحوب.

لسحب نسخة من وحدة مصرفة، نفّذ `cargo yank` في دليل الوحدة المصرفة الذي نشرتَه سابقًا، وحدّد أي إصدار تريد إزالته. على سبيل المثال، إذا نشرنا وحدة مصرفة باسم `guessing_game` الإصدار 1.0.1 وأردنا انتزاعه، في مجلد المشروع `guessing_game` ننفّذ ما يلي:

```
$ cargo yank --vers 1.0.1
    Updating crates.io index
        Yank guessing_game@1.0.1
```

يمكنك أيضًا التراجع عن عملية السحب من خلال إضافة `--undo` إلى الأمر والسماح للمشاريع بالاعتماد على الإصدار مرة أخرى:

```
$ cargo yank --vers 1.0.1 --undo
    Updating crates.io index
```

```
        Unyank guessing_game_:1.0.1
```

لا تحذف عملية السحب أي شيفرة برمجية، إذ من غير الممكن على سبيل المثال حذف بيانات حساسة رُفعَت بالخطأ. إذا حدث ذلك، يجب عليك إعادة تعيين تلك البيانات على الفور.

14.3 مساحة عمل كارجو Cargo Workspaces

بنينا سابقًا حزمة تتضمن وحدة تنفيذية مصرفة Binary Crate ووحدة مكتبة مصرفة Library Crate، وقد تجد مع تطور مشروعك أن وحدة المكتبة المصرفة تزداد حجمًا وستحتاج إلى تقسيم حزمتك إلى عدد من وحدات مكتبة مصرفة. يقدّم كارجو Cargo ميزة تدعى مساحات العمل Workspaces التي تساعد على إدارة حزم متعددة مرتبطة تُطوَّر بالترادف tandem أي واحدًا بعد الآخر.

14.3.1 إنشاء مساحة عمل

مساحة العمل هي مجموعة من الحزم التي تتشارك ملف Cargo.lock ومجلد الخرج ذاتهما. سنستخدم شيفرة برمجية بسيطة لإنشاء مشروع باستخدام مساحة العمل، بهدف التركيز على بُنية مساحة العمل أكثر. هناك الكثير من الطرق لبناء مساحة العمل ولذا سنعمل وفق الطريقة الشائعة. سيكون لدينا مساحة عمل تحتوي على وحدة ثنائية أو تنفيذية واحدة ومكتبتين؛ إذ ستؤمن الوحدة الثنائية الوظيفة الأساسية، وستعتمد بدورها على مكتبتين: مكتبة تؤمن دالة `add_one`، والثانية ستؤمن دالة `add_two`. ستكون الوحدات المصرفة الثلاثة في مساحة العمل ذاتها. نبدأ بعمل مسار جديد لمساحة العمل:

```
$ mkdir add
$ cd add
```

ننشئ بعد ذلك في مجلد الخرج add الملف Cargo.toml الذي سيضبط مساحة العمل كاملةً. لن يكون لهذا الملف قسم `[package]`. بل سيبدأ بقسم `[workspace]` الذي سيسمح لنا بإضافة أعضاء إلى مساحة العمل عن طريق تحديد المسار للحزمة باستخدام الوحدة الثنائية أو التنفيذية المصرفة، وفي هذه الحالة المسار هو "adder":

اسم الملف: Cargo.toml

```
[workspace]

members = [
    "adder",
]
```

ننشئ وحدة ثنائية مصرفة `[adder]` عن طريق تنفيذ `cargo new` في المجلد add:

```
$ cargo new adder
     Created binary (application) `adder` package
```

يمكننا الآن بناء مساحة العمل عن طريق تشغيل `cargo build`. الملفات في مجلد add يجب أن تكون على النحو التالي:

```
├── Cargo.lock
├── Cargo.toml
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

لمساحة العمل مجلد "target" واحد في بداية المستوى الذي ستوضع فيه أدوات التخطيط artifacts المصرفة، ولا تحتوي حزمة `adder` على مجلد "target". حتى لو نفّذنا `cargo build` داخل مجلد "adder"، ستكون أدوات التخطيط المصرفة في "add/target" بدلاً من "add/adder/target". يهيّئ كارجو المجلد target بالشكل هذا لأن الحزم المصرفة في مساحة العمل مهيئة لتعتمد على بعضها بعضًا. إذا كان لكل حزمة مصرفة مجلد "target" خاص بها، فهذا يعني أن كل حزمة مصرفة ستُعيد تصريف باقي الحزم المصرفة في مساحة العمل لوضع أدوات التخطيط في مجلد "target" الخاص بها، إلا أن الحزم تتجنب عملية إعادة البناء غير الضرورية بمشاركة مجلد "target" واحد.

14.3.2 إنشاء الحزمة الثانية في مساحة العمل

دعنا ننشئ حزمة عضو ثانية في مساحة العمل ونسميها `add_one`. غيِّر ملف Cargo.toml الموجود في المستوى العلوي ليحدد المسار add_one في القائمة `members`:

اسم الملف: Cargo.toml

```
[workspace]

members = [
    "adder",
    "add_one",
]
```

أنشئ بعد ذلك حزمة مكتبة مصرفة اسمها `add_one`:

```
$ cargo new add_one --lib
     Created library `add_one` package
```

يجب أن يحتوي مجلد "add" الآن على المجلدات والملفات التالية:

```
├── Cargo.lock
├── Cargo.toml
├── add_one
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

نضيف في الملف add_one/scr/lib.rs الدالة add_one:

اسم الملف: add_one/src/lib.rs

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

الآن بإمكاننا أن نجعل كلًا من الحزمة adder والوحدة الثنائية المصرفة تعتمدان على حزمة add_one الـتي تحتوي مكتبتنا. أولاً، نضيف اعتمادية مسار add_one إلى الملف adder/Cargo.toml.

```
[dependencies]
add_one = { path = "../add_one" }
```

لا يفترض كارجو أن الحزم المصرفة تعتمد على بعضها في مساحة العمـل، لـذا نحتـاج إلى توضـيح علاقـات الاعتمادية.

لنسـتخدم بعـدها دالـة add_one (من الحزمـة المصـرفة add_one) في الحزمـة المصـرفة adder. افتح الملف adder/scr/main.rs وأضف سطر use في الأعلى لإضافة حزمة المكتبة المصـرفة add_one الجديـدة إلى النطاق. ثم عدِّل الدالة main بحيث تستدعي الدالة add_one كما في الشيفرة 7.

اسم الملف: adder/src/main.rs

```
use add_one;

fn main() {
    let num = 10;
    println!("Hello, world! {num} plus one is {}!",
add_one::add_one(num));
}
```

[الشيفرة 7: استخدام حزمة المكتبة المصرفة add_one من الحزمة adder]

دعنا نبني مساحة العمل بتنفيذ cargo build في مجلد "add" العلوي.

```
$ cargo build
   Compiling add_one v0.1.0 (file:///projects/add/add_one)
   Compiling adder v0.1.0 (file:///projects/add/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 0.68s
```

يمكننا تحديد أي حزمة نريد تشغيلها في مساحة العمل باستخدام الوسيط -p واسم الحزمة مع cargo run لتشغيل الحزمة الثنائية المصرفة من المجلد "add":

```
$ cargo run -p adder
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/adder`
Hello, world! 10 plus one is 11!
```

يشغل هذا الأمر الشيفرة الموجودة في الملف "adder/scr/main.rs"، والتي تعتمد على الحزمة المصرفة add_one.

ا. الاعتماد على حزمة خارجية في مساحة العمل

نلاحظ أن مساحة العمل تحتوي على ملف Cargo.lock واحد في المستوى الأعلى، بدلاً من أن يكون هناك ملف Cargo.lock في كل مسار حزمة مصرفة. يضمن ذلك أن كل حزمة مصرفة تستخدم الإصدار ذاته لكل الاعتماديات. إذا أضفنا حزمة rand للملفين "adder/Cargo.toml" و "add_one/Cargo.toml"، سيحوِّل كارجو كلاهما إلى إصدار واحد من rand، ثم سيسجل ذلك في Cargo.lock. جعل كل حزم المصرفة تستخدم نفس الاعتمادية يعني أن كل الحزم المصرفة ستكون متوافقة مع بعضها. دعنا نضيف الحزمة المصرفة rand إلى قسم [dependencies] في ملف add_one/Cargo.toml لكي نستخدم الحزمة المصرفة rand في الحزمة المصرفة add_one:

اسم الملف: add_one/Cargo.toml

```
[dependencies]
rand = "0.8.5"
```

يمكننا الآن إضافة `use rand;` إلى الملف add_one/src/lib.rs، وبناء كامل مساحة العمل عن طريق تنفيذ `cargo build` في المجلد "add" الذي سيجلب ويصرِّف الحزمة المصرفة `rand`. نحصل على تحذير واحد لأننا لم نُشر إلى حزمة `rand` التي أضفناها إلى النطاق:

```
$ cargo build
    Updating crates.io index
  Downloaded rand v0.8.5
   --snip--
   Compiling rand v0.8.5
   Compiling add_one v0.1.0 (file:///projects/add/add_one)
warning: unused import: `rand`
 --> add_one/src/lib.rs:1:5
  |
| use rand;
  |     ^^^^
  |
  = note: `#[warn(unused_imports)]` on by default

warning: `add_one` (lib) generated 1 warning
   Compiling adder v0.1.0 (file:///projects/add/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 10.18s
```

يحتوي ملف Cargo.lock في المستوى العلوي على معلومات عن الاعتمادية لكل من `add_one` و `rand`، ولكن وعلى الرغم من أننا نستخدم `rand` في مكان ما ضمن مساحة العمل إلا أننا لا نستطيع استخدامها في الحزم المصرفة الأخرى إلا إذا اضفنا `rand` إلى ملف Cargo.toml الخاص بها أيضاً. على سبيل المثال إذا أضفنا `use rand;` إلى ملف adder/scr/main.rs من أجل الحزمة `adder` سنحصل على خطأ:

```
$ cargo build
  --snip--
   Compiling adder v0.1.0 (file:///projects/add/adder)
error[E0432]: unresolved import `rand`
 --> adder/src/main.rs:2:5
  |
| use rand;
```

```
   |     ^^^^ no external crate `rand`
```

لحل هذه المشكلة، عدِّل ملف Cargo.toml لحزمة adder وأشِر إلى أن rand هي اعتمادية لها أيضاً. بنـاء الحزمة adder سيضيف rand إلى لائحة اعتماديات adder في ملف cargo.lock، ولكن لن يجـري أي تنزيـل لنسخ إضافية من rand. يضمن كارجو أن كل حزمة مصـرفة في كـل حزمـة في مسـاحة العمـل تسـتخدم نفس الإصدار من الحزمة rand، وبالتالي ستقل المساحة التخزينية المستخدمة وسنضمن أن كل الحزم المصــرفة في مساحة العمل ستكون متوافقة مع بعضها بعضًا.

ب. إضافة اختبار إلى مساحة العمل

سنضيف اختبارًا للدالة add_one::add_one داخل الحزمة المصرفة add_one للمزيد من التحسينات:

اسم الملف: add_one/src/lib.rs

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(3, add_one(2));
    }
}
```

نفّذ الأمر cargo test ضمن مجلد "add" العلوي، إذ سيؤدي تنفيـذ cargo test في مسـاحة عمـل مهيكلة بهذا الشكل إلى تنفيذ الاختبارات الخاصة بالحزم المصرفة في مساحة العمل:

```
$ cargo test
   Compiling add_one v0.1.0 (file:///projects/add/add_one)
   Compiling adder v0.1.0 (file:///projects/add/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.27s
     Running unittests src/lib.rs (target/debug/deps/add_one-
f0253159197f7841)
```

```
running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

     Running unittests src/main.rs (target/debug/deps/adder-
49979ff40686fa8e)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

   Doc-tests add_one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

يُظهر أول قسم في الخرج نجاح اختبار `it_works` في الحزمة المصرفة `add_one`، بينما يظهر القسم الثاني أنه لم يُعثر على أي اختبار في الحزمة المصرفة `adder`، ويظهر القسم الأخير عدم العثور على اختبارات توثيق documentation tests في الحزمة المصرفة `add_one`.

يمكن أيضاً تنفيذ اختبارات لحزمة مصرفة محددة في مساحة عمل من المجلد العلوي باستخدام الراية `-p` وتحديد اسم الحزمة المصرفة المراد اختبارها:

```
$ cargo test -p add_one
    Finished test [unoptimized + debuginfo] target(s) in 0.00s
     Running unittests src/lib.rs (target/debug/deps/add_one-
b3235fea9a156f74)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

```
   Doc-tests add_one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

يظهر الخرج أن `cargo test` نفّذ فقط الاختبارات الموجودة في الحزمة المصرفة `add_one` ولم ينفّذ الاختبارات الموجودة في الحزمة المصرفة `adder`.

إذا أردت نشر الحزم المصرفة في مساحة العمل على crates.io، فيجب على كل حزمة مصرفة في مساحة العمل أن تُنشر على حدة. نستطيع نشر حزمة مصرفة معينة في مساحة العمل باستخدام الراية `-p` وتحديد اسم الحزمة المصرفة المراد نشرها بصورةٍ مماثلة للأمر `cargo test`.

للتدرّب على العملية بصورةٍ أفضل، ضِف الحزمة المصرفة `add_two` لمساحة العمل هذه بنفس طريقة الحزمة المصرفة `add_one`.

ضع في الحسبان استخدام مساحة العمل كلما كبر مشروعك، فمن الأسهل فهم مكونات صغيرة ومنفردة على كتلة كبيرة من الشيفرة البرمجية. إضافةً إلى ذلك، إبقاء الحزم المصرفة في مساحة عمل واحدة يجعل التنسيق بين الحزم المصرفة أسهل إذا كانت تُعدَّل باستمرار في نفس الوقت.

14.4 تثبيت الملفات الثنائية binaries باستخدام cargo install

يسمح لك أمر `cargo install` بتثبيت واستخدام الوحدات الثنائية المصرفة محليًا، وليس المقصود من ذلك استبدال حزم النظام، إذ أن الأمر موجود ليكون بمثابة طريقة ملائمة لمطوري رست لتثبيت الأدوات التي شاركها الآخرون على crates.io. لاحظ أنه يمكنك فقط تثبيت الحزم التي تحتوي أهداف ثنائية binary targets، والهدف الثنائي هو برنامج قابل للتشغيل يُنشأ إذا كانت الحزمة المصرفة تحتوي على ملف src/main.rs أو ملف آخر محدد على أنه ملف تنفيذي، على عكس هدف المكتبة library target الذي لا يمكن تشغيله لوحده، فهو موجود لضمِّه داخل برامج أخرى. تحتوي الحزم المصرفة عادةً على معلومات في ملف README وتدل هذه المعلومات فيما إذا كانت الوحدة المصرفة مكتبية أو تحتوي هدفًا ثنائيًا أو كلاهما.

تُخزَّن كل الوحدات الثنائية المصرفة المثبتة عند تنفيذ `cargo install` في مجلد التثبيت الجذر الذي يدعى "bin". إذا ثبتّت رست باستخدام "rustup.rs" ولم يكن لديك أي إعدادات افتراضية فإن المجلد سيكون $HOME/.cargo/bin. تأكد أن هذا المجلد في `$PATH` الخاص بك لتكون قادراً على تشغيل البرامج التي ثبتتها باستخدام `cargo install`.

ذكرنا سابقًا أن هناك تنفيذ لأداة `grep` بلغة رست اسمه `ripgrep` للبحث عن الملفات، ولتثبيت `ripgrep` يمكنك تنفيذ الأمر التالي:

```
$ cargo install ripgrep
    Updating crates.io index
  Downloaded ripgrep v13.0.0
  Downloaded 1 crate (243.3 KB) in 0.88s
  Installing ripgrep v13.0.0
--snip--
   Compiling ripgrep v13.0.0
    Finished release [optimized + debuginfo] target(s) in 3m 10s
  Installing ~/.cargo/bin/rg
   Installed package `ripgrep v13.0.0` (executable `rg`)
```

يظهر السطر الثاني قبل الأخير من المخرجات مكان واسم الثنائية المثبتة، وهي `rg` في حالة `ripgrep`. إذا كان مجلد التثبيت موجودًا في `$PATH` الخاص بك، فيمكنك تشغيل `rg --help` والبدء باستخدام أداة أسرع مكتوبة بلغة رست للبحث عن الملفات.

14.5 توسيع استخدامات كارجو عن طريق أوامر مخصصة

كارجو مصمم بحيث يمكن توسيع استخداماته بأوامر فرعية دون الحاجة لتعديله. إذا كان هناك وحدة ثنائية binary ضمن `$PATH` اسمها `cargo-something`، فهذا يعني أنه يمكنك تشغيلها كما لو كانت أمر فرعي لكارجو عن طريق تنفيذ `cargo something`. تستطيع استعراض الأوامر المخصصة بتنفيذ `cargo --list`. قدرتك على استخدام `cargo install` لتثبيت الإضافات وتشغيلها كما في أدوات الكارجو المضمّنة built-in هي ميزة ملائمة جداً بتصميم كارجو.

14.6 خاتمة

تجعل مشاركة الشيفرة البرمجية مع كارجو و crates.io من بيئة رست مفيدةً لإنجاز العديد من المهام المختلفة. مكتبة رست القياسية صغيرة ومستقرة، ولكن الحزم المصرفة سهلة المشاركة والاستخدام والتحسين على خط زمني مختلف للخط الزمني الخاص باللغة. لا تكن خجولاً وشارك شيفرتك البرمجية المفيدة على crates.io فلربما تكون مفيدة لشخص آخر أيضًا.

دورة إدارة تطوير المنتجات

تعلم تحويل أفكارك لمنتجات ومشاريع حقيقية بدءًا من دراسة السوق
وتحليل المنافسين وحتى إطلاق منتج مميز وناجح

التحق بالدورة الآن

15. المؤشرات الذكية Smart Pointers

يُعد المؤشر pointer مفهومًا عامًا لمتغيرٍ يحتوي على عنوان في الذاكرة، ويشير هذا العنوان أو "يؤشـر إلى" بعض البيانات الأخرى. أكثر أنواع المؤشرات شيوعًا في رست هو المرجع reference، الذي تعلمناه في الفصـل الرابع. يُحدّد المرجع بالرمز "&" وتُستعار القيمة الـتي يشـير إليهـا، ولا يوجـد للمؤشـرات أي قـدرات خاصـة عـدا الإشارة إلى البيانات، ولا يتطلّب استخدامها أي حِمل إضافي overhead.

من جهة أخرى، تُعدّ المؤشرات الذكية smart pointers هياكل بيانات تعمل مثـل مؤشـر ولكن لهـا أيضًـا بيانات وصفية metadata وقدرات إضافية، إذ لا يقتصر مفهوم المؤشرات الذكية على رسـت، فهي نشـأت في لغة سي بلس بلس C++ وتوجد بلغات أخرى أيضًا. تحتوي رست على مجموعـة متنوعـة من المؤشـرات الذكيـة المعرَّفة في المكتبة القياسية التي تقدم وظائف أكثر من تلك التي توفرها المراجع، وللتعرف على المفهوم العـام، سـنلقي نظـرةً على بعض الأمثلـة المختلفـة للمؤشـرات الذكيـة، بمـا في ذلـك نـوع مؤشـر ذكي لعـدّ المراجـع reference counting. يمكّنك هذا المؤشر من السماح بوجود عدّة مـالكين owners للبيانـات من خلال تتبـع عددهم، ويُحرّر البيانات في حال لم يتبقَّ أي مالكين.

يوجد بمفهوم رست للملكية والاسـتعارة فـرقٌ إضـافي بين المراجـع والمؤشـرات الذكيـة؛ إذ بينمـا تسـتعير المراجع البيانات فقط، تمتلك المؤشرات الذكية في كثير من الحالات البيانات التي تشير إليها.

صادفنا مسبقًا بعض المؤشرات الذكية -على الرغم من أننا لم ندعوها على هذا النحو في ذلـك الـوقت- بمـا في ذلـك `String` و `Vec<T>`، وكلا النـوعين مؤشـرات ذكيـة لأنهمـا يمتلكـان بعض الـذاكرة و تسـمحان لـك بالتلاعب بها، إضافةً لوجود بيانات وصفية وإمكانيات أو ضـمانات إضـافية. تخـزّن `String` على سـبيل المثـال سعتها على أنها بيانات وصفية ولديها قدرة إضافية لتضمن أن تكون بياناتها دائمًا بترميز UTF-8 صالح.

تُطبَّق عادةً المؤشرات الذكية باستخدام الهياكل، وتنفّذ المؤشرات الذكية على عكس البنيـة العاديـة `Deref` و `Drop`، إذ تسمح سمة `Deref` لنسخة instance من هيكل المؤشر الذكي بالتصرف بمثابة مرجع حـتى تتمكن

من كتابة شيفرتك البرمجية للعمل مع المراجع أو المؤشرات الذكية، بينما تسمح لك سمة `Drop` بتخصيص الشيفرة التي تُنفَّذ عندما تخرج نسخة المؤشر الذكي عن النطاق، وسنناقش هنا كلًا من traits ونوضح سبب أهميتها للمؤشرات الذكية.

لن يغطي هذا الفصل كل مؤشر ذكي موجود بما أن نمط المؤشر الذكي smart pointer pattern هو نمط تصميم عام يستخدم بصورةٍ متكررة في رست. تمتلك العديد من المكتبات مؤشراتها الذكية الخاصة بها، ويمكنك حتى كتابة المؤشرات الخاصة بك. سنغطي المؤشرات الذكية الأكثر شيوعًا في المكتبة القياسية:

- `Box<T>` لحجز مساحة خاصة بالقيم على الكومة heap.
- `Rc<T>` نوع عدّ مرجع يمكّن الملكية المتعددة.
- `Ref<T>` و `RefMut<T>` اللذين يمكن الوصول إليهما عن طريق `RefCell<T>`، وهو نمط يفرض قواعد الاستعارة وقت التنفيذ runtime بدلًا من وقت التصريف compile time.

سنغطي بالإضافة إلى ذلك نمط قابلية التغيير الداخلي interior mutability pattern، إذ يعرّض النوع الثابت immutable واجهة برمجية لتعديل قيمة داخلية، كما سنناقش أيضًا دورات المرجع reference cycles، وسنرى كيف بإمكانها تسريب leak الذاكرة وكيفية منعها من ذلك.

دعنا نبدأ.

15.1 استخدام المؤشر Box للإشارة إلى البيانات المخزنة على الكومة

يُعد الصندوق "Box" واحدًا من أكثر المؤشرات الذكية وضوحًا وبساطةً، ويُكتب نوعه بالشكل `Box<T>`. تسمح لك الصناديق أن تخزن البيانات على الكومة بدلًا من المكدس stack، إذ يبقى المؤشر على المكدس الذي يشير بدوره للبيانات الموجودة على الكومة. عد إلى الفصل 4 لمراجعة الفرق بين الكومة والمكدس.

لا تملك الصناديق أي أفضلية في الأداء عدا أنها تخزن بياناتها على الكومة عوضًا عن المكدس، ولا تملك الكثير من الإمكانيات الإضافية. سنستخدمها غالبًا في أحد هذه الحالات:

- عندما يكون لديك نوع بحجم غير معروف وقت التصريف وتريد أن تستخدم قيمةً لهذا النوع في سياق يتطلب حجمه المحدد.
- عندما يكون لديك حجم كبير من البيانات وتريد أن تنقل ملكيتها ولكنك تريد التيقن أن البيانات لن تُنسَخ عندما تفعل هذا.
- عندما تريد أن تملك قيمةً ما وتهتم فقط أنها من نوع يناسب سمة محددة بدلًا عن كونها من نوع محدد.

سنستعرض الحالة الأولى في فقرة "تمكين الأنواع التعاودية باستخدام الصناديق"، أما في الحالة الثانية فيمكن أن يأخذ نقل ملكية كبيرة من البيانات وقتًا طويلًا وذلك لأن البيانات نُسخت على المكدس، ويمكننا تخزين الكمية الكبيرة من البيانات على الكومة في صندوق لتحسين الأداء في هذه الحالة، وبذلك تُنسخ كميةٌ صغيرةٌ من بيانات المؤشر على المكدس، بينما تبقى البيانات التي تشير إليها في مكان واحد على الكومة. تُعرف الحالة الثالثة باسم "سمة الكائن" وسنتكلم عنها لاحقًا، إذ أنك ستطبق ما تعلمته هنا لاحقًا.

15.1.1 استخدام Box لتخزين البيانات على الكومة

سنتكلم عن طريقة كتابة `Box<T>` وكيفية تفاعل هذا النوع مع القيم المخزنة داخله قبل أن نناقش حالة استخدام تخزين الكومة للنوع `Box<T>`.

توضح الشيفرة 1 كيفية استخدام صندوق لتخزين قيمة `i32` على الكومة:

اسم الملف: src/main.rs

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

[الشيفرة 1: تخزين قيمة من النوع `i32` على الكومة باستعمال صندوق box]

نعرّف المتغير `b` ليملك القيمة `Box` التي تشير إلى القيمة "5" المخزنة على الكومة. سيطبع هذا البرنامج "b = 5" وفي هذه الحالة سنصل للبيانات الموجودة في الصندوق بطريقة مشابهة في حال كانت البيانات مخزنة على المكدس. ستُحرَّر القيمة deallocated كما في أي قيمة ممتلكة، عندما يخرج صندوق عن النطاق كما تفعل `b` في نهاية `main`، وتحدث عملية التحرير لكل من الصندوق (المخزن على المكدس) والبيانات التي يشير إليها (المخزنة على الكومة).

وضع قيمة وحيدة على الكومة غير مفيد، لأنك لن تستخدم الصناديق بحد ذاتها كثيرًا، ووجود قيم مثل قيمة وحيدة من النوع `i32` على المكدس -إذ تُخزن افتراضيًا هناك- مناسبٌ أكثر في أغلب الحالات. لننظر إلى حالة تسمح لنا الصناديق أن نعرّف أنواع لن يُسمح لنا بتعريفها إن لم يكن لدينا صناديق.

15.1.2 تمكين الأنواع التعاودية باستخدام الصناديق

يمكن للقيمة من نوع تعاودي recursive type أن تملك قيمةً أخرى من النوع ذاته مثل جزء من نفسها. تمثّل الأنواع التعاودية مشكلة إذ أن رست تحتاج لمعرفة المساحة التي يحتلها نوع ما وقت التصريف، ويمكن لتداخل قيم الأنواع التعاودية نظريًا أن يستمر إلى ما لا نهاية، لهذا لا يمكن أن تعرف رست كم تحتاج القيمة من

مساحة، إلا أنه يمكننا استخدام الأنواع التعاودية بإدخال صندوق في تعريف النوع التعاودي نظـرًا لأن الصـناديق لها حجم معروف.

لنكتشف قائمة البنية "cons list" مثالًا على نوع تعاودي، إذ أن نوع البيانات هذا موجـود كثـيرًا في لغـات البرمجة الوظيفية. يُعَدّ نوع قائمة البنية بسيطًا وواضحًا باستثناء نقطة التعاود فيـه، وبالتـالي سـتكون المفـاهيم في الأمثلـة الـتي سـنعمل عليهـا مفيـدةً في أي وقت ستصـادف فيـه حـالات أكـثر تعقيـدًا من ضـمنها الأنواع التعاودية.

١. المزيد من المعلومات عن قائمة البنية

تُعد قائمة البنية هيكل بيانات أتى من لغـة البرمجـة ليسـب Lisp وشـبيهاتها، وتتـألف من أزواج متداخلـة، وهي نسخة ليسب من القائمة المترابطة linked list، ويأتي اسم هيكل البيانات هذا من الدالة `cons` (اختصارًا لدالة البنية construct function) في ليسب التي تبني بدورها زوجًا جديدًا من وسيطين arguments. يمكننا بناء قوائم بنية مؤلفة من أزواج تعاودية عن طريق استدعاء `cons` على زوج يحتوي على قيمة وزوج آخر.

إليـك المثـال التوضـيحي pseudocode لقائمـة بنيـة تحتـوي على القائمـة 1، 2، 3 مـع وجـود كـل زوج داخل قوسين:

```
(1, (2, (3, Nil)))
```

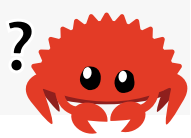
يحتوي كل عنصر في قائمة البنية على عنصرين: القيمة للعنصر الحالي والعنصر التالي، إلا أن العنصر الأخير في القائمة يحتوي فقط على قيمة تُدعى `Nil` دون عنصر تالي. يمكن إنشاء قائمة البنية عن طريق استدعاء دالة `cons` بصورة تعاودية، والاسم المتعارف عليه للدلالة على الحالة الأساسية base case للتعاودية هـو `Nil`، مـع العلم أنه ليس خاضعًا لنفس مبدأ المصطلحين "null" أو "nil" الـذين ناقشـناهما سـابقًا، فهمـا يمثلان مؤشـرًا على قيمة غير موجودة أو غير صالحة.

لا تعد قائمة بينة من هياكل البيانـات المُسـتخدمة بكـثرة في رسـت، إذ يُعـد النـوع `Vec<T>` خيـارًا أفضـل للاستعمال في معظم الوقت عندما تملك قائمة عناصر في رست، كما يوجد أنواع أخرى لبيانات تعاودية مفيـدة في حالات متعددة، لكن من خلال البدء بقائمة البنية هنـا، سـنتعرف كيـف تمكّننـا للصـناديق من تعريـف نـوع بيانات تعاودية دون ارتباك.

تتضمن الشيفرة 2 على تعريف لمعدّد enum لقائمة بنية. لاحظ أن هذه الشيفرة لن تُصرَّف بعـد لأن النـوع `List` لا يملك حجمًا محددًا وهذا ما سنوضحه لاحقًا.

اسم الملف: src/main.rs

```
enum List {
    Cons(i32, List),
```



```
    Nil,
}
```

[الشيفرة 2: المحاولة الأولى لتعريف معدّد لتمثيل هيكل البيانات قائمة البنية لقيم من النوع `i32`]

> نطبّق قائمة البنية التي تحمل فقط قيم من النوع `i32` بهدف التوضيح، إذ يمكننا تنفيذها باستعمال الأنواع المعمّمة generics كما ناقشنا في الفصل العاشر وذلك لتعريف نوع قائمة بنية يخزّن قيمًا من أي نوع.

يبدو استعمال النوع `List` لتخزين القائمة "1, 2, 3" كما توضح الشيفرة 3:

اسم الملف: src/main.rs

```
use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```

["الشيفرة 3: استعمال المعدّد `List` لتخزين القائمة "1, 2, 3]

تحمل قيمة `Cons` الأولى على "1" وقيمة `List` أخرى، قيمة `List` هذه هي قيمة `Cons` أخرى تحتوي على "2" وقيمة `List` أخرى، قيمة `List` هذه هي قيمة `Cons` أخرى تحتوي على "3" وقيمة `List` التي هي في النهاية `Nil` ألا وهو المتغاير variant غير التعاودي الذي يشير إلى نهاية القائمة.

إذا حاولنا تصريف الشيفرة البرمجية الموجودة في الشيفرة 3، فسنحصل على الخطأ الموضح في الشيفرة 4:

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0072]: recursive type `List` has infinite size
 --> src/main.rs:1:1
  |
| enum List {
  | ^^^^^^^^^ recursive type has infinite size
|     Cons(i32, List),
  |               ---- recursive without indirection
  |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make
`List` representable
```

```
  |
|     Cons(i32, Box<List>),
  |               ++++    +

For more information about this error, try `rustc --explain E0072`.
error: could not compile `cons-list` due to previous error
```

[الشيفرة 4: الخطأ الذي نحصل عليه عندما نحاول تعريف معدّد تعاودي]

يُظهِر الخطأ أن هذا النمط "له حجم لا نهائي"، والسبب هو تعريفنا للنوع `List` بمتغير تعاودي، أي أنه يحمل قيمةً أخرى لنفسه مباشرةً، ونتيجة لذلك، لا تستطيع رست معرفة مقدار المساحة التي يحتاجها لتخزين قيمة `List`. دعونا نوضح لماذا نحصل على هذا الخطأ. أولًا، سننظر إلى كيفية تحديد رست لمقدار المساحة التي يحتاجها لتخزين قيمة لنوع غير تعاودي.

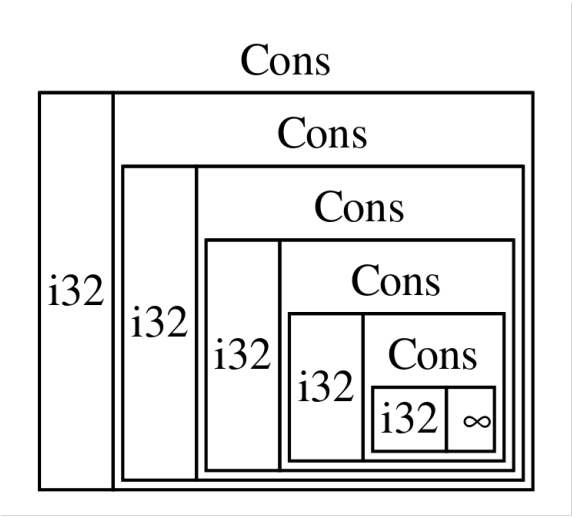
15.1.3 حساب حجم نوع غير تعاودي

تذكر معدّد `Message` الذي عرّفناه سابقًا في الفصل السادس الشيفرة 2 عندما ناقشنا تعريفات المعدّد:

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {}
```

تمر رست عبر كل من المتغيرات لمعرفة المتغير الذي يحتاج إلى أكبر مساحة وذلك لتحديد مقدار المساحة المراد تخصيصها لقيمة `Message`. ترى رست أن `Message::Quit` لا تحتاج إلى أي مساحة، بينما تحتاج `Message::Move` إلى مساحة كافية لتخزين قيمتين من نوع `i32`، وهكذا دواليك، ونظرًا لاستخدام متغير واحد فقط فإن أكبر مساحة تحتاجها قيمة `Message` هي المساحة التي ستأخذها لتخزين أكبر متغيراتها.

قارن هذا مع ما يحدث عندما تحاول رست تحديد مقدار المساحة التي يحتاجها نوع تعاودي مثل المعدد `List` في الشيفرة 2، إذ يبدأ المصرّف بالنظر إلى المتغاير `Cons` الذي يحمل قيمةً من النوع `i32` وقيمةً من النوع `List`، لذلك يحتاج `Cons` إلى مساحة مساوية لحجم النوع `i32` إضافةً إلى حجم النوع `List`. لمعرفة مقدار الذاكرة التي يحتاجها النوع `List` ينظر المصرّف إلى المتغايرات بدءًا من المتغاير `Cons`، الذي يحمل قيمةً من النوع `i32` وقيمةً من النوع `List`، وتستمر هذه العملية لما لا نهاية، كما هو موضح في الشكل 1.



الشكل 11: List لانهائية مؤلفة من متغايرات Cons لانهائية

15.1.4 استخدام Box<T> للحصول على نوع تعاودي بحجم معروف

يعطينا المصرّف الخطأ التالي لأن رست لا يمكنها معرفة مقدار المساحة المراد تخصيصها لأنواع معرّفة بصورةٍ تعاودية مرفقًا مع هذا الاقتراح المفيد:

```
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make
`List` representable
  |
|     Cons(i32, Box<List>),
  |               ++++    +
```

يعني "التحصيل indirection" -في هذا الاقتراح- أنه بدلًا من تخزين قيمة مباشرةً، يجب علينا تغيير هيكل البيانات المُستخدَم لتخزين القيمة بصورةٍ غير مباشرة عن طريق تخزين مؤشر يشير إلى القيمة عوضًا عن ذلك.

نظرًا لأن `Box<T>` هو مؤشر فإن رست تعرف دائمًا مقدار المساحة التي يحتاجها `Box<T>`، إذ أن حجم المؤشر لا يتغير بناءً على كمية البيانات التي يشير إليها، وهذا يعني أنه يمكننا وضع `Box<T>` داخل المتغاير `Cons` بدلًا من قيمة `List` أخرى مباشرةً.

سيشير `Box<T>` إلى قيمة `List` التالية التي ستكون على الكومة بدلًا من داخل المتغاير `Cons`. نظريًا، لا يزال لدينا قائمة أنشئت باستخدام قوائم تحتوي على قوائم أخرى، ولكن هذا التطبيق الآن أشبه بوضع العناصر بجانب بعضها بدلًا من وضع بعضها داخل الأخرى.

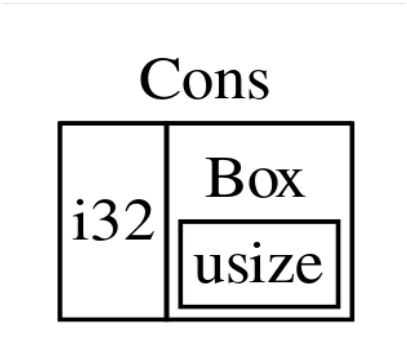
يمكننا تغيير تعريف معدّد `List` في الشيفرة 2 باستخدام `List` في الشيفرة 3 كما هو موضح في الشيفرة 5 التي ستصرَّف بنجاح:

اسم الملف: src/main.rs

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3,
Box::new(Nil))))));
}
```

[الشيفرة 5: تعريف `List` التي تستخدم `Box<T>` للحصول على حجم معروف]

يحتاج المتغاير `Cons` إلى حجم`i32` بالإضافة إلى مساحة لتخزين بيانات مؤشر الصندوق، وبما أن المتغاير `Nil` لا يخزن أي قيم فهو يحتاج إلى مساحة أقل من المتغاير `Cons`. نعلم الآن أن أي قيمة `List` ستشغل حجم `i32` إضافةً إلى حجم بيانات مؤشر الصندوق. كسرنا السلسلة اللانهائية التعاودية باستخدام الصندوق، بحيث يمكن للمصرف الآن معرفة الحجم الذي يحتاجه لتخزين قيمة `List`. يوضح الشكل 2 ما يبدو عليه متغاير `Cons` الآن.



الشكل 12: List ذات حجم محدد لأن Cons يحمل Box]

توفر الصناديق التحصيل وتخصيص الكومة فقط، إذ لا تتوافر على أي إمكانيات خاصة أخرى مثل تلك التي سنراها مع أنواع المؤشرات الذكية الأخرى لاحقًا، كما أنها لا تتمتع بالأداء العام الذي تتحمله هذه الإمكانيات الخاصة لتكون مفيدةً في حالات مثل قائمة البنية، إذ تكون ميزة التحصيل هي الميزة الوحيدة التي نحتاجها. سنلقي نظرةً على المزيد من حالات استعمال الصناديق لاحقًا في الفصل 17.

النوع `Box<T>` هو مؤشر ذكي لأنه يطبق السمة `Deref` التي تسمح لقيم `Box<T>` أن تُعامَل بمثابة مراجع. عندما تخرج قيمة `Box<T>` عن النطاق، تُمسَح بيانات الكومة التي يشير إليها الصندوق بسبب تطبيق السمة

`Drop`. ستبرز أهمية هاتان السمتان أكثر عندما نناقش أنواع المؤشرات الذكية الأخرى لاحقًا. لنكتشف هاتين السمتين بتفاصيل أكثر.

15.2 معاملة المؤشرات الذكية مثل مراجع نمطية Regular References باستخدام Deref

يسمح لك تطبيق سمة `Deref` بتخصيص سلوك عامل التحصيل dereference operator `*` (انتبه عدم الخلط مع عامل عمليات الضرب أو عامل glob) يمكننا عند تطبيق `deref` بطريقة معينة تسمح بمعاملة المؤشرات الذكية smart pointers مثل مراجع نمطية، كتابة الشيفرة البرمجية بحيث تعمل على المراجع وتُستخدم بالمؤشرات الذكية أيضًا.

لننظر أولًا إلى كيفية عمل عامل التحصيل مع المراجع النمطية regular references، ومن ثم سنحاول تعريف نوع مخصص يتصرف مثل `Box<T>`، وسنرى سبب عدم عمل عامل التحصيل مثل مرجع على النوع المخصص المعرف حديثاً. سنكتشف كيف يسمح تطبيق سمة `Deref` للمؤشرات الذكية بأن تعمل بطريقة مماثلة للمراجع، ثم سننظر إلى ميزة التحصيل القسري deref coercion في رست وكيف تسمح لنا بالعمل مع المراجع أو المؤشرات الذكية.

> هناك فرق كبير بين النوع `MyBox<T>` الذي سننشئه و `Box<T>` الحقيقي: إذ لن يخزن إصدارنا منه البيانات على الكومة heap، وسنركز في مثالنا هذا على السمة `deref`، فمكان تخزين البيانات ليس مهمًا بقدر أهمية السلوك المشابه لسلوك المؤشر.

15.2.1 تتبع المؤشر للوصول إلى القيمة

المرجع النمطي هو نوع من المؤشرات، ويمكنك التفكير بالمؤشر على أنه سهم يشير إلى قيمة مخزنة في مكان آخر. أنشأنا في الشيفرة 6 مرجعًا إلى قيمة من النوع `i32` ومن ثم استخدمنا عامل التحصيل لتتبع هذا المرجع وصولًا إلى القيمة.

اسم الملف: src/main.rs

```
fn main() {
    let x = 5;
    let y = &x;

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

[الشيفرة 6: استخدام عامل التحصيل لتتبع المرجع وصولًا إلى قيمة من النوع `i32`]

يخزّن المتغير `x` قيمةً من النوع `i32` هي 5. ضبطنا قيمة `y` بحيث تساوي مرجعًا إلى المتغير `x`، ويمكننا التأكد أن `x` تساوي 5، ولكن إذا أردنا التأكد من قيمة `y` علينا استخدام `*y` لتتبع المرجع وصولًا للقيمة التي يدل عليها لتحصيلها (هذا هو السبب في حصول عملية التحصيل على اسمها)، وذلك لكي يستطيع المصرّف أن يقارنها مع القيمة الفعلية. نستطيع الحصول على قيمة العدد الصحيح `y` بعد تحصيل `y` وهي القيمة التي تشير على ما يمكن مقارنته مع 5.

نحصل على الخطأ التالي عند التصريف إذا حاولنا كتابة `assert_eq!(5,y);`:

```
$ cargo run
   Compiling deref-example v0.1.0 (file:///projects/deref-example)
error[E0277]: can't compare `{integer}` with `&{integer}`
 --> src/main.rs:6:5
  |
|     assert_eq!(5, y);
  |     ^^^^^^^^^^^^^^^^ no implementation for `{integer} ==
&{integer}`
  |
  = help: the trait `PartialEq<&{integer}>` is not implemented for
`{integer}`
  = help: the following other types implement trait `PartialEq<Rhs>`:
            f32
            f64
            i128
            i16
            i32
            i64
            i8
            isize
          and 6 others
  = note: this error originates in the macro `assert_eq` (in Nightly
builds, run with -Z macro-backtrace for more info)

For more information about this error, try `rustc --explain E0277`.
error: could not compile `deref-example` due to previous error
```

مقارنة مرجع لرقم مع رقم غير مسموح لأنهما من نوعين مختلفين ويجب علينا استخدام عامل التحصيل لتتبع المرجع وصولًا إلى القيمة التي يشير إليها.

15.2.2 استخدام Box مثل مرجع

يمكننا إعادة كتابة الشيفرة البرمجيـة في الشـيفرة 6 باسـخدام Box<T> بـدلاً من مرجـع، وذلـك عن طريـق استخدام عامل التحصيل الذي استخدمناه على Box<T> كما هو موضـح في الشـيفرة 7 بطريقـة مماثلـة لعمـل عامل التحصيل المُستخدم في الشيفرة 6:

اسم الملف:src/main.rs

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

[الشيفرة 7: استخدام عامل التحصيل على Box<i32>]

الفرق الأساسي بين الشـيفرة 7 والشـيفرة 6 هـو أننـا حـددنا y هنـا لتكـون نسـخةً instance عن Box<T> وتشير إلى قيمة منسوخة من x بـدلًا من أن تكـون مرجعًـا يشـير إلى قيمـة x، ويمكننـا في التوكيـد assertion الأخير استخدام عامل التحصيل لتتبُّع مؤشر Box<T> بالطريقة ذاتهـا الـتي اتبعناهـا عنـدما كـان المرجـع هـو y. سنبحث تاليًا عن الشيء الذي يميّز Box<T> ليسمح لنا باستخدام عامل التحصيل بتعريف نوع خاص بنا.

15.2.3 تعريف المؤشر الذكي الخاص بنا

دعنا نبني مؤشرًا ذكيًا خاصًا بنا بصورةٍ مماثلة للنوع Box<T> الـذي تزودنـا بـه المكتبـة القياسـية لملاحظـة كيف أن المؤشرات الذكية تتصرف على نحوٍ مختلف عن المراجع افتراضيًا، وسننظر بعدها إلى كيفية إضافة قدرة استخدام عامل التحصيل.

النـوع Box<T> معـرفٌ مثـل هيكـل صـف struct tuple بعنصـر واحـد، لـذا نعـرّف في الشـيفرة 8 نـوع MyBox<T> بالطريقة ذاتها، كما نعرف أيضاً دالة new لتطابق الدالة new المعرفة في Box<T>.

اسم الملف: src/main.rs

```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
```

```
    }
}

fn main() {}
```

[الشيفرة 8: تعريف النوع MyBox<T>]

نعرف بنية بالاسم MyBox ونعرف معاملًا مُعمّمًا generic T لأننا نريد لنوعنا أن يحتفظ بكل أنواع القيم. نوع MyBox هو صف tuple بعنصر واحد من النوع T. تأخذ دالة MyBox::new معاملًا واحدًا من النوع T وتُعيد نسخةً من MyBox تحتفظ بالقيمة المُمرّرة.

لنجرب إضافة دالة main الموجودة في الشيفرة 7 إلى الشيفرة 8 ونعدّلها بحيث تستخدم النوع MyBox<T> الذي عرفناه بدلًا من Box<T>. لن تُصّرف الشيفرة 9 لأن رست لا تعرف كيفية تحصيل قيمة MyBox.

اسم الملف: src/main.rs

```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}

fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

[الشيفرة 9: محاولة استخدام Mybox<T> بطريقة استخدام المراجع و Box<T>]

إليك الخطأ التصريفي الناتج عن الشيفرة السابقة:

```
$ cargo run
```

```
   Compiling deref-example v0.1.0 (file:///projects/deref-example)
error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
  --> src/main.rs:14:19
   |
|     assert_eq!(5, *y);
   |                   ^^

For more information about this error, try `rustc --explain E0614`.
error: could not compile `deref-example` due to previous error
```

لا يمكن تحصيل النوع `MyBox<T>` الخاص بنا لأننا لم نطبـق هـذه المـيزة على نوعنـا، ولتطـبيق التحصـيل باستخدام العامل `*`، نطبّق السمة `Deref`.

15.2.4 معاملة النوع مثل مرجع بتطبيق السمة Deref

لتطبيق السمة نحن بحاجة تأمين تطبيقات لتوابع السمة المطلوبة كما ذكرنا سابقًا في الفصل 10، إذ تحتاج السمة `Deref` الموجودة في المكتبة القياسية لتطبيق تـابع واحـد اسـمه `deref` يسـتعير `self` ويعيـد مرجعًـا للبيانات الداخلية. تحتوي الشيفرة 10 على تطبيق `Deref` لإضافة تعريف `MyBox`:

اسم الملف: src/main.rs

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

```
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

[الشيفرة 10: تطبيق Deref على MyBox<T>]

نعرّف في السطر type Target = T; نوعًا مرتبط associated type لسمة Deref لتستخدمه، وتختلف الأنواع المرتبطة قليلًا في تعريف المعاملات المعمّمة، ولكن لا داعي للقلق بخصوصها حاليًا إذ سنتطرق لهذا الموضوع لاحقًا.

نكتب في متن التابع deref المرجع &self.0 بحيث يُعيد deref مرجعًا للقيمة التي نريد الوصول إليها باستخدام العامل * (تذكر سابقًا من الفصل 5 أن .0 تصل إلى القيمة الأولى في هيكل الصف). تستطيع الدالة main في الشيفرة 9 التي تستدعي * على القيمة MyBox<T> أن تُصرّف الآن مع نجاح التأكيدات.

يستطيع المصرّف أن يحصّل المراجع & فقط بدون سمة Deref، إذ يعطي تابع deref المصرف القدرة على أن يأخذ القيم من أي نوع يطبق Deref وأن يستدعي التابع deref للحصول على مرجع & يعرف كيفية تحصيله.

عندما أدخلنا *y في الشيفرة 9، نفذت رست الشيفرة التالية خلف الكواليس:

```
*(y.deref())
```

تستبدل رست العامل * باستدعاء للتابع deref ومن ثم إلى تحصيل عادي حتى لا نحتاج إلى التفكير فيما إذا كنا نريد استدعاء تابع deref، وتسمح لنا ميزة رست هذه بكتابة شيفرة برمجية تعمل بالطريقة نفسها سواءً أكان لدينا مرجع عادي أو نوع يطبق Deref.

يعود السبب في إعادة تابع deref المرجع إلى قيمة وأن التحصيل العادي خارج القوسين في *(y.dere f()) لا يزال ضروريًا إلى نظام الملكية؛ فإذا أعاد التابع deref القيمة مباشرة بدلاً من مرجع للقيمة فإن القيمة ستنتقل خارج self، ولا نريد أن نأخذ ملكية القيمة الداخلية في MyBox<T> في هذه الحالة وفي معظم الحالات التي نستخدم فيها معامل التحصيل.

لاحظ أن المعامل * يُستبدل باستدعاءٍ للتابع deref ثم استدعاء للمعامل * مرةً واحدةً وذلك في كل مرة نستخدم * في شيفرتنا البرمجية. ينتهي بنا المطاف بقيمة من النوع i32 تُطابق "5" في assert_eq! في الشيفرة 9 وذلك لأن عملية استبدال المعامل * لا تُنفّذ إلى ما لا نهاية.

15.2.5 التحصيل القسري الضمني مع الدالات والتوابع

يحوّل التحصيل القسري المرجع من نوع يطبق السمة Deref إلى مرجع من نوع آخر، فمثلاً يحوّل التحصيل القسري &String إلى &str لأن String يطبق السمة Deref بطريقة تُعيد &str. التحصيل القسري هو عملية ملائمة في رست تُجرى على وسطاء arguments الدوال والتوابع وتعمل فقط على الأنواع التي تطبق السمة Deref، وتحصل هذه العملية تلقائيًا عندما نمرر مرجعًا لقيمة ذات نوع معين مثل وسيط لدالة أو تابع لا يطابق نوع المعامل في تعريف الدالة أو التابع. تحوّل سلسلةً من الاستدعاءات إلى التابع deref النوع المُقدم إلى نوع يحتاجه المعامل.

أُضيف التحصيل القسري إلى رست بحيث لا يضطر المبرمجون الذين يكتبون استدعاءات لدالات وتوابع إلى إضافة العديد من المراجع الصريحة والتحصيلات باستخدام & و *، كما تسمح لنا خاصية التحصيل القسري أيضاً بكتابة شيفرة برمجية تعمل لكل من المراجع أو المؤشرات الذكية في الوقت ذاته.

لمشاهدة عمل التحصيل القسري عمليًا، نستخدم النوع MyBox<T> الذي عرفناه في الشيفرة 8 بالإضافة إلى تطبيق Deref الذي أضفناه في الشيفرة 10. توضح الشيفرة 11 تعريف دالة تحتوي على معامل شريحة سلسلة نصية string slice:

اسم الملف: src/main.rs

```
fn hello(name: &str) {
    println!("Hello, {name}!");
}

fn main() {}
```

[الشيفرة 11: الدالة hello التي تحتوي على معامل name من النوع &str]

بإمكاننا استدعاء الدالة hello باستخدام شريحة سلسلة نصية بمثابة وسيط مثل hello(“Rust”);. يجعل التحصيل القسري استدعاء hello مع مرجع للقيمة MyBox<String> ممكنًا كما هو موضح في الشيفرة 12:

اسم الملف: src/main.rs

```
use std::ops::Deref;
```

```
impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.0
    }
}

struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}

fn hello(name: &str) {
    println!("Hello, {name}!");
}

fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

[الشيفرة 12: استدعاء `hello` باستخدام مرجع إلى القيمة `MyBox<String>` ويمكن تنفيذ ذلك بفضل التحصيل القسري]

نستدعي هنا الدالة `hello` مع الوسيط `&m` الذي يمثـل مرجعًـا إلى القيمـة `MyBox<String>`، وتسـتطيع رسـت تحويـل `&MyBox<String>` إلى `&String` باسـتدعاء `deref` وذلـك لأننـا طبقنـا السـمة `Deref` على `MyBox<T>` كما هو موضح في الشيفرة 10. تقّدم المكتبة القياسية تطبيقًا للسـمة `Deref` على النـوع `String` الذي يعيد لنا شريحة سلسلة نصية ويمكنك العثـور على هـذه التفاصـيل في توثيـق الواجهـة البرمجيـة الخاصـة بالسـمة `Deref`. تسـتدعي رسـت التـابع `deref` مجـددًا لتحويـل `&String` إلى `&str` الـذي يطـابق تعريف دالة `hello`.

إذا لم تطبق رست التحصيل القسري فسيتوجب علينا كتابة الشيفرة 13 بدلًا من الشيفرة 12 لاستدعاء `hello` مع قيمة من النوع `&MyBox<String>`.

اسم الملف: src/main.rs

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.0
    }
}

struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}

fn hello(name: &str) {
    println!("Hello, {name}!");
}

fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&(*m)[..]);
}
```

[الشيفرة 13: الشيفرة التي يجب علينا كتابتها إذا لم تحتوي رست على ميزة التحصيل القسري]

يحصّل `(*m)` النوع `MyBox<String>` إلى `String` ومن ثم إلى `&` وتأخذ `[..]` شريحة سلسلة نصية `String` تساوي قيمة السلسلة النصية كاملةً وذلك لمطابقة بصمة الدالة `hello`. ستكون هذه الشيفرة البرمجية صعبة القراءة والفهم بدون التحصيل القسري وذلك مع كل الرموز اللازمة، إذ يسمح التحصيل القسري للغة رست بمعالجة هذه التحويلات تلقائياً نيابةً عنّا.

عندما تُعرَّف سمة `Deref` للأنواع المستخدمة، ستحلّل رست هذه الأنواع وتستخدم `Deref::deref` بعدد المرات اللازم لتحصل على مرجع يطابق نوع المعامل، ويُعرَّف عدد مرات إضافة `Deref::deref` اللازمة عند وقت التصريف لذا لا يوجد أي عبء إضافي عند وقت التشغيل runtime مع التحصيل القسري.

15.2.6 كيفية تعامل التحصيل القسري مع قابلية التغيير

يمكننا استخدام السمة `DerefMut` لتجاوز عمل العامل * على المراجع المتغيّرة mutable references بصورةٍ مشابهة لاستخدامنا لسمة `Deref` لتجاوز عمل العامل * على المراجع الثابتة immutable.

تنفّذ رست عملية التحصيل القسري عندما تجد تطبيقات لأنواع وسمات في ثلاث حالات معيّنة:

- من `&T` إلى `&U` عندما `T: Deref<Target=U>`.
- من `&mut T` إلى `&mut U` عندما `T: DerefMut<Target=U>`.
- من `&mut T` إلى `&U` عندما `T: Deref<Target=U>`.

الحالتان الأولى والثانية متماثلتان مع فرق أن الثانية هي تطبيق لحالة متغيّرة، بينما تنصّ الحالة الأولى أنه إذا كان لديك `&T` وتطبّق `T` سمة `Deref` لنوع ما من `U`، يمكن الحصول على `&U` بوضوح، والحالة الثانية تشير إلى أن عملية التحصيل القسري ذاتها تحدث للمراجع المتغيّرة.

تعدّ الحالة الثالثة أكثر تعقيدًا؛ إذ تجبر رست تحويل مرجع متغيّر إلى مرجع ثابت، إلا أن العكس غير ممكن، فالمراجع الثابتة لن تُحوّل قسريًا إلى مراجع متغيرة، وذلك بسبب قواعد الاستعارة، وإذا كان لديك مرجعًا متغيّرًا فإن هذا المرجع سيكون المرجع الوحيد لتلك البيانات (وإلا فلن يمكنك تصريف البرنامج). لن يكسرتحويل مرجع متغيّر إلى مرجع ثابت قواعد الاستعارة الافتراضية. تتطلب عملية تحويل المرجع الثابت إلى مرجع متغيّر أن يكون المرجع الثابت الابتدائي هو المرجع الوحيد الثابت للبيانات الخاصة به، إلا أن قوانين الاستعارة لا تضمن لك ذلك، وبالتالي لا يمكن لرست الافتراض بأن تحويل مرجع ثابت إلى مرجع متغيّر هي شيء ممكن.

15.3 تنفيذ شيفرة عند تحرير الذاكرة cleanup باستخدام السمة Drop

السمة الثانية المهمة لنمط المؤشرات الذكية smart pointer pattern هي السمة `Drop`، التي تسمح لك بتخصيص ماذا يحدث إذا كانت القيمة ستخرج من النطاق scope. يمكنك تأمين تنفيذ لسمة `Drop` على أي نوع، ويمكن استخدام هذه الشيفرة البرمجية لتحرير الموارد، مثل الملفات واتصالات الشبكة.

قدّمنا السمة `Drop` في سياق المؤشرات الذكية، إذ تُستخدم وظيفة سمة `Drop` دائمًا عند تطبيق مؤشر ذكي، ومثال على ذلك: عندما يُحرَّر `Box<T>`ستحرّر المساحة المخصصة له في الكومة heap التي يشير إليها الصندوق box.

يتوجب على المبرمج في بعض لغات البرمجة ولبعض الأنواع استدعاء الشيفرة البرمجية التي تحرّر مساحة تخزين أو موارد في كل مرة ينتهي من استخدام نسخة instance من هذه الأنواع، ومن الأمثلة على ذلك مقابض الملفات file handles والمقابس sockets أو الأقفال locks، وإذا نسي المبرمجون استدعاء تلك الشيفرة البرمجية (لتحرير مساحة التخزين والموارد)، سيزداد التحميل على النظام وسيتوقف النظام عن العمل بحلول نقطة معيّنة. يمكنك في لغة رست تحديد قسم معين من الشيفرة تُنفذ عندما تخرج قيمة ما من النطاق، إذ سيضيف المصرف هذه الشيفرة تلقائيًا ونتيجةً ذلك لا تحتاج أن تكون حذرًا بخصوص وضع شيفرة تحرير المساحة البرمجية cleanup code في كل مكان في البرنامج عندما تكون نسخة من نوع معين قد انتهت، أي لن يكون هناك أي هدر في الموارد.

يمكنك تحديد الشيفرة البرمجية التي تريد تنفيذها عندما تخرج قيمة ما عن النطاق وذلك باستخدام تنفيذ سمة `Drop`، إذ تحتاج سمة `Drop` أن تطبّق تابع method واحد اسمه `drop` يأخذ مرجعًا متغيّرًا إلى `self` لينتظر استدعاء رست للدالة `drop`. دعنا ننفّذ `drop` مع تعليمات `println!` في الوقت الحالي.

توضّح الشيفرة 15 البنية `CustomSmartPointer` بوظيفة مخصّصة وحيدة هي طباعة `Dropping CustomSmartPointer!` عندما تخرج النسخة عن النطاق لإظهار لحظة تنفيذ رست للخاصية `drop`.

اسم الملف: src/main.rs

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!",
self.data);
    }
}

fn main() {
    let c = CustomSmartPointer {
        data: String::from("my stuff"),
    };
    let d = CustomSmartPointer {
        data: String::from("other stuff"),
    };
```

```
        println!("CustomSmartPointers created.");
    }
```

[الشيفرة 15: بنية CustomSmartPointer التي تنفّذ السمة Drop عند مكان وضع شيفرة تحرير الذاكرة]

السمة Drop مضمّنة في المقدمة لذا لا نحتاج لأن نضيفها إلى النطاق. ننفّذ سمة Drop على CustomSmartPointer ونقدّم تنفيذًا لتابع drop الذي يستدعي بدوره println!، ونضع في متن دالة drop أي منطق نريد تنفيذه عند تخرج نسخة من النوع خارج النطاق، كما أننا نطبع نصًا هنا لنوضح كيف تستدعي رست drop بصريًا.

أنشأنا في main نسختين من CustomSmartPointer ومن ثم طبعنا CutsomSmartPointers created، سيخرج CustomSmartPointer بنهاية main خارج النطاق، مما يعني أن رست ستستدعي الشيفرة التي وضعناها في تابع drop مما سيتسبب بطباعة رسالتنا النهائية، مع ملاحظة بأننا لم نستدعي التابع drop صراحةً.

نحصل على الخرج التالي عندما ننفذ هذا البرنامج:

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.60s
     Running `target/debug/drop-example`
CustomSmartPointers created.
Dropping CustomSmartPointer with data `other stuff`!
Dropping CustomSmartPointer with data `my stuff`!
```

استدعت رست drop تلقائيًا عندما خرجت نُسخنا عن النطاق واستدعى drop بدوره الشيفرة البرمجية التي حددناها. تُحرَّر المتغيرات عكس ترتيب انشائها لذا تُحرَّر d قبل c. يهدف هذا المثال إلى منحك دليلًا بصريًا مُلاحَظ لكيفية عمل التابع drop، إذ أنك تحدّد عادةً شيفرة تحرير الذاكرة التي تحتاجها بدلاً من طباعة رسالة.

15.3.1 تحرير قيمة مبكرًا باستخدام دالة std::mem::drop

لسوء الحظ، ليس من السهل تعطيل خاصية drop التلقائية، إلا أن تعطيل drop ليس ضروريًا عادةً، إذ أن الهدف من سمة Drop هي أنها تحدث تلقائيًا. نريد أحياناً تحرير قيمة ما مبكراً ومثال على ذلك هو استخدام المؤشرات الذكية لإدارة الأقفال، إذ قد تضطر لإجبار تابع drop على تحرير القفل لتستطيع الشيفرة البرمجية الموجودة في النطاق ذاته الحصول عليه. لا تتيح لك رست استدعاء تابع drop الخاص بسمة Drop يدويًا، بل يجب عليك بدلاً من ذلك استدعاء دالة std::mem::drop المضمّنة في المكتبة القياسية، إذا أردت تحرير قيمة قسريًا قبل أن تخرج عن نطاقها.

نحصل على خطأ تصريفي إذا أردنا استدعاء التابع `drop` الخاص بسمة `drop` يدويًا وذلك بتعديل دالة `main` من الشيفرة 15 كما هو موضح:

اسم الملف: src/main.rs

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!",
self.data);
    }
}

fn main() {
    let c = CustomSmartPointer {
        data: String::from("some data"),
    };
    println!("CustomSmartPointer created.");
    c.drop();
    println!("CustomSmartPointer dropped before the end of main.");
}
```

[الشيفرة 15: محاولة استدعاء تابع `drop` من السمة `Drop` يدوياً لتحرير الذاكرة المبكر]

نحصل على الخطأ التالي عندما نحاول تصريف الشيفرة البرمجية السابقة:

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
error[E0040]: explicit use of destructor method
  --> src/main.rs:16:7
   |
|     c.drop();
   |     --^^^^--
   |     | |
   |     | explicit destructor calls not allowed
```

```
    |     help: consider using `drop` function: `drop(c)`

For more information about this error, try `rustc --explain E0040`.
error: could not compile `drop-example` due to previous error
```

تبيّن لنا رسالة الخطأ هذه أنه من غير المسموح لنا استدعاء drop صراحةً. تستخدم رسالة الخطأ المصطلح مُفكّك destructor وهو مصطلح برمجي عام لدالة تنظف نسخة ما؛ والمفكّك هو مصطلح معاكس للباني constructor وهو الذي ينشئ نسخةً ما، ودالة drop في رست هي نوع من أنواع المفكّكات.

لا تسمح لنا رست باستدعاء drop صراحةً لأن رست ستستدعي تلقائيًا التابع drop على القيمة في نهاية الدالة main مما سيسبب خطأ التحرير المزدوج double free لأن رست سيحاول تحرير القيمة ذاتها مرتين.

لا يمكننا تعطيل إدخال drop التلقائي عندما تخرج قيمة ما عن النطاق، ولا يمكننا استدعاء التابع drop صراحةً، لذا نحن بحاجة لإجبار القيمة على أن تُنظف مبكرًا باستخدام الدالة std::mem::drop.

تعمل دالة std::mem::drop بصورةٍ مختلفة عن التابع drop في سمة Drop، إذ نستدعيها بتمرير القيمة التي نريد تحريرها قسريًا مثل وسيط argument. الدالة مضمّنة في البداية لذا يمكننا تعديل الدالة main في الشيفرة 15 بحيث تستدعي الدالة drop كما في الشيفرة 16:

اسم الملف: src/main.rs

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!",
self.data);
    }
}

fn main() {
    let c = CustomSmartPointer {
        data: String::from("some data"),
    };
    println!("CustomSmartPointer created.");
    drop(c);
```

```
    println!("CustomSmartPointer dropped before the end of main.");
}
```

[الشيفرة 16: استدعاء `std::mem::drop` لتحرير القيمة صراحةً قبل الخروج من النطاق]

ينتج الخرج الآتي عن تنفيذ الشيفرة السابقة:

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.73s
     Running `target/debug/drop-example`
CustomSmartPointer created.
Dropping CustomSmartPointer with data `some data`!
CustomSmartPointer dropped before the end of main.
```

يُطبع النص "Dropping CustomSmartPointer with data some data!" بين النصين "CustomSmartpointer created." و "CustomSmartPointer dropped before the end of main."، ويدلّ ذلك إلى أن شيفرة التابع `drop` استُدعيت لتحرير `c` في تلك النقطة.

يمكنك استخدام شيفرة برمجية مخصصة في تطبيق سمة `drop` بطرق عديدة وذلك بهدف جعل عملية تحرير الذاكرة سهلة ومريحة، إذ يمكنك مثلًا استخدامها لإنشاء مُخَصص ذاكرة memory allocator خاص بك. ولا داعي لتذكر عملية تحرير الذاكرة مع سمة `Drop` وفي نظام رست للملكية وذلك لأن رست تفعل ذلك تلقائيًا، ولا داعي أيضًا للقلق من المشاكل الناتجة في حال تحرير قيم لا تزال قيد الاستخدام، إذ يضمن نظام الملكية أن المراجع صحيحة وأن `drop` يُستدعى فقط عندما تكون القيمة غير مستخدمة بعد الآن.

الآن بعد أن رأينا `Box<T>` وبعض من خصائص المؤشرات الذكية، لنرى بعض المؤشرات الذكية الأخرى المُعرفة في المكتبة القياسية.

15.4 المؤشر Rc الذكي واستخدامه للإشارة إلى عدد المراجع

مبدأ الملكية ownership واضح في معظم الحالات، إذ تسمح لك الملكية بمعرفة أي متغير يملك قيمةً ما، ولكن هناك حالات تكون فيها القيمة الواحدة مملوكةً من أكثر من مالك، فمثلًا في شعبة هيكل البيانات graph data structure قد تؤشر العديد من الأضلع edges إلى العقدة node ذاتها، وبالتالي تمتلك هذه العقدة كل الأضلع التي تشير إليها. لا يجب تحرير العقدة من الذاكرة إلا في حال عدم وجود أي ضلع يشير إليها وبالتالي عدم امتلاكها من قبل أحد.

يمكنك تمكين وجود عدة مالكين صراحةً باستخدام النوع `Rc<T>` وهو اختصار لعدّ المرجع reference counting، إذ يُحصي النوع `Rc<T>` الخاص برست عدد المراجع التي تشير إلى قيمة محددة لتحديد فيما إذا

كانت تلك القيمة قيد الاستخدام أو لا، وإذا لم يكن هناك أي مراجع تشير للقيمة، عندها يمكن تحرير القيمة دون التسبب بجعل أي مرجع غير صالح.

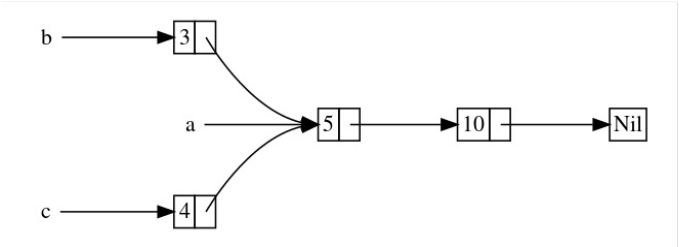
تخيل `Rc<T>` مثل تلفاز في غرفة الجلوس، فعندما يدخل شخص الغرفة ليشاهد التلفاز يشغله، كما يمكن لآخرين القدوم للغرفة والمشاهدة أيضًا، وعندما يغادر آخر شخص الغرفة يطفئ التلفاز لأنه لم يعد يُستخدم، ولكن إذا أطفأ أحدهم التلفاز بينما يشاهده الآخرون فسيغضب الذين يشاهدون التلفاز.

نستخدم النوع `Rc<T>` عندما نريد وضع بعض البيانات في الكومة heap بحيث يقرؤها عدة أجزاءٌ مختلفة من برنامجنا ولا نستطيع معرفة أي الأجزاء سينتهي من استخدام هذه البيانات أخيرًا عند تصريف البرنامج. نستطيع جعل الجزء الأخير مالك البيانات إذا كنّا نعرفه وبذلك تُطبَّق قواعد الملكية الاعتيادية لرست عند وقت التصريف.

لاحظ أن `Rc<T>` موجود فقط للاستخدام في حالات استخدام الخيط الواحد single-threaded، وسنتحدث عن عدّ المراجع في البرامج ذات الخيوط المتعددة multithreaded لاحقًا في الفصل 16 عندما نتحدث عن التزامن concurrency.

15.4.1 استخدام Rc لمشاركة البيانات

لنعاود النظر إلى قائمة البنية cons list في الشيفرة 5، تذكّر أننا عرفنا القائمة باستخدام `Box<T>` إلا أننا سنُنشئ هذه المرة لائحتين يتشاركان ملكية قائمة ثالثة كما يوضح الشكل 3.



الشكل 13: قائمة b وقائمة c يتشاركان ملكية قائمة ثالثة a

ننشئ القائمة `a` التي تحتوي على 5 وبعدها 10، ومن ثم ننشئ قائمتين؛ قائمة `b` تبدأ بالقيمة 3 وقائمة `c` تبدأ بالقيمة 4، إذ ستستمر قيم كل من القائمتين `b` و `c` ضمن القائمة الثالثة `a` التي تحتوي على 5 و10، أي ستتشارك القائمتان القائمة الأولى التي تحتوي على 5 و10.

لن تنجح محاولة تنفيذ هذه الحالة باستخدام تعريفنا للنوع `List` مع النوع `Box<T>` كما هو موضح في الشيفرة 17:

اسم الملف: src/main.rs

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

[الشيفرة 17: شيفرة توضّح أنه ليس من المسموح استخدام قائمتين النوع Box<T> مع مشاركة ملكيتهما لقائمة ثالثة]

عندما نصرّف الشيفرة السابقة نحصل على هذا الخطأ:

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0382]: use of moved value: `a`
  --> src/main.rs:11:30
   |
|     let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
   |         - move occurs because `a` has type `List`, which does not
implement the `Copy` trait
|     let b = Cons(3, Box::new(a));
   |                              - value moved here
|     let c = Cons(4, Box::new(a));
   |                              ^ value used here after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `cons-list` due to previous error
```

تمتلك متغايرات Cons البيانات التي تحتفظ بها، لذا عندما ننشئ القائمة b تنتقل القائمة a إلى القائمــة b وتمتلك القائمة b، ثم عندما نحاول استخدام a مجددًا عند إنشاء c لا يُسمح لنا، وذلك لأن القائمة a نُقلت.

يمكننا تغيير تعريف `Cons` بحيث يخزّن المراجع عوضًا عن ذلك، ولكن علينا عندها تعريف معاملات دورة حياة lifetime parameters، إذ سيستطيع بذلك عنصر في القائمة أن يعيش مدّةً تساوي مدة حياة القائمة على الأقل، وهذه هي حالة العناصر الموجودة ضمن القائمة في الشيفرة 17 ولكن هذا لا ينطبق في كل حالة.

عوضاً عن ذلك سنغير تعريف `List` لتستخدم `Rc<T>` بدلاً من `Box<T>` كما هو موضح في الشيفرة 18. يحفظ كل متغاير `Cons` قيمةً بالإضافة لمؤشر `Rc<T>` يشير إلى `List`، عندما ننشئ `b` بدلًا من أخذ ملكية `a`، سننسخ `Rc<List>` التي يحتفظ بها `a` وبذلك نزيد عدد المراجع من 1 إلى 2 ونجعل القائمة `a` والقائمة `b` تتشارك ملكية البيانات في `Rc<List>`، كما سننسخ أيضًا `a` عندما ننشئ `c`، وبذلك يزيد عدد المراجع من 2 إلى 3 مراجع، وفي كل مرة نستدعي `Rc::clone` سيزيد عدد المراجع للبيانات داخل `Rc<List>` ولن تُحرَّر البيانات إلا إذا لم يكن هناك أي مرجع يشير إليها.

اسم الملف: src/main.rs

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

[الشيفرة 18: تعريف `List` التي تستخدم `Rc<T>`]

نحتاج لإضافة تعليمة `use` وذلك لإضافة `Rc<T>` إلى النطاق لأنه غير موجود في البداية. أنشأنا في الدالة `main` قائمة تحتوي على 5 و10 وخزّناها في قيمة من نوع `Rc<List>` جديدة ضمن `a`، ومن ثمّ استدعينا الدالة `Rc::clone` بعد أن أنشأنا `b` و `c` ومررنا مرجعًا مثل وسيط argument يشير إلى `Rc<List>` في `a`.

يمكننا استدعاء `a.clone()` بدلًا من `Rc::clone(&a)`، إلا أن الطريقة الاصطلاحية في رست تستخدم `Rc::clone` في هذه الحالة. لا ينسخ تنفيذ `Rc::clone` نسخةً فعليةً deep copy للبيانات مثل باقي تنفيذات أنواع `clone` الأخرى، إذ يزيد استدعاء الدالة `Rc::clone` عدد المراجع وهو ما لا يأخذ وقتًا طويلًا، بينما يأخذ النسخ الفعلي للبيانات وقتًا طويلًا، إلا أنه يمكننا التمييز بصريًا بين النسخ الفعلي والنسخ الذي يزيد

عدد المراجع باستخدام الدالة `Rc::clone`. نحتاج لأخذ موضوع النسخ الفعلية بالحسبان عندما نبحث عن مشاكل متعلقة بأداء الشيفرة البرمجية وذلك بإهمال استدعاءات `Rc::clone`.

15.4.2 نسخ قيمة من النوع Rc يزيد عدد المراجع

لنغير مثالنا الموجود في الشيفرة 18 بحيث يمكننا رؤية تغيّر عدد المراجع عندما ننشئ ونحرّر مرجعًا إلى `Rc<T>` في `a`.

نغيّر من الدالة `main` في الشيفرة 19 بحيث تحتوي على نطاق داخلي حول القائمة `c` لكي نستطيع ملاحظة تغيّر قيمة عداد المراجع عندما تخرج `c` خارج النطاق.

اسم الملف: src/main.rs

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}",
Rc::strong_count(&a));
}
```

[الشيفرة 19: طباعة عدد المراجع]

نطبع عدد المراجع في كل نقطة يتغير فيه قيمته ضمن البرنامج، ونحصل على تلك القيمة باستدعاء الدالة `Rc::strong_count`. نسمّي الدالة `strong_count` بدلًا من `count`، وذلك لأن النوع `Rc<T>` يحتوي أيضًا على `weak_count`، وسنستخدم `weak_count` في فقرة لاحقة "منع دورات المراجع: تحويل `Rc<T>` إلى

."Weak<T>

تطبع الشيفرة السابقة ما يلي:

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.45s
     Running `target/debug/cons-list`
count after creating a = 1
count after creating b = 2
count after creating c = 3
count after c goes out of scope = 2
```

نلاحظ أن Rc<List> في a تحتوي على عدد مراجع مبدئي يساوي إلى 1، ثم يزداد العدد كل مـرة نسـتدعي فيها clone بمقدار 1، وعنـدما تخـرج c خـارج النطـاق ينقص العـدد بمقـدار 1. لسـنا بحاجـة لاسـتدعاء الدالـة لإنقاص عدد المراجع كما نفعل عند اسـتدعاء Rc::clone بهـدف زيـادة عـدد المراجـع، إذ يُنقص تنفيـذ سـمة drop من عدد المراجع تلقائيًا عندما تخرج قيمة Rc<T> من النطاق.

ما لا نستطيع رؤيته في هذا المثال هو خروج a من النطاق بعد خروج b في نهاية الدالة main، ويُضبط عدد المراجع فيما بعد إلى القيمة 0، وعندها تصـبح Rc<List> محـرَّرة تمامًـا. يسـمح اسـتخدام Rc<T> بـأن يكـون لقيمة واحدة أكثر من مالك، كما أن عدد المراجع يؤكد أن القيمة لا تزال صالحةً طالما لا يزال هناك مالك.

يسمح Rc<T> عن طريق المراجع الثابتة immutable مشـاركة البيانـات بين أقسـام متعـددة من البرنـامج للقراءة فقط. قد تخرق بعض قوانين الاستعارة -الـتي ناقشـناها سـابقًا في الفصـل الخـامس- إذا سـمح Rc<T> بوجود عدة مراجع متغيّرة أيضًا؛ إذ قد تسبب الاستعارات المتعددة المتغيّرة لنفس المكان حالة تعارض وتناقض للبيانات data race، إلا أن القدرة على تغيير البيانات مفيدة جدًا. سنناقش في القسـم القـادم النمـط الـداخلي المتغيّر interior mutability pattern، إضافةً إلى النوع RefCell<T> الـذي يمكن اسـتخدامه مـع Rc<T> لتجاوز قيد الثبات هذا.

15.5 المؤشر الذكي Refcell<T> ونمط قابلية التغيير الداخلي interior mutability

تُعد قابلية التغيير الداخلي interior mutability نمط تصميم في رست يسمح لك بتغيير البيانـات حـتى في حالة وجود مراجع ثابتة immutable تشير لتلك البيانـات، ولا تسـمح قواعـد الاسـتعارة بهـذا الإجـراء عـادةً. يستخدم النمط شيفرة "unsafe" لتغيير البيانات وذلك داخل هيكل بيانات للتحايل على قواعد رسـت المعتـادة

التي تتحكم بقابلية التعديل والاستعارة، إذ تشير الشيفرة غير الآمنة للمصرّف أننا نتحقق من القواعد يدويًا عوضًا عن الاعتماد على المصرّف للتحقق منها لنا، وسنناقش الشيفرة البرمجية غير الآمنة بالتفصيل لاحقًا.

يمكننا استخدام الأنواع التي تستخدم نمط قابلية التغيير الداخلي فقط عندما نتأكد أن قواعد الاستعارة تُتّبع في وقت التنفيذ، على الرغم من أن المصرّف لا يمكنه ضمان ذلك. تُغلّف wrap الشيفرة غير الآمنة "unsafe" ضمن واجهة برمجية API آمنة، بحيث يبقى النوع الخارجي ثابتًا.

لنكتشف هذا المفهوم من خلال النظر إلى نوع `RefCell<T>` الذي يتبع نمط التغيير الداخلي.

15.5.1 فرض قواعد الاستعارة عند وقت التنفيذ باستخدام RefCell<T>

يمثل النمط `RefCell<T>` بعكس `Rc<T>` ملكيةً مفردةً للبيانات التي يحملها. إذًا ما الذي يجعل `RefCell<T>` مختلفًا عن نمط مثل `Box<T>`؟ تذكر قواعد الاستعارة التي تعلمناها سابقًا في الفصل الرابع سابقًا:

- يمكنك بأي وقت أن تمتلك إما مرجعًا متغيرًا واحدًا أو أي عدد من المراجع الثابتة ولكن ليس كليهما.
- يجب على المراجع أن تكون صالحة دومًا.

تُفرض ثوابت قواعد الاستعارة borrowing rules' invariants عند وقت التصريف مع المراجع و `Box<T>`، وتُفرض هذه الثوابت مع `RefCell<T>` في وقت التنفيذ. نحصل على خطأ تصريفي مع المراجع إذا خرقنا هذه القواعد، إلا أنه في حالة `RefCell<T>` سيُصاب البرنامج بالهلع ويتوقف إذا خرقت القواعد ذاتها.

تتمثل إيجابيات التحقق من قواعد الاستعارة في وقت التصريف في أنه ستُكتشف الأخطاء مبكرًا في عملية التطوير ولا يوجد أي تأثير على الأداء وقت التنفيذ لأن جميع التحليلات قد اكتملت مسبقًا، لهذه الأسباب يُعد التحقق من قواعد الاستعارة في وقت التصريف هو الخيار الأفضل في معظم الحالات وهذا هو السبب في كونه الخيار الافتراضي في رست.

تتمثل إيجابيات ميزة التحقق من قواعد الاستعارة في وقت التنفيذ بدلًا من ذلك، بأنه يُسمح بعد ذلك بسيناريوهات معينة خاصة بالذاكرة الآمنة إذ يجري منعها عادةً من خلال عمليات التحقق في وقت التصريف. يُعد التحليل الساكن static analysis صارمًا كما هو الحال في مصرّف رست. من المستحيل اكتشاف بعض خصائص الشيفرة البرمجية من خلال تحليلها والمثال الأكثر شهرة هو مشكلة التوقف halting problem التي تتجاوز نطاق موضوعنا هنا إلا أنه موضوع يستحق البحث عنه.

قد ترفض رست البرنامج الصحيح إذا لم يكن مصرّف رست متأكدًا من أن الشيفرة البرمجية تتوافق مع قواعد الملكية، وذلك بفضل وجود عملية التحليل، وتوضّح هذه الحالة صرامة العملية، وإذا قبلت رست برنامجًا خاطئًا فلن يتمكن المستخدمون من الوثوق بالضمانات التي تقدمها رست، وعلى الجانب الآخر إذا رفضت رست برنامجًا صحيحًا فسيتسبب ذلك بإزعاج للمبرمج، ولكن لا يمكن أن يحدث أي شيء كارثي. يُعد النوع

`RefCell<T>` مفيدًا عندما تكون متأكدًا من أن الشيفرة الخاصة بك تتبع قواعد الاستعارة ولكن المصرّف غير قادر على فهم وضمان ذلك.

وكما في `Rc<T>` تُستخدم `RefCell<T>` فقط في السيناريوهات ذات الخيوط المفردة single-threaded وستعطيك خطأً وقت التنفيذ إذا حاولت استخدامه في سياق متعدد الخيوط multithreaded. سنتحدث عن كيفية الحصول على وظيفة `RefCell<T>` في برنامج متعدد الخيوط لاحقًا في الفصل 16.

فيما يلي ملخص لأسباب اختيار `Box<T>` أو `Rc<T>` أو `RefCell<T>`:

- يمكّن النوع `Rc<T>` وجود عدّة مالكين لنفس البيانات بينما يوجد للنوعين `RefCell<T>` و `Box<T>` مالك وحيد.
- يسمح النوع `Box<T>` بوجود استعارات متغيّرة أو ثابتة يُتحقَّق منها وقت التصريف، بينما يسمح النوع `RefCell<T>` باستعارات متغيّرة أو ثابتة وقت التنفيذ.
- بما أن النوع `RefCell<T>` يسمح باستعارات متغيّرة مُتحقق منها في وقت التنفيذ، يمكنك تغيير القيمة داخل `RefCell<T>` حتى عندما يكون النوع `RefCell<T>` ثابتًا.

تغيير القيمة داخل قيمة ثابتة هو نمط التغيير الداخلي. لنلقي نظرةً على موقف يكون فيه نمط التغيير الداخلي مفيدًا ونفحص كيف يكون ذلك ممكنًا.

15.5.2 التغيير الداخلي: استعارة متغيرة لقيمة ثابتة

عندما يكون لديك قيمة ثابتة فإنك لا تستطيع استعارتها على أنها متغيرة وفقًا لقواعد الاستعارة، على سبيل المثال، لن تُصرَّف الشيفرة البرمجية التالية:

```
fn main() {
    let x = 5;
    let y = &mut x;
}
```

إذا حاولت تصريف الشيفرة السابقة فسيظهر لك الخطأ التالي:

```
$ cargo run
   Compiling borrowing v0.1.0 (file:///projects/borrowing)
error[E0596]: cannot borrow `x` as mutable, as it is not declared as
mutable
 --> src/main.rs:3:13
  |
|     let x = 5;
```

```
  |         - help: consider changing this to be mutable: `mut x`
|     let y = &mut x;
  |             ^^^^^^ cannot borrow as mutable

For more information about this error, try `rustc --explain E0596`.
error: could not compile `borrowing` due to previous error
```

على الرغم من ذلك هناك حالات قد يكون من المفيد فيها لقيمة ما أن تغير نفسها باستخدام توابعها methods الخاصة مع جعلها ثابتة بالنسبة للشيفرات الأخرى، بحيث لا تستطيع الشيفرة البرمجية خارج توابع القيمة تغيير القيمة. يُعد استخدام `RefCell<T>` إحدى الطرق للحصول على التغيير الداخلي إلا أن النوع `RefCell<T>` لا يتغلب على كامل قواعد الاستعارة، إذ يسمح مدقق الاستعارة في المصرّف بالتغيير الداخلي ويجري التحقق من قواعد الاستعارة في وقت التنفيذ بدلًا من ذلك. إذا خُرقت القواعد فسوف تحصل على حالة هلع `panic!` بدلًا من خطأ تصريفي.

لنعمل من خلال مثال عملي يمكّننا من استخدام `RefCell<T>` لتغيير قيمة ثابتة ومعرفة لماذا يُعد ذلك مفيدًا.

ا. حالة استخدام للتغيير الداخلي: الكائنات الزائفة Mock Objects

يستخدم المبرمج نوعًا بدلًا من آخر في بعض الأحيان أثناء الاختبار من أجل مراقبة سلوك معين والتأكد من تنفيذه بصورةٍ صحيحة، ويُسمى هذا النوع من وضع القيمة المؤقتة placeholder بالاختبار المزدوج test double. فكِّر بهذا الأمر بسياق "دوبلير stunt double" في صناعة الأفلام، إذ يتدخل الشخص ويحل محل الممثل لإنجاز مشهد معين صعب. يُستخدم الاختبار المزدوج لأنواع أخرى عندما نجري الاختبارات. الكائنات الزائفة هي أنواع محددة من الاختبار المزدوج التي تسجل ما يحدث أثناء الاختبار حتى تتمكن من أن تتأكد أن الإجراءات الصحيحة قد أُنجزت.

لا تحتوي رست على كائنات بنفس معنى الكائنات في لغات البرمجة الأخرى، ولا تحتوي رست على قدرة التعامل مع الكائنات الزائفة مُضمَّنة في المكتبة القياسية كما تفعل بعض اللغات الأخرى، ومع ذلك يمكنك بالتأكيد إنشاء هيكل يخدم الهدف من الكائنات الزائفة.

إليك السيناريو الذي سنختبره: سننشئ مكتبةً تتعقب قيمةً مقابل قيمة قصوى وترسل رسائل بناءً على مدى قرب القيمة الحالية من القيمة القصوى، بحيث يمكن استخدام هذه المكتبة لتتبع حصة المستخدم لعدد استدعاءات الواجهة البرمجية المسموح بإجرائها على سبيل المثال.

ستوفر مكتبتنا فقط وظيفة تتبع مدى قرب الحد الأعظمي للقيمة وما يجب أن تكون عليه الرسائل في أي وقت، ومن المتوقع أن توفّر التطبيقات التي تستخدم مكتبتنا آلية إرسال الرسائل، يمكن للتطبيق وضع رسالة

في التطبيق، أو إرسال بريد إلكتروني، أو إرسال رسالة نصية، أو شيء آخر، إذ لا تحتـاج المكتبـة إلى معرفـة هـذا التفصيل، وكل ما تحتاجه هو شيء ينفِّذ سمة سنوفّرها تدعى `Messenger`.

تظهر الشيفرة 20 الشيفرة البرمجية الخاصة بالمكتبة:

اسم الملف: src/lib.rs

```
pub trait Messenger {
    fn send(&self, msg: &str);
}

pub struct LimitTracker<'a, T: Messenger> {
    messenger: &'a T,
    value: usize,
    max: usize,
}

impl<'a, T> LimitTracker<'a, T>
where
    T: Messenger,
{
    pub fn new(messenger: &'a T, max: usize) -> LimitTracker<'a, T> {
        LimitTracker {
            messenger,
            value: 0,
            max,
        }
    }

    pub fn set_value(&mut self, value: usize) {
        self.value = value;

        let percentage_of_max = self.value as f64 / self.max as f64;

        if percentage_of_max >= 1.0 {
            self.messenger.send("Error: You are over your quota!");
        } else if percentage_of_max >= 0.9 {
```

```
            self.messenger
                .send("Urgent warning: You've used up over 90% of your
quota!");
        } else if percentage_of_max >= 0.75 {
            self.messenger
                .send("Warning: You've used up over 75% of your
quota!");
        }
    }
}
```

[الشيفرة 20: مكتبة لتتبع مدى قرب قيمة من قيمة العظمى وإرسال تحذير عندما تصل القيمة لمقدار معيّن]

أحد الأجزاء المهمة من هذه الشيفرة هو أن سمة `Messenger` لها تابعٌ واحدٌ يدعى `send` يقبل مرجعًا ثابتًا يشير إلى `self` بالإضافة إلى نص الرسالة، وهذه السمة هي الواجهة التي يحتاج الكائن الزائف الخاص بنا إلى تنفيذها بحيث يمكن استخدام الكائن الزائف بنفس الطريقة التي يُستخدم بها الكائن الحقيقي؛ والجزء المهم الآخر هو أننا نريد اختبار سلوك التابع `set_value` على `LimitTracker`. يمكننا تغيير ما نمرّره لمعامل `value`، إلا أن `set_value` لا تُعيد لنا أي شيء لإجراء تأكيدات assertions عليه؛ إذ نريد أن نكون قادرين على القول أنه على المُرسل أن يرسل رسالة معيّنة إذا أنشأنا `LimitTracker` بشيء يطبّق السمة `Messenger` وقيمة معينة لـ `max`، عندما نمرر أرقامًا مختلفة مثل قيمة `value`.

نحتاج هنا إلى كائن زائف لتتبُّع الرسائل التي يُطلب منه إرسالها عندما نستدعي `send`، وذلك بدلًا من إرسال بريد إلكتروني أو رسالة نصية. يمكننا إنشاء نسخة جديدة من كائن زائف وإنشاء `LimitTracker` يستخدم الكائن الزائف واستدعاء التابع `set_value` على `LimitTracker` ثم التحقق من أن الكائن الزائف يحتوي على الرسائل التي نتوقعها. تُظهر الشيفرة 21 محاولة تطبيق كائن زائف لفعل ذلك فقط إلا أن مدقق الاستعارة لن يسمح بذلك.

اسم الملف: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    struct MockMessenger {
        sent_messages: Vec<String>,
    }
```

```
    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: vec![],
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        let mock_messenger = MockMessenger::new();
        let mut limit_tracker = LimitTracker::new(&mock_messenger,
100);

        limit_tracker.set_value(80);

        assert_eq!(mock_messenger.sent_messages.len(), 1);
    }
}
```

[الشيفرة 21: محاولة لتطبيق `MockMessenger` غير المسموح به بواسطة مدقق الاستعارة]

تعرِّف شيفرة الاختبار السابقة هيكل `MockMessenger` يحتوي على حقل `sent_messages` مع قيم `Vec` من نوع سلسلة نصية `String` لتتبع الرسائل المطلوب إرسالها؛ كما نعرّف أيضًا دالة مرتبطة associated function بالاسم `new` لتسهيل إنشاء قيم `MockMessenger` الجديدة التي تبدأ بلائحة فارغة من الرسائل، ثم نطبّق السمة `Messenger` على `MockMessenger` حتى نستطيع إعطاء ملكية لنسخة `MockMessenger`. نأخذ الرسالة الممرّرة مثل معامل في تعريف التابع `send` ونخزّنها في قائمة `sent_messages` الخاصة بالحقل `MockMessenger`.

نختبر في الاختبار ما يحدث عندما يُطلب من `LimitTracker` تعيين `value` لشيء يزيد عن 75 بالمائة من `max`، إذ نُنشئ أولًا نسخةً جديدةً من `MockMessenger` تبدأ بقائمة فارغة من الرسائل، ثم نُنشئ

`LimitTracker` جديد ونمرّر له مرجعًا يشير إلى `MockMessenger` الجديد وقيمة `max` هي 100. نستدعي التابع `set_value` على `LimitTracker` بقيمة 80 والتي تزيد عن 75 بالمئة من 100 ثم نتأكد أن قائمة الرسائل التي يتتبعها `MockMessenger` يجب أن تحتوي الآن على رسالة واحدة.

ومع ذلك هناك مشكلة واحدة في هذا الاختبار كما هو موضح هنا:

```
$ cargo test
   Compiling limit-tracker v0.1.0 (file:///projects/limit-tracker)
error[E0596]: cannot borrow `self.sent_messages` as mutable, as it is
behind a `&` reference
  --> src/lib.rs:58:13
   |
|     fn send(&self, msg: &str);
   |             ----- help: consider changing that to be a mutable
reference: `&mut self`
...
|             self.sent_messages.push(String::from(message));
   |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `self`
is a `&` reference, so the data it refers to cannot be borrowed as
mutable

For more information about this error, try `rustc --explain E0596`.
error: could not compile `limit-tracker` due to previous error
warning: build failed, waiting for other jobs to finish...
```

لا يمكننا تعديل `MockMessenger` لتتبع الرسائل لأن التابع `send` يأخذ مرجعًا ثابتًا يشير إلى `self`، كما لا يمكننا أيضًا تنفيذ الاقتراح في نص الخطأ الذي يدلنا على استخدام `&mut self` بدلًا من ذلك لأن بصمة التابع `send` لن تتطابق مع بصمة تعريف السمة `Messenger` (ننصحك بقراءة رسالة الخطأ بنفسك).

هذه هي الحالة التي يُمكن أن يساعد فيها نمط التغيير الداخلي، إذ سنخزن `sent_messages` داخل `RefCell<T>`، ومن ثم سيتمكن التابع `send` من تعديل `sent_messages` لتخزين الرسائل التي رأيناها، وتظهر الشيفرة 22 كيف يبدو التغيير:

اسم الملف: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;
    use std::cell::RefCell;
```

```
    struct MockMessenger {
        sent_messages: RefCell<Vec<String>>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: RefCell::new(vec![]),
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
self.sent_messages.borrow_mut().push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        // --snip--

        assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
    }
}
```

[الشيفرة 22: استخدام `RefCell<T>` لتغيير قيمة داخلية بينما تكون القيمة الخارجية ثابتة]

أصبح حقل `sent_messages` الآن من النـوع `RefCell<Vec<String>>` بـدلًا من `Vec <String>`. نُنشئ في الدالة `new` نسخةً جديدةً من `RefCell<Vec<String>>` حول الشعاع الفارغ.

لا يزال المعامل الأول يمثّل استعارة `self` ثابتة تتطابق مع تعريف السمة، ولتطبيق التابع `send` نسـتدعي `borrow_mut` على `RefCell<Vec<String>>` في `self.sent_messages` للحصول على مرجـع متغيّـر للقيمة داخل `RefCell<Vec<String>>` التي تمثّل الشعاع، ومن ثم يمكننا أن نسـتدعي `push` على المرجـع المتغيّر الذي يشير إلى الشعاع لتتبع الرسائل المرسلة أثناء الاختبار.

التغيير الأخير الذي يتعين علينا إجراؤه هو ضمن التأكيد assertion، وهو لمعرفة عدد العناصر الموجودة في الشعاع الداخلي، ولتحقيق ذلك نستدعي `borrow` على `RefCell<Vec<String>>` للحصول على مرجع ثابت يشير إلى الشعاع.

الآن بعد أن رأيت كيفية استخدام المؤشر `RefCell<T>` لنتعمق في كيفية عمله.

ب. تتبع عمليات الاستعارة وقت التنفيذ عن طريق `RefCell<T>`

نستخدم عند إنشاء مراجع متغيّرة وثابتة الصيغة `&` و `& mut` على التوالي، بينما نستخدم في `RefCell<T>` التابعين `borrow` و `borrow_mut` اللذين يعدّان جزءًا من واجهة برمجية آمنة تنتمي إلى `RefCell<T>`. يُعيد التابع `borrow` نوع المؤشر الذكي `Ref <T>` ويُعيد التابع `borrow_mut` نوع المؤشر الذكي `RefMut<T>`، وينفّذ كلا النوعين السمة `Deref` لذلك يمكننا معاملتهما على أنهما مراجع نمطيّة regular references.

يتعقّب المؤشر `RefCell<T>` عدد المؤشرات الذكية النشطة حاليًا من النوعين `Ref<T>` و `RefMut<T>`، وفي كل مرة نستدعي فيها التابع `borrow` يزيد المؤشر `RefCell<T>` من عدد عمليات الاستعارة النشطة الثابتة، وعندما تخرج قيمة من النوع `Ref<T>` عن النطاق، ينخفض عدد الاستعارات الثابتة بمقدار واحد. يتيح لنا المؤشر `RefCell<T>` الحصول على العديد من الاستعارات الثابتة أو استعارة واحدة متغيّرة في أي وقت تمامًا مثل قواعد الاستعارة وقت التصريف.

إذا حاولنا انتهاك هذه القواعد، سيهلع تنفيذ `RefCell<T>` وقت التنفيذ بدلًا من الحصول على خطأ تصريفي كما اعتدنا حصوله مع المراجع. تظهر الشيفرة 23 تعديلًا في تطبيق الدالة `send` من الشيفرة 22، ونحاول هنا إنشاء استعارتين نشطتين متغيّرتين لنفس النطاق عمدًا لتوضيح أن `RefCell<T>` سيمنعنا من فعل ذلك وقت التنفيذ.

اسم الملف: src/lib.rs

```
impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        let mut one_borrow = self.sent_messages.borrow_mut();
        let mut two_borrow = self.sent_messages.borrow_mut();

        one_borrow.push(String::from(message));
        two_borrow.push(String::from(message));
    }
}
```

[الشيفرة 23: إنشاء مرجعين متغيّرين في النطاق ذاته لملاحظة هلع `RefCell<T>`]

نُنشئ متغير اسمه `one_borrow` للمؤشر الـذكي `RefMut<T>` الـذي أُعيـد من `borrow_mut`، ثم نُنشـئ استعارةً متغيّرة أخرى بنفس الطريقة في المتغير `two_borrow`، ممـا يـؤدي إلى إنشـاء مـرجعين متغيّـرين في النطاق ذاته وهو أمر غير مسموح به.

عندما ننفذ الاختبارات لمكتبتنا، تُصرَّف الشيفرة البرمجيـة الموجـودة في الشـيفرة 23 دون أي أخطـاء إلا أن الاختبار سيفشل:

```
$ cargo test
   Compiling limit-tracker v0.1.0 (file:///projects/limit-tracker)
    Finished test [unoptimized + debuginfo] target(s) in 0.91s
     Running unittests src/lib.rs (target/debug/deps/limit_tracker-
e599811fa246dbde)

running 1 test
test tests::it_sends_an_over_75_percent_warning_message ... FAILED

failures:

---- tests::it_sends_an_over_75_percent_warning_message stdout ----
thread 'main' panicked at 'already borrowed: BorrowMutError',
src/lib.rs:60:53
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace



failures:
    tests::it_sends_an_over_75_percent_warning_message

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0
filtered out; finished in 0.00s

error: test failed, to rerun pass `--lib`
```

لاحظ أن الشيفرة هلعت مع الرسالة `already borrowed: BorrowMutError`، وهذه هي الطريقـة التي يتعامل بها المؤشر `RefCell<T>` مع انتهاكات قواعد الاستعارة عند وقت التنفيذ.

اختيار اكتشاف أخطاء الاستعارة وقت التنفيذ بدلًا من وقت التصريف كما فعلنا هنا يعني أنك من المحتمل أن تجد أخطاءً في الشيفرة الخاصة بك لاحقًا في عملية التطوير، وربما حتى عند نشر الشـيفرة البرمجيـة الخاصـة بك ووصولها لمرحلة الإنتاج. قد تتكبّد شيفرتك البرمجية أيضًا خسارةً صغيرةً في الأداء عند وقت التنفيـذ وذلـك

نتيجة لتتبع الاستعارات عند وقت التشغيل بدلًا من وقت التصريف، ومع ذلك فإن استخدام `RefCell<T>` يجعل من الممكن كتابة كائن زائف يمكنه تعديل نفسه لتتبع الرسائل التي شاهدها أثناء استخدامه في سياق يسمح فقط بالقيم الثابتة. يمكنك استخدام `RefCell<T>` على الرغم من المقايضات للحصول على وظائف أكثر مما توفره المراجع العادية.

15.5.3 وجود عدة مالكين للبيانات المتغيرة باستخدام `Rc<T>` و `RefCell<T>`

هناك طريقة شائعة لاستخدام `RefCell<T>` بالاشتراك مع `Rc<T>`، تذكر أن `Rc<T>` يتيح لك وجود عدة مالكين لبعض البيانات إلا أنه يمنحك وصولًا ثابتًا إلى تلك البيانات. إذا كان لديك `Rc<T>` يحتوي على `RefCell<T>` فيمكنك الحصول على قيمة يمكن أن يكون لها عدة مالكين ويمكنك تغييرها.

على سبيل المثال تذكر مثال قائمة البنية في الشيفرة 18 من القسم السابق، إذ استخدمنا `Rc<T>` للسماح لقوائم متعددة بمشاركة ملكية قائمة أخرى، ونظرًا لأن `Rc<T>` يحتوي على قيم ثابتة فقط، فلا يمكننا تغيير أي من القيم الموجودة في القائمة بمجرد إنشائها. دعنا نضيف `RefCell<T>` لاكتساب القدرة على تغيير القيم في القوائم. تُظهر الشيفرة 24 أنه يمكننا تعديل القيم المخزّنة في جميع القوائم وذلك باستخدام `RefCell<T>` داخل تعريف `Cons`:

اسم الملف: src/main.rs

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));
```

```
        *value.borrow_mut() += 10;

        println!("a after = {:?}", a);
        println!("b after = {:?}", b);
        println!("c after = {:?}", c);
    }
```

[الشيفرة 24: استخدام `Rc<RefCell<i32>>` لإنشاء `List` يمكن تغييرها]

نُنشئ قيمة بحيث تكون نسخةً من النـوع `Rc<RefCell<i32>>` ونخزنهـا في متغـير باسـم `value` حـتى نتمكن من الوصول إليها مباشرةً لاحقًا، ثم نُنشئ `List` في `a` مع متغـاير `Cons` يحمـل `value`، نحتـاج هنـا إلى استنساخ `value` بحيث يكون لكل من `a` و `value` ملكية للقيمة الداخلية 5 بـدلًا من نقـل الملكيـة من `value` إلى `a` أو استعارة `a` من `value`.

نغلّف القائمة `a` داخل `Rc<T>` عند إنشاء القائمتين `a` و `b` بحيث يمكن لكلّ من القائمتين الرجوع إلى `a` وهو ما فعلناه في الشيفرة 18 سابقًا.

نريـد إضـافة القيمـة 10 إلى `value` بعـد إنشـاء القـوائم في `a` و `b` و `c`، ونحقّـق ذلـك عن طريـق اسـتدعاء `borrow_mut` على `value` التي تستخدم ميزة التحصيل التلقائي automatic differencing التي ناقشـناها سابقًا في الفصل 6 ضمن فقرة بعنوان (أين العامل `->`؟) لتحصيل `Rc<T>` إلى قيمـة `RefCell<T>` الداخليـة. يُعيد التابع `borrow_mut` المؤشر الذكي `RefMut<T>`، ومن ثم نستخدم عامـل التحصـيل عليـه ونغـير القيمـة الداخلية.

عندما ننفذ `a` و `b` و `c`، يمكننا أن نرى أن لجميعهم القيمة المعدلة 15 بدلًا من 5:

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.63s
     Running `target/debug/cons-list`
a after = Cons(RefCell { value: 15 }, Nil)
b after = Cons(RefCell { value: 3 }, Cons(RefCell { value: 15 }, Nil))
c after = Cons(RefCell { value: 4 }, Cons(RefCell { value: 15 }, Nil))
```

يا لهذه الطريقة الجميلة، فقد أصبح لدينا قيمة `List` ثابتة خارجيًا باستخدام `RefCell<T>`، إلا أنه يمكننا استخدام التوابع الموجودة في `RefCell<T>` التي تـوفر الوصـول إلى قابليـة التغيـير الداخليـة حـتى نتمكن من تعديل بياناتنا عندما نحتاج إلى ذلك. تحمينا عمليات التحقق وقت التنفيذ لقواعد الاستعارة من حـالات تسـابق البيانات data race وفي بعض الأحيان يكون الأمر مستحقًا لمقايضة القليل من السرعة مقابل هذه المرونة في

هياكل البيانات التي لدينا. لاحظ أن `RefCell<T>` لا يعمل مع الشيفرة متعددة الخيوط، ويعدّ `Mutex<T>` الإصدار الآمن من سلسلة `RefCell<T>`، وسنناقش `Mutex<T>` لاحقًا في الفصل 16.

15.6 حلقات المرجع Reference Cycles وتسببها بتسريب الذاكرة

قد تكون عملية ملء الذاكرة دون تحريرها (العملية المعروفة بتسريب الذاكرة memory leak يعني ملء الذاكرة) صعبة الحدوث بمستوى أمان الذاكرة الذي تقدمه لغة رست Rust، إلا أن حدوث هذا الأمر غير مستحيل، إذ لا تضمن رست منع تسريب الذاكرة بصورةٍ كاملة، والمقصود هنا أن الذاكرة المُسرّبة آمنة في رست. نلاحظ أن رست تسمح بتسريب الذاكرة باستخدام `Rc<T>` و `RefCell<T>`، إذ أنه من الممكن إنشاء مراجع تشير إلى بعضها ضمن حلقة cycle، وسيسبب هذا تسريب ذاكرة لأن عدد المراجع لكل عنصر لن يصل إلى 0 أبدًا، وبهذا لن تُحرَّر أي قيمة.

15.6.1 إنشاء حلقة مرجع

لنلاحظ كيف من الممكن أن نشكّل حلقة مرجع لتفادي هذا الأمر، بدءًا بتعريف معدّد `List` وتابع `tail` في الشيفرة 25:

اسم الملف: src/main.rs

```
use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}
```

```
fn main() {}
```

[الشيفرة 25: تعريف قائمة بنية تخزّن `RefCell<T>` داخلها، بحيث نستطيع تعديل القيمة التي يشير إليها متغاير `Cons`]

نستخدم هنا نوعًا مختلفًا من تعريف `List` من الشيفرة 5، إذ يصبح العنصر الثاني من المتغاير `Cons` أي variant مساويًا للنوع `RefCell<Rc<List>>`، مما يعني أننا نحتاج تعديل قيمة `List` المشار إليها في `Cons` عوضًا عن حاجتنا لقابلية التعديل على قيمة النوع `i32` كما فعلنا في الشيفرة 24. نضيف أيضًا التابع `tail` لتسهيل الوصول إلى العنصر الثاني إذا وُجد متغاير `Cons`.

أضفنا في الشيفرة 26 الدالة `main` التي تستخدم التعريف الموجود في الشيفرة 25، إذ تُنشئ الشيفرة قائمةً في `a` وقائمة في `b` تشير إلى القائمة `a`، وثم تعدِّل القائمة في `a` لتشير إلى `b` لتنشئ بذلك حلقة مرجع. كتبنا أيضًا تعليمات `println!` لتظهر قيمة عدد المراجع في نقاط مختلفة ضمن هذه العملية.

اسم الملف: src/main.rs

```
use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}

fn main() {
```

```
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a initial rc count = {}", Rc::strong_count(&a));
    println!("a next item = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("a rc count after b creation = {}",
Rc::strong_count(&a));
    println!("b initial rc count = {}", Rc::strong_count(&b));
    println!("b next item = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("b rc count after changing a = {}",
Rc::strong_count(&b));
    println!("a rc count after changing a = {}",
Rc::strong_count(&a));

    // ألغِ تعليق السطر التالي لتلاحظ وجود حلقة المرجع، إذ ستتسبب الشيفرة البرمجية بطفحان المكدس
    // println!("a next item = {:?}", a.tail());
}
```

[الشيفرة 26: إنشاء حلقة مرجع بحيث تشير قيمتي `List` إلى بعضهما بعضًا]

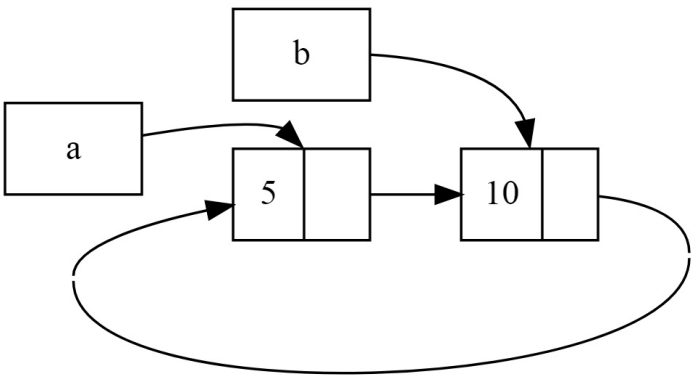
أنشأنا نسخةً من `Rc<List>` تحتوي على القيمة `List` في المتغير `a` مع قائمة مبدئية تحتوي على القيم `5, Nil`، ثم أنشأنا نسخة `Rc<List>` أخرى تحتوي قيمة `List` أخرى في المتغير `b` تحوي القيمة 10 وتشير إلى القائمة في `a`.

عدّلنا `a` لتشير إلى `b` بدلًا من `Nil` لنحصل بذلك على حلقة، وحقّقنا ذلك باستخدام التابع `tail` للحصول على مرجع إلى `RefCell<Rc<List>>` في `a`، والذي وضعناه بعد ذلك في المتغير `link`، ثم استخدمنا التابع `borrow_mut` على القيمة `RefCell<Rc<List>>` لتغيير القيمة داخل `Rc<List>` التي تخزّن القيمة `Nil` إلى القيمة `Rc<List>` الموجودة في `b`.

نحصل على الخرج التالي عندما ننفذ هـذه الشـيفرة البرمجيـة مـع الإبقـاء على تعليمـة `println!` الأخـيرة معلّقة:

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.53s
     Running `target/debug/cons-list`
a initial rc count = 1
a next item = Some(RefCell { value: Nil })
a rc count after b creation = 2
b initial rc count = 1
b next item = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })
b rc count after changing a = 2
a rc count after changing a = 2
```

يبلغ عدد مراجع نُسخ `Rc<List>` في كلٍّ من `a` و `b` القيمة 2 وذلك بعد تغيـير القائمـة في `a` لتشـير إلى `b`. تُسقِط أو تحذف لغة رست في نهاية الدالة `main` المتغير `b` مما يغيّر من عدد مراجـع نسـخة `b` من `Rc<List>` من 2 إلى 1، ولن تُحرّر الذاكرة التي تشغلها `Rc<List>` على الكومة heap في هذه اللحظة لأن عدد المراجع هـو 1 وليس 0، ومن ثم تُحـرّر رسـت `a`، ممـا يُنقص عـداد المرجـع لنسـخة `a` من `Rc<List>` من 2 إلى 1 أيضًـا. لا يُمكن تحرير ذاكرة النسخة هذه لأن نسخة `Rc<List>` الأخـرى لا تـزال تشـير إليهـا، وسـتبقى الـذاكرة المحجـوزة للقائمة شاغرة للأبد، ولتخيّل حلقة المرجع هذه بصريًا أنشأنا المخطط التالي في الشكل 4.



الشكل 14: حلقة مرجع خاصة بقائمتين a و b، تشيران إلى بعضهما بعضًا

إذا أزلت التعليق عن آخر تعليمة `println!` ونفّذت البرنامج، فستحاول رسـت طباعـة الحلقـة مـع إشـارة القائمة `a` إلى القائمة `b` وإشارتها بدورها إلى القائمة `a` وهكذا دواليك حتى يطفح المكدّس.

ليست عواقب إنشاء حلقة مرجعية هنا خطيرة مقارنةً بالبرامج الواقعيـة، إذ أن برنامجنـا ينتهي بعـد إنشـائنا حلقة مرجع reference cycle، إلا أن البرنامج سيستخدم ذاكرة أكثر ممـا يحتـاج إذا كـان البرنـامج أكـثر تعقيـدًا

وشَغَل ذاكرة أكثر في الحلقة واحتفظ بها لفترة أطول، وربما سيتسبّب ذلك بتحميل النظام عبئًا كبيرًا مسببًا نفاد الذاكرة المتوفرة.

إنشاء حلقات مرجع ليس بالأمر السهل ولكنه ليس مستحيلًا، فإذا كان لديك قيم `RefCell<T>` تحتوي على قيم `Rc<T>` أو تشكيلات متداخلة مشابهة لأنواع مع قابلية التغيير الداخلي وعدّ المراجع، فيجب عليك التأكد أنك لم تنشئ حلقات، إذ لا يجب عليك الاعتماد على رست لإيجادها. إنشاء حلقة مرجع يحصل نتيجة خطأ منطقي في برنامجك ولذلك يجب عليك استخدام الاختبارات التلقائية ومراجعة الشيفرة البرمجية ووسائل تطوير البرامج الأخرى لتقليل احتمالية حدوثها.

يمكن إعادة تنظيم هيكلية البيانات كحلٍ آخر لتفادي حلقات المرجع، بحيث تعبّر بعض المراجع عن الملكية بينما لا يعبّر بعضها الآخر، ونتيجةً لذلك سيكون لديك حلقات مكونة من بعض علاقات الملكية ownership وبعضها من علاقات لا ملكية non-ownership، بحيث تؤثر علاقات الملكية فقط إذا كانت القيمة ستُحرَّر.

نريد من المتغاير `Cons` في الشيفرة 25 أن يملك قائمةً خاصةً به دائمًا، لذا فإن عملية إعادة تنظيم هيكلية البيانات ليست ممكنة. لنتابع مثالًا آخرًا باستخدام البيانات graphs بحيث تتكون من عُقد أب وعُقد أبناء لنرى كيف أن العلاقات اللا ملكية هي طريقة مناسبة لمنع حصول حلقات المرجع.

15.6.2 منع حلقات المرجع: بتحويل Rc إلى Weak

وضّحنا بحلول هذه النقطة أن استدعاء `Rc::clone` يزيد من قيمة `strong_count` الخاصة بالنسخة `Rc<T>` وأن نسخة `Rc<T>` تُحرَّر فقط عندما تكون قيمة `strong_count` هي 0. بإمكاننا أيضًا إنشاء مرجع ضعيف weak reference للقيمة داخل نسخة `Rc<T>` وذلك باستدعاء `Rc::downgrade` وتمرير مرجع إلى `Rc<T>`. المراجع القوية هي الطريقة التي يمكنك بها مشاركة الملكية لنسخة `Rc<T>`، بينما لا تعبّر المراجع الضعيفة عن علاقة ملكية، ولا يتأثر عددها عندما تُحرَّر نسخة من `Rc<T>`، ولا يسبب حلقة مرجعية، إذ ستُكسر الحلقات المتضمنة لمراجع ضعيفة عندما تصبح قيمة عدد المراجع القوية مساويةً إلى 0.

نحصل على مؤشر ذكي من النوع `Weak<T>` عندما نستدعي `Rc::downgrade`، وبدلًا من زيادة `strong_count` في نسخة `Rc<T>` بقيمة 1، سيزيد استدعاء `Rc::downgrade` من قيمة `weak_count` بمقدار 1. يستخدم النوع `Rc<T>` القيمة `weak_count` لمتابعة عدد مراجع `Weak<T>` الموجودة بصورةٍ مشابهة للقيمة `strong_count`، إلا أن الفرق هنا هو أن `weak_count` لا يحتاج أن يكون 0 لكي تُنظف نسخة `Rc<T>`.

يجب علينا التأكد أن القيمة موجودة فعلًا إذا أردت إجراء أي عمليات على القيمة التي يشير إليها `Weak<T>`، وذلك لأن القيمة قد تُحرَّر، ويمكننا التحقق من ذلك باستدعاء تابع `upgrade` على نسخة `Weak<T>` التي تعيد قيمةً من النوع `Option<Rc<T>>`، وسنحصل على نتيجة `Some` إذا لم تُحرَّر القيمة `Rc<T>` ونتيجة

None إذا حُرّرت القيمة، تضمن رست أن حالتي Some و None ستُعامل على النحو الصحيح ولن تصبح مؤشرًا غير صالح، وذلك لأن upgrade تعيد Option<Rc<T>>.

لنأخذ مثالًا، فبدلًا من استخدام قائمة تعرف عناصرها العنصر الذي يليها فقط، سننشئ شجرةً تعرف عناصرها كلٍ من أبنائها وآبائها.

15.6.3 إنشاء هيكل بيانات الشجرة يحتوي على عقدة مع عقد أبناء

أولًا، ننشئ شجرة مع عقد nodes تعرف عقد أبنائها، ولتحقيق ذلك ننشئ بنيةً ندعوها Node تحتوي على قيم من النوع i32 إضافةً إلى مراجع تشير لقيم أبنائها Node:

اسم الملف: src/main.rs

```
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
struct Node {
    value: i32,
    children: RefCell<Vec<Rc<Node>>>,
}

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        children: RefCell::new(vec![]),
    });

    let branch = Rc::new(Node {
        value: 5,
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });
}
```

نريد من Node أن تمتلك أبنائها، كما نريد مشاركة الملكية مع المتغيرات لكي نستطيع الوصول لكل Node في الشجرة مباشرةً، ولتحقيق ذلك نعرّف عناصر Vec<T> بحيث تكون قيمًا من النوع Rc<Node>؛ ونريد أيضًا تعديل العقد الأبناء لعقدة أخرى، لذلك سيكون لدينا RefCell<T> في children وحول Vec<Rc<Node>>.

سنستخدم تعريف الهيكلية لإنشاء نسخة `Node` واحدة بالاسم `leaf` وقيمتها 3 دون أن تحتوي على أبناء، ونسخةً أخرى اسمها `branch` قيمتها 5 تحتوي على `leaf` بمثابة واحد من أبنائها كما هو موضح في الشيفرة 27:

اسم الملف: src/main.rs

```
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
struct Node {
    value: i32,
    children: RefCell<Vec<Rc<Node>>>,
}

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        children: RefCell::new(vec![]),
    });

    let branch = Rc::new(Node {
        value: 5,
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });
}
```

[الشيفرة 27: إنشاء عقدة `leaf` دون أبناء وعقدة `branch` مع `leaf` بمثابة واحد من أبنائها]

ننسخ القيمة `Rc<Node>` الموجودة في `leaf` ونخزّنها في `branch` وبذلك تصبح `Node` في `leaf` مُمتلكةً من قبل مالكين، هما `leaf` و `branch`. يمكننا الانتقال من `branch` إلى `leaf` عبر `branch.children` ولكن لا يوجد طريقة للوصول إلى `leaf` من `branch`، وسبب ذلك هو أن `leaf` لا تمتلك مرجع إلى `branch`، وبالتالي لا تعرف بأنها مرتبطة مع `branch`، إذًا نحن بحاجة لأن تعرف أن `branch` هي العقدة الأب وهذا ما سنفعله.

15.6.4 إضافة مرجع يشير إلى عقدة ابن داخل عقدة أب

نحتاج لإضافة حقل `parent` لجعل عقدة ابن تعرف بوجود آبائها وذلك ضمن تعريف الهيكل `Node`. تكمن صعوبة الأمر في اختيار النوع الذي يجب استخدامه لتخزين القيمة `parent`، إلا أننا نعلم أنه لا يمكن للقيمة أن تحتوي النوع `Rc<T>` لأننا نحصل بذلك على حلقة مرجع تحتوي على القيمة `leaf.parent` مشيرةً إلى `branch` و `branch.children` مشيرةً إلى `leaf` مما يجعل قيم `strong_count` غير مساوية إلى القيمة 0 في أي من الحالات.

لنفكّر بالعلاقات بطريقة أخرى؛ إذ يجب على العقدة الأب أن تمتلك أبنائها، إذا حُرِّرَت العقدة الأب فيجب على العُقد التابعة لها (الأبناء) أن تُسقط أيضًا، إلا أنه ليس من المفترض أن تمتلك عقدة ابن العقدة الأب، فإذا حرّرنا العقدة الابن يجب أن تبقى العقدة الأب موجودة، وهذه هي الحالة التي صُمّمت من أجلها المراجع الضعيفة.

لذا نجعل النوع `parent` يستخدم `Weak<T>` بدلًا من `Rc<T>` وتحديدًا النوع `RefCell<Weak<Node>>`، وسيصبح تعريف الهيكل `Node` على النحو التالي:

اسم الملف: src/main.rs

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
```

```
    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}
```

يمكن للعقدة أن تُشير إلى العقدة الأب ولكن لا يمكن أن تمتلكها. عـدّلنا من الدالـة `main` في الشـيفرة 28 بحيث تستخدم التعريف الجديد لكي تكون للعقدة `leaf` طريقة للإشارة إلى العقدة الأب `branch`:

اسم الملف: src/main.rs

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
```

```
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}
```

[الشيفرة 28: عقدة `leaf` مع مرجع ضعيف للعقدة الأب `branch`]

يبدو إنشاء العقدة `leaf` شبيهًا للشيفرة 27، باختلاف الحقل `parent`، إذ تبدأ العقدة `leaf` بدون أب وهذا يمكّننا من إنشاء نسخة لمرجع `Weak<Node>` فارغ.

عندما نحاول الحصول على مرجع للعقدة الأب الخاصة بالعقدة `leaf` وذلك باستخدام التابع `upgrade`، نحصل على قيمة `None`، ويمكن رؤية الخرج الناتج من أول تعليمة `println!`:

```
leaf parent = None
```

نحصل على مرجع `Weak<Node>` جديد عندما نُنشئ العقدة `branch` وذلك في الحقل `parent` لأن `branch` لا تحتوي على عقدة أب. لا يزال لدينا `leaf` وهي عقدة ابن للعقدة `branch`، وحالما يوجد لدينا نسخة من `Node` في `branch` سيكون بإمكاننا تعديل `leaf` بمنحها مرجعًا إلى العقدة الأب الخاصة بها من النوع `Weak<Node>`. نستخدم التابع `borrow_mut` على النوع `RefCell<Weak<Node>>` في حقل `parent` الخاص بالعقدة `leaf`، ومن ثم نستخدم الدالة `Rc::downgrade` لإنشاء مرجع من النوع `Weak<Node>` يشير إلى العقدة `branch` من النوع `Rc<Node>` في `branch`.

نحصل على متغاير `Some` يحتوي على `branch` عندما نطبع أب العقدة `leaf` مجددًا، إذ يمكن للعقدة `leaf` الآن الوصول إلى العقدة الأب. نستطيع أيضًا تفادي إنشاء الحلقة التي ستتسبب أخيرًا في طفحان المكدّس عند طباعة `leaf` كما هو الحال في الشيفرة 26؛ إذ تُطبع مراجع `Weak<Node>` مع الكلمة `(Weak)` جانبها:

```
leaf parent = Some(Node { value: 5, parent: RefCell { value: (Weak) },
children: RefCell { value: [Node { value: 3, parent: RefCell { value:
(Weak) },
children: RefCell { value: [] } }] } })
```

يشير عدم وجود خرج لانهائي إلى أن الشيفرة لم تُنشئ حلقة مرجع، ويمكنك التأكد من ذلك عن طريق ملاحظة القيم التي نحصل عليها باستدعاء كل من `Rc::strong_count` و `Rc::weak_count`.

15.6.5 مشاهدة التغييرات التي تحصل على strong_count و weak_count

دعنا نلاحظ كيف تتغير قيم نُسخ `Rc<Node>` لكل من النسخة `strong_count` و `weak_count`، وذلك عن طريق إنشاء نطاق داخلي جديد ونقل عملية إنشاء `branch` إلى هذا النطاق، إذ نستطيع بفعل ذلك مشاهدة ما الذي يحصل عند إنشاء العقدة `branch` وتحريرها بعد أن تخرج من النطاق. التعديلات موضحة في الشيفرة 29:

اسم الملف:src/main.rs

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );

    {
        let branch = Rc::new(Node {
            value: 5,
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(vec![Rc::clone(&leaf)]),
```

```
        });

        *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

        println!(
            "branch strong = {}, weak = {}",
            Rc::strong_count(&branch),
            Rc::weak_count(&branch),
        );

        println!(
            "leaf strong = {}, weak = {}",
            Rc::strong_count(&leaf),
            Rc::weak_count(&leaf),
        );
    }

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );
}
```

[الشيفرة 29: إنشاء `branch` في نطاق داخلي وفحص عدد المراجع القوية والضعيفة]

يبلغ عدد المراجع القوية للنوع `Rc<Node>` القيمة 1 بعد إنشاء `leaf`، وعدد المراجع الضعيفة يساوي 0. نُنشئ في النطاق الداخلي العقدة `branch` ونربطها مع `leaf` وهذه هي النقطة التي نبدأ فيها بطباعة عدد المراجع، يبلغ عدد المراجع القوية الخاصة بالنوع `Rc<Node>` في `branch` القيمة 1 وعدد المراجع الضعيفة 1 (لأن `leaf.parent` تشير إلى `branch` باستخدام قيمة من النوع `Weak<Node>`). نلاحظ تغيّر عداد المراجع القوية إلى 2 عندما نطبع عدد المراجع القوية والضعيفة في `leaf` وذلك لأن `branch` هي نسخةٌ من النوع `Rc<Node>` من القيمة `leaf` ومخزّنة في `branch.children`، إلا أننا ما زلنا نحصل على عدد مراجع ضعيفة يساوي 0.

تخـرج `branch` عن النطـاق عنـدما ينتهي النطـاق الـداخلي وينقص عـدد المراجـع القـوي الخـاص بـالنوع `Rc<Node>` إلى 0، لـذا تُحـرّر قيمـة `Node` الخاصـة بـه. لا يوجـد تـأثير لعـدد المراجـع الضـعيفة البـالغ 1 ضـمن `leaf.parent` على تحرير القيمة `Node` أو عدم تحريرها ولذلك لا نحصل على تسريب للذاكرة.

إذا أردنا الوصول إلى العقدة الأب الخاصة بالعقدة `leaf` في نهايـة النطـاق، فسنحصـل على `None` مجـددًا. عدد المراجع القوية الخاصة بالنوع `Rc<Node>` في `leaf` هو 1، وعدد المراجع الضعيفة هو 0 في نهاية البرنامج، وذلك لأن المتغير `leaf` هو المرجع الوحيد للنوع `Rc<Node>` مجددًا.

المنطق الذي يُدير عدّ وتحرير القيم مُطبَّق في كلٍّ من `Rc<T>` و `Weak<T>`، إضافةً إلى تنفيذ السمة `Drop`. سـيجعل تحديـد أن علاقـة العقـدة الابن بالعقـدة الأب يجب أن تكـون مرجعًـا ضـعيفًا من النـوع `Weak<T>` في تعريـف `Node`، وجـود عقـدة أب تشـير إلى عُقـد ابن وبـالعكس ممكنًـا دون إنشـاء حلقـة مرجـع والتسـبُّب بتسريب ذاكرة.

15.7 خاتمة

ناقشنا في هذا الفصل كيفية استخدام المؤشـرات الذكيـة لتقـديم ضـمانات وتنـازلات معيّنـة لأجـل أخـرى إضافية عن التي تقدّمها رست بالمراجع الافتراضية والعادية. للنوع `Box<T>` حجمٌ معـروف ويُشـير إلى البيانـات المخزّنة على الكومة، بينما يُحصي النوع `Rc<T>` عـدد المراجـع الـتي تشـير إلى البيانـات الموجـودة على الكومـة بحيث يكون لهذه البيانات عدّة مالكين، ويمنحنـا `RefCell<T>` مـع قابليـة التغيـير الداخليـة الخاصـة بـه نوعًـا يمكننا استخدامه عندما نحتاج إلى نوع ثابت، إلا أننا بحاجة أيضًا لتغيير قيمته الداخلية في ذات الوقت، كمـا أنّـه يطبّق قواعد الاستعارة وقت التنفيذ بدلًا من وقت التصريف.

كما ناقشنا أيضًا سـمتي `Deref` و `drop` اللتـان تعطيـان خصـائص كثـيرة للمؤشـرات الذكيـة، إضـافةً إلى حلقات المرجع التي تتسبب بتسريب للذاكرة وكيفية تفاديها باستخدام `Weak<T>`.

إذا أثار هذا الفصل اهتمامك وأردت تطبيق مؤشـرات ذكيـة خاصـة بـك، فراجـع "The Rustonomicon" لمعلومات مفيدة أكثر.

سنتحدث فيما يلي عن التزامن concurrency في رست وسنتعرف على مؤشرات ذكية أُخرى.

هل تريد كتابة سيرة ذاتية احترافية؟

نساعدك في إنشاء سيرة ذاتية احترافية عبر خبراء توظيف
مختصين في أكبر منصة توظيف عربية عن بعد

أنشئ سيرتك الذاتية الآن

16. البرمجة المتزامنة الآمنة

يُعدّ التعامل مع البرمجة المتزامنة concurrent programming بأمان وفعاليـة هـدفًا آخـرًا من الأهـداف الرئيسة للغة رست Rust، وتُعرف البرمجة المتزامنة بأنهـا البرمجـة الـتي يُنفَّذ فيهـا أجـزاء مختلفـة من البرنـامج بصورةٍ مسـتقلة، أمـا البرمجـة المتوازيـة parallel programming فهي عنـدما يُنفَّذ فيهـا أجـزاء مختلفـة من البرنامج في نفس الـوقت، وقـد أصـبح هـذا ذات أهميـة متزايـدة، إذ تسـتفيد الكثـير من أجهـزة الحاسـوب من معالجاتها المتعددة. كانت البرمجة تاريخيًـا في هـذا السـياق صـعبة وعرضـة للخطـأ وتأمـل لغـة رسـت أن تغيّر من ذلك.

اعتقد فريق رست في البداية أن ضمان سلامة الذاكرة ومنع مشاكل التزامن كانا تحـديين منفصـلين ويجب حلّهما بطرق مختلفـة، إلا أن الفريـق اكتشـف بمـرور الـوقت أن أنظمـة الملكيـة والنـوع هي مجموعـة قويـة من الأدوات للمساعدة في إدارة مشاكل أمـان الـذاكرة والـتزامن، وذلـك من خلال الاسـتفادة من التحقـق من النـوع والملكية، إذ أن العديد من أخطاء التزامن في رست هي أخطاء تحدث وقت التصريف compile-time errors وليست أخطاء وقت التنفيذ. لـذلك، بـدلًا من جعلـك تقضـي الكثـير من الـوقت في محاولـة إعـادة إنتـاج نفس الظروف التي يحدث فيها خطأ التزامن في وقت التنفيـذ سـترفض الشـيفرة الخطـأ أن تُصـرَّف وسـتعرض خطـأً يشرح المشكلة. يمكنك نتيجةً لذلك تصحيح شيفرتك أثناء عملك عليها بدلًا من محاولة تصحيحها بعـد وصـولها لمرحلة الإنتاج. لقّبنا هـذا الجـانب من رسـت بـالتزامن الجـريء fearless concurrency، ويـتيح لـك الـتزامن الجريء كتابة شيفرة خالية من الأخطاء الدقيقة بحيث يسهل إعادة بنائها دون حدوث أخطاء جديدة.

> سنشير إلى العديد من المشكلات -على سبيل البساطة- بكونها متزامنة بدلًا من أن تكون أكثر دقة بقولنا متزامنة و/أو متوازية parallel. إذا كان هذا الكتاب عن التزامن و/أو التوازي فسنكون أكثر تحديدًا. بالنسبة لهذا الفصل فيرجى استبدال المتزامن و/أو المتوازي في ذهننا كلما استخدمنا كلمة متزامن.

تعدّ العديد من اللغات متشددة بشأن الحلول التي تقدمها للتعامل مع المشاكل المتزامنة، فعلى سبيل المثال تتمتع لغة إيرلانج Erlang بمزايا أنيقة لتمرير الرسائل المتزامنة ولكن لديها طرق غامضة لمشاركة الحالة بين الخيوط. دعم مجموعة فرعية فقط من الحلول الممكنة هو إستراتيجية معقولة للغات ذات المستوى الأعلى لأن لغة المستوى الأعلى تَعِد بفوائد معيّنة عن طريق التخلي عن بعض السيطرة لاكتساب مفاهيم مجرّدة، وعلى الرغم من أن هذا من المتوقع؛ بحيث توفر اللغات ذات المستوى الأدنى الحل بأفضل أداء في أي موقف يُعطى ويكون لديها مستوى أقل من التجريد على العتاد الصلب، لذلك تقدم رست مجموعةً متنوعةً من الأدوات لنمذجة المشكلات بأي طريقة مناسبة لحالتك ومتطلباتك.

فيما يلي المواضيع التي سنغطيها في هذا الفصل:

- كيفية إنشاء خيوط threads لتنفيذ أجزاء متعددة من الشيفرة في الوقت ذاته.
- تزامن تمرير الرسائل massage-passing إذ ترسل القنوات الرسائل بين الخيوط.
- تزامن الحالة المشتركة shared-state إذ يُتاح للخيوط المتعددة الوصول لبعض البيانات.
- السمتان Sync و Send اللتان توسعان ضمانات تزامن رست لأنواع معرّفة من قبل المستخدم وأيضًا الأنواع المضمّنة في المكتبة القياسية.

16.1 استخدام الخيوط Threads لتنفيذ الشيفرة بصورة متزامنة آنيًا

تُنفّذ شيفرة البرنامج في معظم أنظمة التشغيل الحالية ضمن عملية process ويُدير نظام التشغيل عمليات متعددة في وقت واحد، يمكنك أيضًا العثور على أجزاء مستقلة تعمل بصورةٍ متزامنة داخل البرنامج، وتسمى الميّزات التي تنفّذ هذه الأجزاء المستقلة بالخيوط threads، على سبيل المثال يمكن أن يحتوي خادم الويب web server على خيوط متعددة بحيث يمكنه أن يستجيب لأكثر من طلب واحد في نفس الوقت.

يمكن أن يؤدي تجزئة عمليات الحساب في برنامجك إلى خيوط متعددة لتنفيذ مهام متعددة في نفس الوقت إلى تحسين الأداء ولكنه يضيف تعقيدًا إضافيًا أيضًا. لا يوجد ضمان حول الترتيب الذي ستُنفَّذ فيه أجزاء من شيفرتك البرمجية في الخيوط المختلفة، وذلك بسبب إمكانية تنفيذ الخيوط بصورةٍ متزامنة، ويمكن لهذا أن يؤدي إلى مشاكل مثل:

- حالات التسابق race conditions، إذ يمكن للخيوط الوصول للبيانات أو المصادر بترتيب غير مضبوط.
- أقفال ميتة deadlocks، وهي الحالة التي ينتظر فيها الخيطان بعضهما بعضًا مما يمنع كلا الخيطين من الاستمرار.
- الأخطاء التي تحدث فقط في حالات معينة وتكون صعبة الحدوث دومًا والإصلاح بصورةٍ موثوقة.

تحاول رست التخفيف من الآثار السلبية لاستخدام الخيوط ولكن لا تزال البرمجة في سياق متعدد الخيوط تتطلّب تفكيرًا حذرًا ويتطلب هيكل شيفرة برمجية مختلف عن تلك الموجودة في البرامج المنفذة في خيط مفرد.

تُنفّذ لغات البرمجة الخيوط بطرق مختلفة عن بعضها البعض، وتوفر العديد من أنظمة التشغيل واجهة برمجية يمكن للغة أن تستدعيها لإنشاء خيوط جديدة. تستخدم مكتبة رست القياسية نموذج 1:1 لتنفيذ الخيط، إذ يستخدم البرنامج خيط نظام تشغيل واحد لكل خيط لغة. هناك وحدات مصرفة تنفذ نماذج أخرى للخيوط لتقديم مقايضات مختلفة عن نموذج 1:1.

16.1.1 إنشاء خيط جديد باستخدام spawn

نستدعي الدالة `thread::spawn` لإنشاء خيط جديد ونمرر لها مغلَّف closure يحتوي على الشيفرة التي نريد أن ننفذها في الخيط الجديد (تحدثنا عن المغلفات سابقًا في الفصل 13. يطبع المثال في الشيفرة 1 نصًا ما من خيط رئيسي ونص آخر من خيط جديد:

اسم الملف: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

[الشيفرة 1: إنشاء خيط جديد لطباعة شيء ما بينما يطبع الخيط الرئيسي شيئًا آخر]

لاحظ أنه عندما يكتمل الخيط الرئيسي لبرنامج رست، تُغلَق كل الخيوط المُنشأة بغض النظر إذا أنهت التنفيذ أم لا، ويمكن لخرج هذا البرنامج أن يكون مختلفًا في كل مرة ولكنه سيكون مشابهًا لما يلي:

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

تؤدي استدعاءات `thread::sleep` إلى إجبار الخيط على إيقاف تنفيذه لمدة قصيرة مما يسمح بتشغيل خيوط مختلفة، ومن المحتمل أن تتناوب الخيوط ولكن هذا الأمر غير مضمون الحدوث، إذ يعتمد ذلك على كيفية جدولة نظام التشغيل الخاص للخيوط. يُطبع في التنفيذ السابق الخيط الرئيسي أولًا على الرغم من ظهور عبارة الطباعة من الخيط الذي أُنشئ أولًا في الشيفرة البرمجية، وعلى الرغم من أننا أخبرنا الخيط المُنشأ أن يطبع حتى تصبح `i` مساوية للقيمة 9 إلا أنه وصل إلى 5 فقط قبل إغلاق الخيط الرئيسي.

إذا نفَّذت هذه الشيفرة ورأيت فقط المخرجات من الخيط الرئيسي أو لم ترَ أي تداخل، فحاول زيادة الأرقام في المجالات لإنشاء المزيد من الفرص لنظام التشغيل للتبديل بين الخيوط.

16.1.2 انتظار انتهاء كل الخيوط بضم المقابض Handles عن طريق join

لا توقف الشيفرة البرمجية الموجودة في الشيفرة 1 الخيط المُنتَج قبل الأوان في معظم الأوقات بسبب انتهاء الخيط الرئيسي، وإنما بسبب عدم وجود ضمان على الترتيب الذي تُنفَّذ به الخيوط، إذ لا يمكننا أيضًا ضمان أن الخيط المُنتَج سيُنفذ إطلاقًا.

يمكننا إصلاح مشكلة عدم تشغيل الخيط الناتج spawned أو انتهائه قبل الأوان عن طريق حفظ قيمة إرجاع `thread::spawn` في متغير، إذ يكون النوع المُعاد من `thread::spawn` هو `JoinHandle`؛ الذي يمثّل قيمةً مملوكةً، بحيث عندما نستدعي التابع `join` عليها ستنتظر حتى ينتهي الخيط الخاص بها. تُظهر الشيفرة 2 كيفية استعمال `JoinHandle` الخاص بالخيط التي أنشأناه في الشيفرة 1 واستدعاء `join` للتأكد من انتهاء الخيط المنتج قبل الخروج من الدالة `main`:

اسم الملف: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
```

```
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

[الشيفرة 2: حفظ `JoinHandle` من `thread::spawn` لضمان تنفيذ الخيط لحين الاكتمال]

يؤدي استدعاء `join` على المقبض handle إلى إيقاف تنفيذ الخيط الجاري حتى ينتهي الخيط الذي يمثلـه المقبض، ويعني إيقاف تنفيذ blocking الخيط أن الخيط ممنوع من أداء العمـل أو الخـروج منـه. يجب أن ينتج تنفيذ الشيفرة 2 خرجًا مشابهًا لما يلي لأننا وضعنا استدعاء `join` بعد حلقة الخيط الرئيسي `for`:

```
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 1 from the spawned thread!
hi number 3 from the main thread!
hi number 2 from the spawned thread!
hi number 4 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

يستمرّ الخيطان بالتناوب، وينتظـر الخيـط الرئيسـي بسـبب اسـتدعاء `handle.join()`، ولا ينتهي حـتى ينتهي الخيط الناتج.

دعنا نرى ما سيحدث عندما ننقل `handle.join()` إلى ما قبل حلقة `for` في `main` على النحو التالي:

اسم الملف: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

سينتظر الخيط الرئيسي انتهاء الخيط الناتج، ثم سينفّذ الحلقة `for` الخاصة به لذلك لن تتداخل المخرجـات بعد الآن كما هو موضح هنا:

```
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 3 from the main thread!
```

```
hi number 4 from the main thread!
```

يمكن أن تؤثر التفاصيل الصغيرة على ما إذا كانت الخيوط الخاصة بك تُنفَّذ في نفس الوقت أم لا مثل استدعاء `join`.

16.1.3 استعمال مغلفات move مع الخيوط

نستخدم غالبًا الكلمة المفتاحية `move` مع المغلفات التي تُمرَّر إلى `thread::spawn` وذلك لأن المغلف سيأخذ بعد ذلك ملكية القيم التي يستخدمها من البيئة وبالتالي تُنقل ملكية هذه القيم من خيط إلى آخر، ناقشنا سابقًا في قسم "الحصول على المعلومات من البيئة باستخدام المغلفات" من الفصل 13 الكلمة المفتاحية `move` في سياق المغلفات، إلا أننا سنركز الآن أكثر على التفاعل بين `move` و `thread::spawn`.

لاحظ في الشيفرة 1 أن المغلف الذي نمرّره إلى `thread::spawn` لا يأخذ أي وسطاء arguments، إذ أننا لا نستخدم أي بيانات من الخيط الرئيسي في شيفرة الخيط المنتج، ولاستخدام البيانات من الخيط الرئيسي في الخيط المُنتج يجب أن يحصل مغلّف الخيط الناتج على القيم التي يحتاجها. تُظهر الشيفرة 3 محاولة إنشاء شعاع vector في الخيط الرئيسي واستعماله في الخيط الناتج، ومع ذلك لن ينجح هذا الأمر كما سترى بعد لحظة.

اسم الملف: src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

[الشيفرة 3: محاولة استعمال شعاع منشأ بواسطة الخيط الرئيسي ضمن خيط آخر]

يستخدم المغلّف الشعاع `v` لذلك سوف يحصل على القيمة `v` ويجعلها جزءًا من بيئة المغلف، ونظرًا لأن `thread::spawn` ينفّذ المغلف في خيط جديد فيجب أن نكون قادرين على الوصول إلى `v` داخل هذا الخيط الجديد، ولكن عندما نصرِّف الشيفرة السابقة نحصل على الخطأ التالي:

```
$ cargo run
   Compiling threads v0.1.0 (file:///projects/threads)
error[E0373]: closure may outlive the current function, but it borrows
`v`, which is owned by the current function
 --> src/main.rs:6:32
  |
|     let handle = thread::spawn(|| {
  |                                ^^ may outlive borrowed value `v`
|         println!("Here's a vector: {:?}", v);
  |                                           - `v` is borrowed here
  |
note: function requires argument type to outlive `'static`
 --> src/main.rs:6:18
  |
|       let handle = thread::spawn(|| {
  |  __________________^
| |         println!("Here's a vector: {:?}", v);
| |     });
  | |______^
help: to force the closure to take ownership of `v` (and any other
referenced variables), use the `move` keyword
  |
|     let handle = thread::spawn(move || {
  |                                ++++


For more information about this error, try `rustc --explain E0373`.
error: could not compile `threads` due to previous error
```

تستنتج رست كيفية الحصول على القيمة `v` ولأن `println!` تحتاج فقط إلى مرجع إلى `v`، يحاول المغلّف استعارة `v`، ومع ذلك هناك مشكلة، إذ لا يمكن لرست معرفة المدة التي سيُنفَّذ فيها الخيط الناتج لذلك لا تعرف ما إذا كان المرجع إلى `v` صالحًا دائمًا.

تقدّم الشيفرة 4 سيناريو من المرجح به أن يحتوي على مرجع غير صالح إلى `v`:

اسم الملف: src/main.rs

```
use std::thread;
```

```
fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    drop(v); // oh no!

    handle.join().unwrap();
}
```

[الشيفرة 4: خيط مع مغلف يحاول أن يحصل على مرجع يشير للشعاع v من خيط رئيسي يحرّر v]

إذا سمحت لنا رست بتنفيذ الشيفرة البرمجية السابقة فهناك احتمال أن يوضَع الخيط الناتج في الخلفية فورًا دون تنفيذ إطلاقًا. يحتوي الخيط الناتج على مرجع يشير إلى v من الداخل إلا أن الخيط الرئيسي يحرّر v فورًا باستخدام دالة drop التي ناقشناها في الفصل 15 السابق، وعندما يبدأ تنفيذ الخيط المنتج، لن تصبح v صالحة، لذلك فإن الإشارة إليها تكون أيضا غير صالحة، ولا نريد حدوث ذلك.

لتصحيح الخطأ التصريفي في الشيفرة 3 نتّبع النصيحة المزودة لنا في رسالة الخطأ:

```
help: to force the closure to take ownership of `v` (and any other
referenced variables), use the `move` keyword
  |
|     let handle = thread::spawn(move || {
  |                                ++++
```

يمكننا من خلال من خلال إضافة الكلمة المفتاحية move قبل المغلف أن نفرض عليه الحصول على ملكية القيم التي يستعملها بدلًا من السماح لرست بالتدخل والاستنتاج أن عليها استعارة القيم. التعديل على الشيفرة 3 موضّح في الشيفرة 5 وسيُصرَّف البرنامج ويُنفَّذ وفق المطلوب:

اسم الملف: src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];
```

```
    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

[الشيفرة 5: استعمال الكلمة المفتاحية `move` لنفرض على المغلف أن يأخذ ملكية القيم التي يستعملها]

قد نرغب في تجربة الأمر ذاته لتصحيح الشيفرة البرمجية في الشيفرة 4، إذ استدعى الخيط الرئيسي `drop` باستعمال مغلف `move`، ومع ذلك فإن هذا لن يعمل لأن ما تحاول الشيفرة 4 فعله غير مسموح به لسبب مختلف؛ وإذا أضفنا `move` إلى المغلف فسننقل `v` إلى بيئة المغلف ولن يعد بإمكاننا بعد ذلك استدعاء `drop` عليه في الخيط الرئيسي، وسنحصل على الخطأ التصريفي التالي بدلًا من ذلك:

```
$ cargo run
   Compiling threads v0.1.0 (file:///projects/threads)
error[E0382]: use of moved value: `v`
  --> src/main.rs:10:10
   |
|     let v = vec![1, 2, 3];
   |         - move occurs because `v` has type `Vec<i32>`, which does
not implement the `Copy` trait
|
|     let handle = thread::spawn(move || {
   |                                ------- value moved into closure
here
|         println!("Here's a vector: {:?}", v);
   |                                           - variable moved due to
use in closure
...
|     drop(v); // oh no!
   |          ^ value used here after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `threads` due to previous error
```

أنقذتنا قواعد ملكية رست مرةً أخرى، إذ حصلنا على خطأ في الشيفرة 3 لأن رست كانت صارمة باستعارة `v` للخيط ذاته فقط مما يعني أن الخيط الرئيسي يمكنه نظريًا إبطال مرجع الخيط الناتج، ويمكننا بإخبار رست أن

تنقل ملكية v إلى الخيط الناتج أن نضمن لرست أن الخيط الرئيسي لن يستخدم v بعد الآن. إذا عدّلنا الشيفرة 4 بالطريقة ذاتها فإننا بذلك ننتهك قواعد الملكية عندما نحاول استعمال v في الخيط الرئيسي. تتجاوز الكلمة المفتاحية move الوضع الافتراضي الصارم للاستعارة في رست، فلا يُسمح لنا بانتهاك قواعد الملكية.

بعد فهمنا أساسيات الخيوط وواجهتها البرمجية، لننظر عما يمكننا فعله باستخدام الخيوط.

16.2 استخدام ميزة تمرير الرسائل Message Passing لنقل البيانات بين الخيوط

يُعد تمرير الرسائل message passing أحد الطرق الشائعة لضمان أمن التزامن، إذ تتواصل الخيوط threads أو المنفذون actors فيما بينهم بإرسال رسائل تحتوي على بيانات. يمكن توضيح هذه الفكرة باقتباس من شعار في توثيق لغة البرمجة جو Go: "لا تتواصل بمشاركة الذاكرة، شارك الذاكرة بالتواصل".

تؤمّن مكتبة رست القياسية تنفيذًا للقنوات channels لتحقيق عملية تزامن إرسال الرسائل message-sending concurrency، والقناة هي مفهوم برمجي عام تُرسل به البيانات من خيط إلى آخر. يمكنك تخيل القناة في البرمجة مثل قناة مياه باتجاه واحد مثل جدول أو نهر، فإذا وضعت بطة مطاطية في النهر ستنتقل في المجرى إلى نهاية القناة المائية.

تنقسم القناة إلى نصفين أحدها مُرسل transmitter والآخر مُستقبل receiver؛ والمُرسل هو القسم الموجود أعلى النهر الذي تضع فيه البطة المطاطية؛ والمُستقبل هو ما تصل إليه البطة المطاطية في نهاية النهر. يستدعي قسمٌ من الشيفرة البرمجية التوابع على المُرسل مع البيانات المراد إرسالها، والقسم الآخر يتحقق من وصول الرسالة في القسم المُستقبل. يمكن القول إن القناة قد أُغلقت إذا سقط أي من القسمين المُرسل أو المستقبل.

سنعمل هنا على برنامج يحتوي على خيط واحد لتوليد القيم وإرسالها عبر القناة وخيط آخر سيستقبل القيم ويطبعها، وسنرسل قيمًا بسيطةً بين الخيوط باستخدام القناة وذلك بهدف توضيح هذه الميزة فقط، ويمكنك استخدام القناة -بعد أن نتعرف عليها جيدًا- على أي خيوط تحتاج لأن تتواصل مع بعضها بعضًا، مثل نظام محادثة أو نظام يكون فيه عدة خيوط تجري حسابات وإرسال هذه الأجزاء إلى خيط واحد يجّمع القيم.

نُنشئ في الشيفرة 6 قناةً دون استخدامها. لاحظ أن الشيفرة البرمجية لن تُصرف بعد، لأن رست لا تستطيع معرفة ما هو نوع القيم التي نريد إرسالها عبر هذه القناة.

اسم الملف: src/main.rs

```
use std::sync::mpsc;

fn main() {
```

```
    let (tx, rx) = mpsc::channel();
}
```

[الشيفرة 6: انشاء قناة وإسناد القسمين `tx` و `rx` إليها]

أنشأنا قناةً جديدةً باستخدام دالـة `mpsc::channel`، ويعـني الاختصـار "mpsc" عـدة منتجين multiple producer ومستهلك واحد single consumer. باختصار، طريقة تطبيق القنوات في مكتبة رسـت القياسـية هي أن كل قناة تحتوي على عدة مُرسلين ينتجون القيم ولكن هناك مُستقبل واحد لاسـتهلاك تلـك القيم. تخيّـل عدة مجاري streams تلتقي في نهرٍ واحد كبير: أي سـينتهي كـل شـيء يُرسـل من المجـاري في نهـر واحـد في النهاية. نبدأ بمنتج واحد وبعدها نضيف عدة منتجين بعد عمل هذا المثال.

تُعيد الدالة `mpsc::channel` صفًا tuple، العنصر الأول هو طرف الإرسال (المُرسل) والعنصر الثـاني هـو طرف الاستقبال (المُستقبل)، وتُسـتخدم عـادةً الاختصـارات `tx` و `rx` في العديـد من الحقـول fields للمُرسـل والمُستقبل على التوالي، لذا نُسمي متغيراتنا وفقًا لهذا الاصطلاح للدلالة على كل طرف.

نستخدم التعليمة `let` بنمط pattern يدمّر هيكلية الصفوف، وسنتحدث عن استخدام الأنماط في تعليمـة `let` وتدمير الهيكلية لاحقًا، ويكفي الآن معرفة أن استخدام تعليمة `let` هي الطريقة الأفضل لاسـتخراج أقسـام من الصف المُعاد باستخدام `mpsc::channel`.

لننقل الطرف المُرسل إلى خيط مُنشأ ولنجعله يرسل سلسلةً نصيةً لكي يتواصل الخيط المُنشـأ مـع الخيـط الرئيسي كما هو موضح في الشيفرة 7. يُماثل هذا الأمر وضع بطة مطاطية أعلى النهر أو إرسال رسـالة من خيـط إلى آخر.

اسم الملف: src/main.rs

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });
}
```

[الشيفرة 7: نقل `tx` إلى خيط مُنشأ وإرسال "hi"]

استخدمنا `thread::spawn` مجددًا لإنشاء خيط جديد، ثم استخدمنا `move` لنقل `tx` إلى مُغلّف closure بحيث يمتلك الخيط المُنشأ القيمة `tx`. يجب على الخيط المُنشأ أن يمتلك الطرف المُرسل لكي يستطيع إرسال رسائل عبر القناة، ولدى المرسل تابع `send` الذي يأخذ القيمة المُراد إرسالها، إذ يعيد التابع `send` قيمةً من النوع `Result<T, E>`، لذا إذا كان المُستقبل قد أُسقط وليس هناك مكان لإرسال القيمة، تعيد عملية الإرسال خطأً. استدعينا في هذا المثال `unwrap` ليهلع في حال الخطأ، ولكن يجب التعامل مع حالة الهلع في التطبيقات الحقيقية بطريقة مناسبة، راجع الفصل 9 لمراجعة استراتيجيات التعامل المناسب مع الأخطاء.

سنحصل في الشيفرة 8 على القيمة من المُستقبل في الخيط الأساسي، وتشابه هذه العملية استرجاع البطة المطاطية من نهاية النهر أو استقبال الرسالة.

اسم الملف: src/main.rs

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

[الشيفرة 8: استقبال القيمة "hi" في الخيط الأساسي وطباعتها]

للمستقبل تابعان مفيدان، هما `recv` و `try_recv`، استخدمنا `recv` -وهو اختصارٌ لكلمة استقبال receive- الذي يمنع تنفيذ الخيط الرئيسي وينتظر حتى تصل القيمة إلى نهاية القناة، ويعيد التابع `recv` بعد وصول القيمة إلى نهاية القناة القيمة ذاتها داخل النوع `Result<T, E>`، وعندما يُغلق المُرسل يعيد التابع `recv` خطأ للإشارة إلى أنه لا يوجد المزيد من القيم قادمة.

لا يحجب التابع `try_recv` الخيط الرئيسي وإنما يعيد قيمةً من النوع `Result<T, E>` مباشرةً، وتحتوي قيمة `Ok` رسالةً إذا كان هناك رسالة متوفرة وإلا فقيمة `Err` إذا لا يوجد أي رسائل هذه المرة. استخدام

`try_recv` مفيدٌ إذا كان للخيط أعمالٌ أخرى لينفذها بينما ينتظر الرسائل، وبإمكاننا كتابة حلقة تستدعي `try_recv` بصورةٍ متكررة لتتعامل مع الرسائل في حال قدومها، أو تنفّذ أعمالًا أخرى قبل تحققها مجدداً.

استخدمنا في مثالنا التابع `recv` لبساطته، وليس لدينا أي عمل آخر للخيط الرئيسي غير انتظار الرسائل، لذا فإن حجب الخيط الرئيس مناسب.

عندما ننفذ الشيفرة البرمجية في الشيفرة 8 نلاحظ القيمة المطبوعة في الخيط الرئيسي:

```
Got: hi
```

هذا ممتاز!

16.2.1 القنوات ونقل الملكية

تلعب الملكية دورًا مهمًا في إرسال الرسائل لأنها تساعد في كتابة شيفرة آمنة ومتزامنة، إذ يتطلب منع الأخطاء في البرمجة المتزامنة الانتباه على الملكية ضمن برنامجك باستخدام لغة رست. لنكتب مثالًا يظهر كيف تعمل كل من القنوات والملكية لمنع المشاكل، وسنستخدم قيمة `val` في الخيط المُنشأ بعد أن أرسلناه عبر القناة. جرّب تصريف الشيفرة البرمجية في الشيفرة 9 لترى سبب كون هذه الشيفرة غير مسموحة.

اسم الملف: src/main.rs

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        println!("val is {}", val);
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

[الشيفرة 9: محاولة استخدام `val` بعد إرسالها عبر القناة]

حاولنا هنا طباعة `val` بعد أن أرسلناها عبر القناة باستخدام `tx.send`، وسيكون السماح بذلك فكرةً سيئة، إذ يستطيع الخيط التعديل على القيمة أو إسقاطها بعد إرسالها عبره، ومن الممكن أن يسبّب تعديل الخيط الآخر أخطاءً أو قيمًا غير متوقعة أو بيانات غير موجودة، إلا أن رست تعطينا رسالة خطأ إذا جربنا تصريف الشيفرة البرمجية في الشيفرة 9.

```
$ cargo run
   Compiling message-passing v0.1.0 (file:///projects/message-passing)
error[E0382]: borrow of moved value: `val`
  --> src/main.rs:10:31
   |
|         let val = String::from("hi");
   |             --- move occurs because `val` has type `String`,
which does not implement the `Copy` trait
|         tx.send(val).unwrap();
   |                 --- value moved here
|         println!("val is {}", val);
   |                               ^^^ value borrowed here after move
   |
   = note: this error originates in the macro `$crate::format_args_nl`
which comes from the expansion of the macro `println` (in Nightly
builds, run with -Z macro-backtrace for more info)


For more information about this error, try `rustc --explain E0382`.
error: could not compile `message-passing` due to previous error
```

حدث خطأ وقت التصريف نتيجة حدوث خطأ في التزامن. تأخذ دالة `send` ملكيتها من معاملاتها، وعندما تُنقل القيمة، يأخذ المستقبل ملكيتها، وهذا يمنعنا من استخدامها مرةً أخرى بعد إرسالها؛ وبالنتيجة يتحقق نظام الملكية من أن كل شيء على ما يرام.

16.2.2 إرسال قيم متعددة مع انتظار المستقبِل

استطعنا تصريف وتنفيذ الشيفرة 8، ولكن لم يظهر لنا أن خيطين مستقلين كانا يتكلمان مع بعضهما عبر القناة. أجرينا بعض التعديلات في الشيفرة 10 لنبين أن الشيفرة البرمجية في الشيفرة 8 تُنفَّذ بصورةٍ متزامنة. سيُرسل الخيط المُنشأ الآن عدة رسائل وسيتوقف لمدة ثانية بين كل رسالة.

اسم الملف: src/main.rs

```
use std::sync::mpsc;
```

```
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

[الشيفرة 10: إرسال رسائل متعددة مع التوقف بين كل عملية إرسال]

للخيط المُنشأ هذه المرة شعاع vector من السلاسل النصية التي نريد إرسالها إلى الخيط الأساسي. نمرّ على السلاسل النصية ونرسلها بصورةٍ إفرادية ونتوقف بين كل واحدة وأخرى باستدعاء دالة `thread::sleep` مع قيمة `Duration` مساوية إلى 1 ثانية.

لم نستدعِ في الخيط الأساسي الدالة `recv` صراحةً بل عاملنا `rx` مثل مكرّر iterator، إذ نطبع كل قيمة نستقبلها، وتنتهي عملية التكرار عندما تُغلق القناة.

نلاحظ الخرج التالية عند تنفيذ الشيفرة البرمجية في الشيفرة 10 مع توقف لمدة ثانية بين كل سطر وآخر:

```
Got: hi
Got: from
```

```
Got: the
Got: thread
```

يمكننا معرفة أن الخيط الأساسي ينتظر استلام القيم من الخيط المُنشأ لأنه ليس لدينا شيفرة تتوقف أو تتأخر في الحلقة `for` ضمن الخيط الأساسي.

16.2.3 إنشاء عدة منتجين بواسطة نسخ المرسل

قلنا سابقًا أن `mpsc` هو اختصار لعدة منتجين ومستهلك واحد، دعنا نستخدم `mpsc` للبناء على الشيفرة 10 وذلك لإنشاء خيوط متعددة ترسل كلها قيمًا المُستقبل ذاته، ونستطيع عمل ذلك بنسخ المُرسِل كما هو موضح في الشيفرة 11:

اسم الملف: src/main.rs

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    // --snip--

    let (tx, rx) = mpsc::channel();

    let tx1 = tx.clone();
    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx1.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });
```

```
    thread::spawn(move || {
        let vals = vec![
            String::from("more"),
            String::from("messages"),
            String::from("for"),
            String::from("you"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }

    // --snip--
}
```

[الشيفرة 11: إرسال عدّة رسائل من عدّة منتجين]

نستدعي هذه المرة `clone` على المُرسِل قبل إنشاء أول خيط، وسيمنحنا ذلك مُرسلًا جديدًا يمكننا تمريره إلى الخيط المُنشأ الأول، ثم نمرر المُرسل الأصلي إلى الخيط المنشأ الثاني، وهذا يعطينا خيطين يرسل كل منهما رسالة مختلفة إلى مستقبل واحد.

يجب أن يكون الخرج كما يلي عندما ننفذ الشيفرة السابقة:

```
Got: hi
Got: more
Got: from
Got: messages
Got: for
Got: the
Got: thread
Got: you
```

قد تجد القيم بترتيب مختلف حسب نظامك، وهذا ما يجعل التزامن مثيرًا للاهتمام وصعبًا في الوقت ذاته، وإذا حاولت تجربة التزامن باستخدام `thread::sleep` بإعطائه قيمًا مختلفة على خيوط مختلفة، فكل تنفيذ سيكون غير محدد أكثر، مما يعطيك خرج مختلف في كل مرة.

الآن وبعد تعلمنا كيفية عمل القنوات دعنا نتعلم نوعًا آخرًا من التزامن.

16.3 تزامن الحالة المشتركة Shared-State Concurrency

يُعدّ تمرير الرسائل طريقةً جيدةً للتعامل مع التزامن ولكنها ليست الطريقة الوحيدة، إذ أن هناك طريقةٌ أخرى لوصول خيوط threads متعددة إلى ذات بيانات المُشاركة. ضع بالحسبان هذا الجزء من الشعار من توثيق لغة جو Go "لا تتواصل بمشاركة الذاكرة".

كيف سيبدو التواصل من خلال مشاركة الذاكرة؟ وبالإضافة إلى ذلك، لماذا يحذر المدافعون عن تمرير الرسائل message-passing من استخدام مشاركة الذاكرة memory sharing؟

تشبه القنوات channels في أي لغة برمجة -بطريقة ما- الملكية الفردية لأنه بمجرد نقلك لقيمة ما إلى قناة يجب ألا تستخدم هذه القيمة بعدها. يشبه تزامن الذاكرة المشتركة الملكية المتعددة multiple ownership، إذ يمكن للخيوط المتعددة أن تصل إلى موقع الذاكرة ذاته في الوقت نفسه. كما رأينا سابقًا (في الفصل السابق)، فقد جعلت المؤشرات الذكية smart pointers الملكية المتعددة ممكنة، ويمكن للملكية المتعددة أن تعقد الأمر لأن الملّاك المختلفين بحاجة إلى إدارة. يساعد نظام رست وقواعد الملكية الخاصة به كثيرًا في جعل عملية الإدارة صحيحة. لنلقي على سبيل المثال نظرةً على كائنات المزامنة mutexes التي تعدّ واحدةً من أكثر بدائل التزامن شيوعًا للذاكرة المشتركة.

16.3.1 استعمال كائنات المزامنة للسماح بالوصول للبيانات عبر خيط واحد بالوقت ذاته

كائن المزامنة Mutex هو اختصارٌ للاستبعاد المتبادل mutual exclusion، أي يسمح كائن المزامنة لخيط واحد فقط أن يصل إلى بعض البيانات في أي وقت، وللوصول إلى البيانات في كائن المزامنة يجب أن يشير الخيط أولًا إلى أنه يريد الوصول عن طريق طلب الحصول على قفل كائن المزامنة mutex's lock؛ والقفل هو هيكل بيانات data structure يعد جزءًا من كائن المزامنة الذي يتتبع من لديه حاليًا وصولٌ حصري إلى البيانات، وبالتالي يُوصَف كائن المزامنة بأنه يحمي البيانات التي يحملها عبر نظام القفل.

لكائنات المزامنة سمعة بأنها صعبة الاستعمال لأنك يجب أن تتذكر قاعدتين:

- يجب أن تحاول الحصول على القفل قبل استعمال البيانات.

- يجب أن تلغي قفل البيانات عندما تنتهي من البيانات التي يحميها كائن المزامنة حتى يتسنّى للخيوط الأخرى الحصول على القفل.

لنأخذ تشبيهًا حقيقيًا لكائن المزامنة: تخيل حلقة نقاش في مؤتمر بميكروفون واحد فقط، إذ يجب على أحد أعضاء اللجنة أن يسأل أو يشير إلى أنه يريد استخدام الميكروفون عند رغبته بالتحدث وعندما يحصل على الميكروفون يمكنه التحدث بقدر ما يريد من الوقت ثم يسلّم الميكروفون إلى عضو اللجنة التالي الذي يطلب التحدث؛ وإذا نسي أحد أعضاء اللجنة تسليم الميكروفون عند الانتهاء منه فلن يتمكن أي شخص آخر من التحدث؛ إذا حدث خطأ في إدارة الميكروفون المشترك فلن تعمل حلقة النقاش على النحو المطلوب.

قد تكون إدارة كائنات المزامنة صعبة جدًا وهذا هو سبب تفضيل الكثير من الناس للقنوات، ومع ذلك لا يمكنك الحصول على قفل وفتحه بصورةٍ خاطئة في رست بفضل نظامها وقواعد ملكيتها الخاصة.

ا. واجهة Mutex<T> البرمجية

لنبدأ باستعمال كائن مزامنة بسياق خيط وحيد single-threaded مثالًا عن كيفية استعمال كائن مزامنة كما هو موضح في الشيفرة 12:

اسم الملف: src/main.rs

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m = {:?}", m);
}
```

[الشيفرة 12: تجربة واجهة Mutex<T> البرمجية بسياق خيط وحيد لبساطته]

كما هو الحال مع العديد من الأنواع نُنشئ Mutex<T> باستعمال الدالة المرتبطة new، ونستخدم التابع lock للوصول إلى البيانات داخل كائن المزامنة وذلك للحصول على القفل. سيحظر هذا الاستدعاء الخيط الحالي ولن تتمكن من فعل أي عمل حتى يحين دور الحصول على القفل.

يفشل استدعاء lock إذا هلع خيط آخر يحمل القفل، وفي هذه الحالة لن يتمكن أي شخص إطلاقًا من الحصول على القفل لذلك اخترنا unwrap بحيث يهلع الخيط هذا إذا حصلت هذه الحالة.

يمكننا أن نعالج القيمة التي حصلنا عليها التي تحمل الاسم num بعد حصولنا على القفل وستكون في هذه الحالة بمثابة مرجع متغير mutable يُشير إلى البيانات الموجودة بالداخل. يضمن نظام النوع type system حصولنا على قفل قبل استخدام القيمة في m، ونوع m هو Mutex<i32> وليس i32 لذلك يجب علينا استدعاء lock حتى نتمكن من استخدام القيمة i32، ودعنا لا ننسى أن نظام النوع لن يسمح لنا بالوصول إلى قيمة i32 الداخلية بخلاف ذلك.

كما تعتقد فإن Mutex<T> هو مؤشر ذكي، وبدقة أكبر، يُعيد استدعاء lock مؤشرًا ذكيًا يدعى MutexGuard مُغلَّفًا في LockResult الذي تعاملنا معه في استدعاء unwrap، يُطبّق المؤشر الذكي MutexGuard السمة Deref ليشير إلى بياناتنا الداخلية، كما يحتوي المؤشر الذكي أيضًا على تطبيق للسمة Drop يُحرّر القفل تلقائيًا عندما يخرج MutexGuard عن النطاق وهو الأمر الذي يحدث بنهاية النطاق الداخلي. نتيجة لذلك لا نخاطر بنسيان تحرير القفل وحجب استخدام كائن المزامنة بواسطة الخيوط الأخرى لأن تحرير القفل يحدث تلقائيًا.

يمكننا طباعة قيمة كائن المزامنة بعد تحرير القفل وسنرى أننا تمكنا من تغيير القيمة ذات النوعi32 الداخلية إلى 6.

ب. مشاركة Mutex<T> بين خيوط متعددة

دعنا نحاول الآن مشاركة قيمة بين خيوط متعددة باستخدام Mutex<T>، سنمرّ على عشرة خيوط ونجعل كل خيط منها يزيد قيمة العداد بمقدار 1 بحيث ينتقل العداد من القيمة 0 إلى 10. يحتوي المثال التالي في الشيفرة 13 على خطأ تصريفي compiler error، وسنستفيد من هذا الخطأ حتى نتعلم المزيد حول استعمال Mutex<T> وكيف سيساعدنا رست في استعماله بصورةٍ صحيحة.

اسم الملف: src/main.rs

```
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
```

```
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

[الشيفرة 13: عشرة خيوط يزيد كل واحد منها عداد محميّ باستخدام `Mutex <T>`]

ننشئ متغيرًا ندعوه `counter` ليحمل قيمة من النوع `i32` داخل `Mutex<T>` كما فعلنا في الشيفرة 12، بعد ذلك نُنشئ عشرة خيوط عبر المرور iterate على مجال من الأرقام، ونستخدم لتحقيق ذلك `thread::spawn` ونمنح كل خيط المغلّف ذاته الذي ينقل العداد باتجاه الخيط ويحصل على قفل على `Mutex<T>` عن طريق استدعاء التابع `lock` ومن ثم يضيف القيمة 1 إلى القيمة الموجودة في كائن المزامنة. عندما ينتهي الخيط من تنفيذ مغلّفه يخرج `num` عن النطاق ويحرر القفل بحيث يستطيع خيطٌ آخر الحصول عليه.

نجمع كل مقابض الانضمام join handles في الخيط الرئيسي، ونستدعي بعد ذلك -كما فعلنا في الشيفرة 2 سابقًا- `join` على كل مقبض للتأكد من انتهاء جميع الخيوط، وعند هذه النقطة يحصل الخيط الرئيسي على القفل ويطبع نتيجة هذا البرنامج.

لمّحنا إلى أن هذا المثال لن يُصرَّف، لنتعرف على السبب الآن:

```
$ cargo run
   Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0382]: use of moved value: `counter`
  --> src/main.rs:9:36
   |
|     let counter = Mutex::new(0);
   |         ------- move occurs because `counter` has type
`Mutex<i32>`, which does not implement the `Copy` trait
```

```
...
|         let handle = thread::spawn(move || {
   |                                    ^^^^^^^ value moved into
closure here, in previous iteration of loop
|             let mut num = counter.lock().unwrap();
   |                            ------- use occurs due to use in
closure

For more information about this error, try `rustc --explain E0382`.
error: could not compile `shared-state` due to previous error
```

تشير رسالة الخطأ إلى أن قيمة `counter` نُقِلت في التكرار السابق للحلقة، وتخبرنا رست أنه لا يمكننا نقل ملكية قفل `counter` إلى خيوط متعددة. لنصحّح الخطأ التصريفي بطريقة الملكية المتعددة التي ناقشناها سابقًا في الفصل 15.

ج. ملكية متعددة مع خيوط متعددة

تكلمنا سابقًا في الفصل الخامس عشر عن المالكين المتعددين لقيمة باستخدام المؤشر الذكي Rc لإنشاء قيمة مرجعية معدودة، لننفّذ الشيء ذاته هنا ونلاحظ النتيجة. نغلّف `Mutex<T>` داخل `Rc <T>` ضمن الشيفرة 14 ونستنسخ `Rc<T>` قبل نقل الملكية إلى الخيط.

اسم الملف: src/main.rs

```rust
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
```

```
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

[الشيفرة 14: محاولة استعمال `Rc<T>` للسماح لخيوط متعددة بامتلاك `Mutex<T>`]

نصرّف الشيفرة البرمجية مـرةً أخـرى، ونحصـل هـذه المـرة على أخطـاء مختلفـة، ويعلّمنـا المصـرّف الكثـير من الأشياء.

```
$ cargo run
   Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0277]: `Rc<Mutex<i32>>` cannot be sent between threads safely
  --> src/main.rs:11:36
   |
|           let handle = thread::spawn(move || {
   |                        ------------- ^------
   |                        |             |
   |  ______________________|_____________within this
`[closure@src/main.rs:11:36: 11:43]`
   | |                      |
   | |                      required by a bound introduced by this
call
| |             let mut num = counter.lock().unwrap();
| |
| |             *num += 1;
| |         });
   | |_________^ `Rc<Mutex<i32>>` cannot be sent between threads
safely
   |
   = help: within `[closure@src/main.rs:11:36: 11:43]`, the trait
`Send` is not implemented for `Rc<Mutex<i32>>`
note: required because it's used within this closure
```

```
  --> src/main.rs:11:36
   |
|         let handle = thread::spawn(move || {
   |                                    ^^^^^^^
note: required by a bound in `spawn`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `shared-state` due to previous error
```

رسالة الخطأ هذه شديدة التعقيد، إليك الجزء المهم الذي يجب أن تركز عليه:

```
`Rc<Mutex<i32>>` cannot be sent between threads safely
```

يخبرنا المصرّف أيضًا عن السبب:

```
the trait `Send` is not implemented for `Rc<Mutex<i32>>`
```

سنتحدث عن Send في القسم التالي، إذ أنها أحد السمات التي تضمن أن الأنواع التي نستعملها مع الخيوط مخصصة للاستخدام في الحالات المتزامنة.

لسوء الحظ فإن Rc<T> ليس آمنًا للمشاركة عبر الخيوط، فعندما يُدير Rc<T> عدد المراجع فإنه يضيف عدد كل استدعاء إلى clone ويطرح من العدد عندما تُحرَّر كل نسخة clone، إلا أنه لا يستعمل أي أنواع تزامن أولية للتأكد من أن التغييرات التي حدثت على العدد لا يمكن مقاطعتها بواسطة خيط آخر. قد يؤدي هذا إلى عمليات عدّ خاطئة -أخطاء خفية يمكن أن تؤدي بدورها إلى تسريب الذاكرة memory leak أو تحرير قيمة ما قبل أن ننتهي منها- وما نحتاجه هنا هو نوع مثل Rc<T> تمامًا ولكنه نوع يُجري تغييرات على عدد المراجع بطريقة آمنة للخيوط.

د. عد المراجع الذري باستخدام Arc<T>

لحسن الحظ، يعد Arc<T> نوعًا مثل Rc<T>وهو آمن للاستخدام في الحالات المتزامنة، إذ يرمز الحرف a إلى ذرّي atomic مما يعني أنه يُعد نوع عدّ مرجع ذري atomically reference counted type. تعدّ الأنواع الذرية Atomics نوعًا إضافيًا من أنواع التزامن الأولية concurrency primitive التي لن نغطيها بالتفصيل هنا. راجع توثيق المكتبة القياسية std::sync::atomic للمزيد من التفاصيل، من الكافي الآن معرفة أن الأنواع الذرية تعمل مثل الأنواع الأولية ولكن من الآمن مشاركتها عبر الخيوط.

قد تتساءل عن عدم كون جميع الأنواع الأولية ذرية ولماذا أنواع المكتبات القياسية غير مُطبّقة لاستخدام Arc<T> افتراضيًا، والسبب هنا هو أن سلامة الخيوط تأتي مع ضريبة أداء تريد دفعها فقط عندما تحتاج إليها

حقًا. إذا كنت تجري عمليات على القيم ضمن خيط واحد فيمكن لشيفرتك البرمجية أن تُنفَّذ بصورةٍ أسـرع إذا لم تكن بحاجة لفرض الضمانات التي تقدمها الأنواع الذرية.

بالعودة إلى مثالنا: للنوعين `Arc<T>` و `Rc<T>` الواجهة البرمجية API ذاتها لذلك نصحّح برنامجنا عن طريـق تغيير سطر `use` واستدعاء `new` وكذلك استدعاء `clone`. نستطيع أخيرًا تصريف الشيفرة البرمجيـة في الشـيفرة 15 وتنفيذها.

اسم الملف: src/main.rs

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

[الشيفرة 15: استخدام `Arc<T>` لتغليف `Mutex<T>` بحيث نستطيع مشاركة الملكية عبر خيوط متعددة]

يكون خرج الشيفرة البرمجية السابقة على النحو التالي:

```
Result: 10
```

وأخيرًا نجحنا، فقد عدّدنا من 0 إلى 10، وعلى الرغم من أن هذا قد لا يبدو رائعًا جدًا، إلا أنه علّمنا الكثير عن `Mutex<T>` وأمان الخيط. يمكنك أيضًا استخدام هيكل هذا البرنامج لإجراء عمليات أكثر تعقيدًا من مجرد زيادة العداد، ويمكننا باستخدام هذه الإستراتيجية تقسيم عملية حسابية إلى أجزاء مستقلة وتقسيم تلك الأجزاء عبر خيوط ثم استخدام `Mutex<T>` لجعل كل خيط يحدّث النتيجة النهائية بجزئه.

لاحظ أنه إذا كنت تنفّذ عمليات عددية بسيطة فهناك أنواع أبسط من أنواع `Mutex<T>` التي توفرها وحدة المكتبة القياسية std::sync::atomic، إذ توفر هذه الأنواع وصولًا ذريًا آمنًا ومتزامنًا للأنواع الأولية، وقد اخترنا استخدام `Mutex<T>` مع نوع أولي لهذا المثال حتى نتمكن من التركيز على كيفية عمل `Mutex<T>`.

16.3.2 التشابه بين `RefCell<T>/Rc<T>` و `Mutex<T>/Arc<T>`

ربما لاحظت أن `counter` ثابت immutable ولكن يمكننا الحصول على مرجع متغيّر للقيمة الموجودة داخله، هذا يعني أن `Mutex<T>` يوفر قابلية تغيير داخلية كما تفعله عائلة `Cell`. نستخدم `Mutex<T>` بنفس الطريقة التي استخدمنا بها `RefCell<T>` سابقًا في الفصل 15 للسماح لنا بتغيير المحتويات داخل `Rc<T>` وذلك لتغيير المحتويات داخل `Arc<T>`.

هناك شيء آخر يجب ملاحظته ألا وهو أن رست لا يمكنها حمايتك من جميع أنواع الأخطاء المنطقية عند استخدام `Mutex<T>`. تذكر سابقًا في الفصل 15 أن استخدام `Rc<T>` يأتي مع خطر إنشاء دورات مرجعية reference cycle، إذ تشير قيمتان `Rc<T>` إلى بعضهما بعضًا مما يتسبب في حدوث تسرب في الذاكرة. وبالمثل يأتي تطبيق `Mutex<T>` مع خطر خلق حالات مستعصية deadlocks، إذ تحدث هذه الحالات عندما تحتاج عملية ما إلى الحصول على مرجعين وحصل كل منهما على أحد الأقفال مما يتسبب في انتظارهما لبعضهما بعضًا إلى الأبد.

إذا أثارت الحالات المستعصية اهتمامك وأردت رؤيتها عمليًا فحاول إنشاء برنامج رست به حالة مستعصية، ثم ابحث عن استراتيجيات التخفيف من الحالات المستعصية بالنسبة لكائنات المزامنة في أي لغة ونفذها في رست. يوفر توثيق واجهة المكتبة القياسية البرمجية لـ `Mutex<T>` و `MutexGuard` معلومات مفيدة.

سنكمل هذا الفصل بالحديث عن السمتين `Send` و `Sync` وكيف يمكننا استخدامها مع الأنواع المخصصة.

16.4 التزامن الموسع مع السمة Sync والسمة Send

من المثير للاهتمام أن لغة رست تحتوي على عدد قليل جدًا من ميزات التزامن، إذ تعد كل ميزة تزامن تحدثنا عنها حتى الآن في هذا الفصل جزءًا من المكتبة القياسية standard library وليست اللغة. لا تقتصر خياراتك للتعامل مع التزامن على اللغة أو المكتبة القياسية فيمكنك كتابة ميزات التزامن الخاصة بك أو استخدام تلك المكتوبة من قبل الآخرين.

ومع ذلك، ضُمِّنَ مفهومان للتزامن في اللغة: سمتَا `std::marker` وهما `Sync` و `Send`.

16.4.1 السماح بنقل الملكية بين الخيوط عن طريق Send

تشير الكلمة المفتاحية للسمة `Send` إلى أنه يمكن نقل ملكية القيم من النوع الذي ينفّذ `Send` بين الخيوط، ويطبّق كل نوع في رست تقريبًا السمة `Send`، ولكن هناك بعض الاستثناءات بما في ذلك `Rc<T>` إذ لا يمكن تطبيق السمة `Send` عليه لأنه إذا استنسخت قيمة `Rc<T>` وحاولت نقل ملكية الاستنساخ إلى خيط آخر فقد يحدّث كلا الخيطين عدد المراجع reference count في الوقت ذاته. لهذا السبب تُنفَّذ `Rc<T>` للاستخدام في المواقف أحادية الخيط حيث لا تريد دفع غرامة الأداء الآمن.

لذلك يؤكد نظام نوع رست وحدود السمات trait bounds أنه لا يمكنك أبدًا إرسال قيمة `Rc<T>` بطريق الخطأ عبر الخيوط بصورةٍ غير آمنة. عندما حاولنا فعل ذلك في الشيفرة 14 سابقًا حصلنا على الخطأ:

```
the trait Send is not implemented for Rc<Mutex<i32>>
```

وعندما استبدلنا النوع بالنوع `Arc<T>` الذي يطبّق السمة `Send` صُرِّفَت الشيفرة البرمجية بنجاح.

أي نوع مكون بالكامل من أنواع `Send` يُميَّز تلقائيًا على أنه `Send` أيضًا، وتطبّق جميع الأنواع الأولية primitive types تقريبًا السمة `Send` بغض النظر عن المؤشرات الأولية primitive pointers التي سنناقشها لاحقًا في الفصل 19.

16.4.2 السماح بالوصول لخيوط متعددة باستخدام السمة Sync

تشير الكلمة المفتاحية للسمة `Sync` إلى أنه من الآمن للنوع المطبّق للسمة `Sync` أن يُشار إليه من خيوط متعددة، بمعنى آخر أي نوع `T` يطبّق السمة `Sync` إذا كان `&T` (مرجع غير قابل للتغيير إلى `T`) يطبّق `Send` أيضًا، مما يعني أنه يمكن إرسال المرجع بأمان إلى خيط آخر، وبصورةٍ مشابهة للسمة `Send` فإن الأنواع الأولية تطبّق السمة `Sync` والأنواع المكونة كاملًا من أنواع تطبّق السمة `Sync` هي أيضًا أنواع تطبّق `Sync`.

لا يطبّق المؤشر الذكي `Rc<T>` أيضًا السمة `Sync` للأسباب ذاتها التي تجعل منه غير قابل لتطبيق السمة `Send`، كما أن النمط `RefCell<T>` (الذي تحدثنا عنه سابقًا في الفصل 15) وعائلة الأنواع المرتبطة بالنوع `Cell<T>` لا تطبّق السمة `Sync`. يعدّ تنفيذ فحص الاستعارة الذي تفعله `RefCell<T>` في وقت التنفيذ غير آمن للخيط. يطبّق المؤشر الذكي `Mutex<T>` السمة `Sync` ويمكن استخدامه لمشاركة الوصول مع خيوط متعددة كما رأيت سابقًا في قسم "مشاركة `Mutex<T>` بين خيوط متعددة".

16.4.3 تطبيق السمتين Send و Sync يدويا غير آمن

بما أن الأنواع المكونة من الأنواع التي تطبّق السمتين Send وSync هي أنواع تطبّق السمتين Send و Sync تلقائيًا، فلا يتوجب علينا تطبيق هاتين السمتين يدويًا؛ وبصفتهما سمتان علّامة marker traits، فهما سمتان لا تحتويان أيّ توابع تطبّقها، وهما مفيدتان فقط لتعزيز الثوابت المرتبطة بالمزامنة.

يشمل تنفيذ هاتان السمتان يدويًا تنفيذ شيفرة رست غير آمنة، وسنتحدث عن استعمال شيفرة رست غير آمنة لاحقًا في الفصل 19، ومن الكافي الآن معرفتك أن بناء أنواع متزامنة جديدة غير مؤلفة من أجزاء Send و Sync يتطلّب تفكيرًا حذرًا للمحافظة على ضمانات الأمان في لغة رست. يحتوي الكتاب "The Rustonomicon" معلومات أكثر عن هذه الضمانات وكيف يمكن التمسك بها.

16.5 الخاتمة

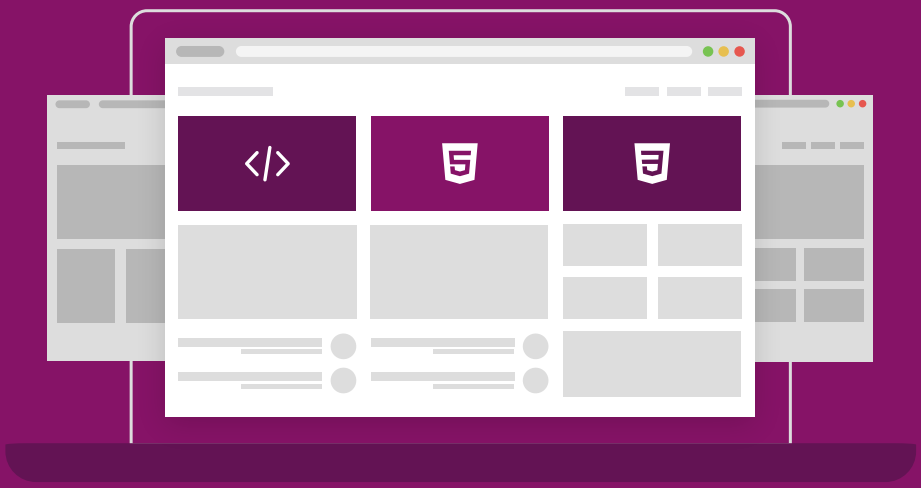
لن يكون هذا الفصل الذكر الأخير للتزامن في هذا الكتاب، إذ سيستخدم المشروع في الفصل 20 المفاهيم الواردة في هذا الفصل في مواقف أكثر واقعية من الأمثلة الصغيرة التي ناقشناها هنا.

كما ذكرنا سابقًا، نظرًا لأن القليل جدًا من كيفية تعامل رست مع التزامن هو جزء من اللغة، تُطبّق العديد من حلول التزامن بمثابة وحدات مصرفة crate. يتطور هذا الأمر بسرعة أكبر من المكتبة القياسية لذا تأكد من البحث عبر الإنترنت عن الوحدات المصرفة الحالية الأكثر حداثة لاستخدامها في المواقف متعددة الخيوط.

توفر مكتبة رست القياسية قنوات channels لتمرير الرسائل وأنواع مؤشرات الذكية، مثل `Mutex<T>` و `Arc<T>` التي تعد آمنة للاستخدام في السياقات المتزامنة. يؤكد نظام الكتابة وفاحص الاستعارة أن الشيفرة البرمجية التي تستعمل هذه الحلول لن ينتهي بها المطاف بحالة سباق بيانات data race أو مراجع غير صالحة. بمجرد الحصول على الشيفرة البرمجية الخاصة بك لتصريفها، يمكنك أن تطمئن إلى أنهها ستُنفَّذ بسلاسة على خيوط متعددة بدون أنواع الأخطاء شائعة الحدوث في لغات البرمجة الأخرى التي يصعب تتبعها. لم تعد البرمجة المتزامنة مفهومًا يجب الخوف منه لذا انطلق واجعل برامجك متزامنة بلا خوف.

بعد ذلك سنتحدث عن الطرق الاصطلاحية لنمذجة المشكلات والحلول الهيكلية مع زيادة حجم برامج رست الخاصة بك. إضافةً إلى ذلك سنناقش كيفية ارتباط مصطلحات رست بتلك التي قد تكون مألوفة لك من البرمجة الموجهة للأشياء.

دورة تطوير واجهات المستخدم



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

17. مزايا البرمجة كائنية التوجه

Object-Oriented Programming

تُعد البرمجة كائنية التوجه Object-oriented programming -أو اختصارًا OOP- وسيلةً لنمذجة البرامج، وقد قُدِّمَت الكائنات بمثابة مفهوم برمجي في لغة البرمجة سيمولا Simula في ستينيات القرن الماضي، إذ أثّرت تلك الكائنات على البنية البرمجية الخاصة بآلان كاي Alan kay بطريقة تمرير الكائنات رسائل لبعضها بعضًا، ولوصف هذه البنية صاغ مصطلح البرمجة كائنية التوجه في 1967. تصف العديد من التعريفات المشابهة ما يُقصد بمصطلح OOP ومن خلال بعض هذه التعريفات تعد رست لغة برمجة كائنية التوجه ولكنها ليست كذلك مقارنةً بالتعريفات الأخرى. سنكتشف في هذا الفصل مزايا محددة تُعدّ كائنية التوجه وكيف تُتَرجم هذه المميزات للغة رست الاصطلاحية. سنستعرض بعد هذا كيفية تنفيذ نمط تصميم design pattern موجّه للكائنات في رست، وسنناقش مقايضات فعل ذلك مقابل تنفيذ حل باستعمال بعض نقاط قوة رست بدلًا من ذلك.

17.1 مزايا البرمجة كائنية التوجه OOP

لا يوجد إجماع في مجتمع البرمجة حول الميزات التي يجب أن تكون موجودة في لغة البرمجة حتى تكون لغة كائنية التوجه، وتتأثر رست بالعديد من نماذج البرمجة programming paradigms بما في ذلك البرمجة كائنية التوجه، إذ اكتشفنا الميزات التي جاءت من البرمجة الوظيفية functional programming سابقًا بدءًا من الفصل 13. يمكن القول أن اللغات كائنية التوجه تشترك ببعض الميزات المشتركة وهي الكائنات objects والتغليف encapsulation والوراثة inheritance، لنلقي نظرةً على ما تعنيه كل من هذه الميزات وما إذا كانت رست تدعمها.

17.1.1 الكائنات واحتوائها على بيانات وسلوك

يُعـدّ الكتـاب "أنمـاط التصـميم: عناصـر البرمجيـات الموجهـة للكائنـات القابلـة لإعـادة الاسـتعمال Design Patterns: Elements of "Reusable Object-Oriented Software للمؤلفين إريك جاما Erich Gamma،وريتشـارد هيلم Richard Helm، ورالـف جونسـون Ralph Johnson، وجـون فليسـيديس John Vlissides، التابع لدار النشر (Addison-Wesley Professional, 1994)، والـذي يشـار إليـه بالعاميـة كتـاب عصابة الأربعة The Gang of Four book، فهرسًا لأنماط التصميم كائنية التوجه، ويعـرّف الكتـاب البرمجـة كائنية التوجّه بهذه الطريقة:

> تتكون البرامج كائنية التوجه من كائنات، ويضمّ الكائن داخله كل من البيانات والإجراءات procedures التي تستخدم تلك البيانات، وتدعى الإجراءات عادًة بالتوابع methods أو العمليات operations.

بالنظر للتعريف السابق تكون رست لغة كائنيـة التوجـه؛ إذ تحتـوي الهياكـل structs والتعـدادات enums على البيانات، وتقدّم كتل `impl` توابعًا على الهياكل والتعـدادات، وعلى الـرغم من أن الهياكـل والتعـدادات ذات التوابع لا تدعى بالكائنات إلا أنها تقدّم الوظيفة نفسها وذلك بحسب تعريف الكائنات في كتاب عصابة الأربعة.

ويمكنك الرجوع إلى توثيق أنماط التصميم العربي في موسوعة حسوب لمزيد من التفاصيل.

17.1.2 التغليف وإخفاءه لتفاصيل التنفيذ

هناك جانبٌ آخر مرتبط جدًا بالبرمجة كائنية التوجه وهو فكرة التغليف، والتي تعني أن تفاصيل تطبيق كـائن ما لا يمكنها الوصول للشيفرة البرمجية من خلال هذا الكائن، لذا فإن الطريقة الوحيدة للتفاعـل مـع كـائن مـا هي من خلال واجهة برمجية عامة Public API خاصة به، وينبغي ألا تكون الشيفرة البرمجية التي يستخدمها الكـائن قادرةً على الوصول إلى الأجزاء الداخلية للكائن وتغيير البيانات أو السلوك مباشرةً، وهذا يمكّن المبرمج من تغيـير وإعادة تشكيل العناصر الداخلية للكائن دون الحاجة إلى تغيير الشيفرة البرمجية التي تستخدم الكائن.

ناقشنا كيفية التحكم في التغليـف سـابقًا بـدءًا من الفصـل 7، إذ يمكننـا اسـتخدام الكلمـة المفتاحيـة `pub` لتحديـد أي من الوحـدات modules والأنـواع types والـدوال functions والتوابـع methods في الشـيفرات البرمجية الخاصة بنا التي ينبغي أن تكون عامة، ويكون كل شيء آخر خاص افتراضيًا، فعلى سبيل المثال يمكننـا تعريف هيكل `AveragedCollection` يحتوي على حقل يضمّ شـعاعًا vector بقيم `i32`، كمـا يمكن للهيكـل أيضًا أن يحتوي على حقل يضمّ متوسط القيم في الشعاع مما يعني أنه لا لـزوم لحسـاب المتوسـط عنـد الطلب كلما احتاجه أي أحد. بعبارة أخرى سيخزّن `AveragedCollection` المتوسـط النـاتج. تحتـوي الشـيفرة 1 على تعريف لهيكل `AveragedCollection`:

اسم الملف: src/lib.rs

```
pub struct AveragedCollection {
```

```
    list: Vec<i32>,
    average: f64,
}
```

[الشيفرة 1: هيكل `AveragedCollection` الذي يخزّن قائمة من الأعداد الصحيحة والمتوسط لعناصر التجميعة]

الهيكل مُشار إليه بالكلمة المفتاحية `pub`، بحيث يمكن لشيفرة برمجية أخرى استخدام ذلك الهيكل، إلا أن الحقول الموجودة داخل الهيكل تبقى خاصة. هذا مهمٌ في هذه الحالة لأننا نريد التأكد من أنه كلما أُضيفت قيمة أو أُزيلت من الشيفرة يُحَدَّث المتوسط أيضًا، ونحقق ذلك من خلال تطبيق توابع `add` و `remove` و `average` كما هو موضح في الشيفرة 2:

اسم الملف: src/lib.rs

```
impl AveragedCollection {
    pub fn add(&mut self, value: i32) {
        self.list.push(value);
        self.update_average();
    }

    pub fn remove(&mut self) -> Option<i32> {
        let result = self.list.pop();
        match result {
            Some(value) => {
                self.update_average();
                Some(value)
            }
            None => None,
        }
    }

    pub fn average(&self) -> f64 {
        self.average
    }

    fn update_average(&mut self) {
        let total: i32 = self.list.iter().sum();
        self.average = total as f64 / self.list.len() as f64;
```

```
        }
    }
```

[الشيفرة 2: تطبيق التوابع العامة `add` و`remove` و`average` على `AveragedCollection`]

تعدّ التوابع العامة `add` و `remove` و `average` الوسائل الوحيدة للوصول إلى البيانات أو تعديلها في نسخ من `AveragedCollection`. يُستدعى التابع الخاص `update_average` عندما يضاف عنصر على `list` باستخدام التابع `add` أو يُزال باستخدام التابع `remove`، وهو التابع الذي يحدّث حقل `average` بدوره.

نترك حقول `list` و `average` خاصة لكي لا يبقى أي وسيلة للشيفرة برمجية الخارجية أن تضيف أو تزيل عناصر إلى أو من حقل `list` مباشرةً، وإلّا، يمكن للحقل `average` ألّا يتوافق مع القيم عندما تتغير `list`. يعيد تابع `average` القيمة في حقل `average` مما يسمح للشيفرة البرمجية الخارجية أن تقرأ `average` دون أن تعدل عليها.

بما أننا غلفنا تفاصيل تنفيذ الهيكل `AveragedCollection`، يمكننا بسهولة مستقبلًا تغيير بعض التفاصيل مثل هيكل البيانات، فعلى سبيل المثال، يمكننا استعمال `HashSet<i32>` بدلًا من `Vec<i32>` لحقل `list`. بما أن بصمات التوابع العامة `add` و `remove` و `average` بقيت على حالها، فلا ضرورة للتعديل على الشيفرة البرمجية التي تستخدم `AveragedCollection`. ولكنّ هذا الأمر لن يكون محققًا إذا جعلنا `list` عامة بدلًا من ذلك، إذ تملك `HashSet<i32>` و `Vec<i32>` توابعًا مختلفة لإضافة وإزالة العناصر بحيث ستحتاج إلى تغيير الشيفرة البرمجية الخارجية غالبًا في حال كانت تعدل على `list` مباشرةً.

إذا كان التغليف جزءًا مطلوبًا للغة البرمجة حتى تصبح لغة كائنية التوجه، فإن رست تلبي هذا المطلب، إذ يتيح خيار استعمال `pub` أو عدم استعماله لأجزاء مختلفة من الشيفرة، التغليف لتفاصيل التنفيذ.

17.1.3 الوراثة واستخدامها مثل نظام نوع ومشاركة الشيفرة البرمجية

الوراثة هي آلية يمكن بواسطتها للكائن أن يرث عناصر من تعريف كائن آخر وبالتالي يكتسب بيانات الكائن الأصل وسلوكه دون الحاجة إلى تعريفها مرةً أخرى.

إذا كانت الوراثة متطلبًا للغة برمجية حتى تكون اللغة كائنية التوجه فإن رست ليست بلغة كائنية التوجه، إذ لا توجد طريقة لتعريف هيكل يرث حقول وتطبيقات تابع الهيكل الأصلي دون استخدام ماكرو، ومع ذلك إذا كنت معتادًا على وجود الوراثة في اللغة البرمجية التي تتعامل معها، فيمكنك استخدام حلول أخرى في رست بحسب سبب حاجتك للوراثة.

قد تحتاج للتوريث لسببين رئيسيين؛ أحدهما لإعادة استعمال الشيفرة البرمجية، ويمكنك في هذه الحالة تنفيذ سلوك معين لنوع واحد، ويمكّنُك التوريث بدوره من إعادة استعمال هذا التنفيذ لنوع مختلف. يمكنك استخدام هذه الوسيلة بطريقة محدودة في شيفرة رست باستخدام تطبيقات تابع السمة الافتراضية التي رأيتها

في الشيفرة 14 من الفصل 10 عندما أضفنا تنفيذًا افتراضيًا لتابع `summarize` على السمة `Summary`. سيُتاح لأي نوع يطبق السمة `Summary` التابع `summarize` دون أي شيفرة برمجية إضافية، وهذا مشابه للصنف الأب parent class الذي يحتوي على تنفيذ لتابع وصنف ابن يرث الصف الأب ويحتوي على تنفيذ التابع. يمكننا أيضًا تجاوز التطبيق الافتراضي لتابع `summarize` عندما نطبق السمة `Summary` التي تشبه الصنف الابن الذي يُعيد تعريف تابع موروث من صنف أب.

تتعلق الحاجة الأخرى لاستخدام التوريث بنظام النوع بتمكين استعمال نوع فرعي في نفس الأماكن مثل النوع الأصل. يسمى هذا أيضًا التعددية الشكلية polymorphism مما يعني أنه يمكنك استبدال كائنات متعددة ببعضها في وقت التنفيذ إذا كانت تشترك في خصائص معينة.

التعددية الشكلية Polymorphism

ينظر الكثير من الناس إلى التعددية الشكلية polymorphism بكونها مشابهة للوراثة لكنها فعليًا مفهوم أوسع، إذ تشير إلى الشيفرة البرمجية التي يمكنها العمل مع بيانات ذات أنواع مختلفة، بينما تكون هذه الأنواع بالنسبة للوراثة أصنافًا فرعيةً subclass عمومًا.

يستخدم رست أنواع معممة generics بدلًا من هذا لتجريد الأنواع المختلفة الممكنة وحدود السمات trait bounds، وذلك لفرض قيود على ما يجب أن توفره هذه الأنواع، إذ يسمى ذلك أحيانًا بالتعددية الشكلية المحدودة المقيّدة bounded parametric polymorphism.

فقد التوريث مكانته مؤخرًا مثل حل برمجي تصميمي في العديد من لغات البرمجة لأنه غالبًا ما يكون عرضةً لخطر مشاركة شيفرة برمجية زيادةً عن اللزوم. لا يجب أن تشترك الأصناف الابن دائمًا في جميع خصائص صنفها الأب ولكنها ستفعل ذلك مع التوريث، ومن شأن ذلك جعل تصميم البرنامج أقل مرونة. يُضيف ذلك أيضًا إمكانية استدعاء توابع على الأصناف الابن التي لا معنى لها أو التي تتسبب بأخطاء لأن التوابع لا تنطبق على الأصناف الابن. إضافةً إلى ذلك تسمح بعض اللغات فقط بالوراثة الفردية single inheritance (بمعنى أنه يمكن للصنف الابن أن يرث فقط من صنف واحد) مما يقيد مرونة تصميم البرنامج أكثر.

تتخذ رست لهذه الأسباب طريقةً مختلفة في استعمال كائنات السمة بدلاً من الوراثة، وسنلقي نظرةً على كيفية تمكين كائنات السمة من تعدد الأشكال في رست.

17.2 استخدام كائنات السمة Object Trait

ذكرنا سابقًا في الفصل الثامن وما بعده أن أحد قيود الشعاع vector هي تخزينه لعناصر من نوع واحد فقط، وقد أنشأنا حلًا بديلًا فيما بعد في الشيفرة 8 من الفصل 9، إذ عرّفنا التعداد `SpreadsheetCell` وداخله متغايرات variants تحتوي على أعداد صحيحة integers وعشرية floats ونص text، وهذا يعني أنه يمكننا تخزين أنواع مختلفة من البيانات في كل خلية مع المحافظة على شعاع يمثل صفًا من الخلايا. يعد هذا حلًا جيدًا

عندما تمثّل العناصر القابلة للتبديل مجموعةً ثابتةً من الأنواع التي نعرّفها عند تصريف الشيفرة البرمجية الخاصة بنا.

نريد أحيانًا أن يتمكن مستخدم مكتبتنا من توسيع مجموعة الأنواع الصالحة في حالة معينة، ولإظهار كيف يمكننا تحقيق ذلك سننشئ مثالًا لأداة واجهة المستخدم الرسومية graphical user interface -أو اختصارًا GUI- التي تتكرر من خلال قائمة من العناصر مستدعيةً تابع `draw` على كل عنصر لرسمه على الشاشة، وهي تقنية شائعة لأدوات واجهة المستخدم الرسومية. سننشئ وحدة مكتبة مصرّفة crate library تدعى `gui` تحتوي على هيكل مكتبة لأدوات واجهة المستخدم الرسومية GUI، وقد تتضمن هذه الوحدة المصرّفة بعض الأنواع ليستخدمها الأشخاص، مثل `Button`، أو `TextField`، كما سيرغب مستخدمو `gui` بإنشاء أنواعهم الخاصة التي يمكن رسمها، فعلى سبيل المثال قد يضيف أحد المبرمجين `Image` وقد يضيف آخر `SelectBox`.

لن نبرمج كامل مكتبة GUI في هذا المثال لكننا سنبين كيف ستعمل الأجزاء معًا. لا يمكننا في وقت كتابة المكتبة معرفة وتعريف جميع الأنواع التي قد يرغب المبرمجون الآخرون بإنشائها، لكننا نعلم أن `gui` تحتاج إلى تتبُّع العديد من القيم ومن أنواع مختلفة وتحتاج إلى استدعاء تابع `draw` على كل من هذه القيم المكتوبة بصورةٍ مختلفة. لا يتطلب الأمر معرفة ماذا سيحدث فعلًا عندما نستدعي تابع `draw`، ومن الكافي معرفة أن القيمة ستحتوي على تابع متاح ضمنها يمكننا استدعاؤه.

لتحقيق هذا الأمر في لغة برمجة تحتوي على خاصية التوريث قد نعرّف صنفًا يدعى `Component` له تابع يسمى `draw`. ترث الأصناف الأخرى، مثل `Button`، و `Image`، و `SelectBox` من `Component`، وبالتالي ترث تابع `draw`. يمكن لكل من الأصناف السابقة إعادة تعريف تابع `draw` لتعريف سلوكهم المخصص، لكن إطار العمل framework قد يتعامل مع جميع الأنواع كما لو كانت نُسخًا instance من `Component` ويستدعي التابع `draw` عليها، ولكن بما أن رست لا تحتوي على توريث، فنحن بحاجة إلى طريقة أخرى لهيكلة مكتبة `gui` للسماح للمستخدمين بتوسيعها بأنواع جديدة.

17.2.1 تعريف سمة لسلوك مشترك

سنعرّف سمةً باسم `Draw` يكون لها تابعٌ واحد يسمى `draw`، لتنفيذ السلوك الذي نريد من الوحدة المصرّفة `gui` أن تملكه. بعد ذلك، يمكننا تعريف شعاع يأخذ كائن سمة trait object، الذي يشير إلى نسخة من نوع ينفّذ السمة المحددة لدينا، إضافةً إلى جدول يُستخدم للبحث عن توابع السمات في هذا النوع في وقت التنفيذ. ننشئ كائن سمة عن طريق استخدام نوع من المؤشرات مثل المرجع `&`، أو المؤشر الذكي `Box<T>`، متبوعًا بالكلمة المفتاحية `dyn`، ثم تحديد السمة ذات الصلة. سنتحدث عن السبب الذي يجعل من المحتمل لكائنات السمة أن تستخدم مؤشرًا لاحقًا في الفصل 19. يمكننا استخدام كائنات السمات بدلًا من النوع المعمم أو الحقيقي. يُضمّن نظام النوع في رست وقت التصريف أينما نستخدم كائن سمة، وإن أي قيمة مُستخدمة في هذا السياق ستنفّذ سمة كائن السمة، وبالتالي لا نحتاج إلى معرفة جميع الأنواع الممكنة وقت التصريف.

لقد ذكرنا أننا نمتنع في رست عن تسمية الهياكل structs والتعدادات enums بالكائنات لتمييزها عن كائنات اللغات البرمجية الأخرى؛ إذ تُفصل البيانات الموجودة في البنية أو التعداد في حقول الهيكل والسلوك في كتل `impl`؛ بينما تُسمّى البيانات والسلوك معًا في اللغات الأخرى غالبًا مثل كائن. مع ذلك، تشبه كائنات السمة إلى حد كبير الكائنات في اللغات الأخرى بمعنى أنها تجمع بين البيانات والسلوك، لكن تختلف كائنات السمة عن الكائنات التقليدية في أنه لا يمكننا إضافة بيانات إلى كائن سمة. لا تعدّ كائنات السمة مفيدةً عمومًا مثل الكائنات في اللغات الأخرى، إذ أن الغرض المحدد منها هو السماح بالتجريد عبر سلوكها المشترك.

توضّح الشيفرة 3 كيفية تعريف سمة تسمى`Draw` مع تابع واحد يسمى `draw`.

اسم الملف: src/lib.rs

```
pub trait Draw {
    fn draw(&self);
}
```

[الشيفرة 3: تعريف السمة `Draw`]

يجب أن تبدو الشيفرة السابقة مألوفةً من حديثنا عن كيفية تعريف السمات سابقًا في الفصل 10. إلا أن هناك بعض الأشياء الجديدة: إذ تعرِّف الشيفرة 4 هيكلًا يدعى `Screen` يحمل شعاعًا باسم `components`. هذا الشعاع من النوع `Box<dyn Draw>`، وهو كائن سمة، ويُعدّ بديلًا لأي نوع داخل `Box` ينفّذ السمة `Draw`.

اسم الملف: src/lib.rs

```
pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}
```

[الشيفرة 4: تعريف هيكل `Screen` مع حقل `components` الذي يحمل شعاعًا من كائنات سمة تطبّق السمة `Draw`]

نعرّف على هيكل `Screen` تابعًا يدعى `run` يستدعي التابع `draw` على كل من `components` الخاصة به كما هو موضح في الشيفرة 5.

اسم الملف: src/lib.rs

```
impl Screen {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
```

```
    }
}
```

[الشيفرة 5: تابع run على Screen الذي يستدعي تابع draw على كل مكون]

يعمل هذا بصورةٍ مختلفة عن تعريف هيكل يستخدم معامل نـوع معمّم generic type مـع حـدود السـمة trait bounds، إذ لا يمكن استبدال معامل النوع المعمم إلا بنوع واحد صريح في كل مرة، بينما تسمح كائنـات السمة بأنواع حقيقية متعددة لتحل مكان كـائن السـمة وقت التنفيـذ. على سـبيل المثـال، كـان من الممكن أن نعرّف هيكل Screen باستخدام نوع معمم وحدود سمة كما في الشيفرة 6.

اسم الملف: src/lib.rs

```
pub struct Screen<T: Draw> {
    pub components: Vec<T>,
}

impl<T> Screen<T>
where
    T: Draw,
{
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

[الشيفرة 6: تنفيذ بديل لهيكل Screen وتابعه run باستخدام أنواع معممة وحدود السمة]

يقيّدنا هذا بنسخة Screen التي تحتوي على قائمة من المكونات جميعها من النوع Button، أو من النـوع TextField؛ فإذا كان لديك مجموعات متجانسة homogeneous collections فقط، يُفضّل استعمال أنواع معممـة وحـدود السـمات لأن التعريفـات سـتكون أحاديـة الشـكل monomorphized في وقت التصـريف لاستخدام الأنواع الفعلية.

من جهة أخرى، يمكن لنسخة Screen واحـدة أن تحمـل النـوع Vec<T> باسـتخدام التـابع الـذي يسـتخدم كائنات السـمة، الـذي يحتـوي بـدوره على Box<Button>، إضـافةً إلى Box<TextField>. لنلقي نظـرةً على كيفية عمل ذلك، ثم سنتحدث عن الآثار المترتبة على وقت التنفيذ.

17.2.2 تنفيذ السمة

سنضيف الآن بعض الأنواع التي تنفّذ السمة `Draw`، وسنأخذ النـوع `Button` مثـالًا على هـذه الأنـواع. يُعـد تنفيذ مكتبة GUI كما ذكرنا سابقًا خارج موضوعنا هنا، لذا لن يكون لتابع `draw` أي تنفيـذ فعلي داخلـه. لنتخيـل الشـكل الـذي قـد يبـدو عليـه التنفيـذ، فقـد يحتـوي هيكـل `Button` على حقـول لكـل من `width` و `height` و `label` كما هو موضح في الشيفرة 7.

اسم الملف: src/lib.rs

```
pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // الشيفرة البرمجية المسؤولة عن رسم الزر
    }
}
```

[الشيفرة 7: هيكل `Button` الذي يطبّق السمة `Draw`]

ستختلف حقول `width` و `height` و `label` الموجودة في `Button` عن الحقول الموجودة في المكونـات الأخرى، فعلى سبيل المثال قد يحتـوي النـوع `TextField` نفس الحقـول، إضـافةً إلى حقـل `placeholder`. ستنفّذ كل الأنواع التي نريد رسمها على الشاشة سمة `Draw`، لكن ستستعمل شيفرةً برمجيةً مختلفـة في التـابع `draw` لتعريف كيفية رسم ذلك النوع تحديدًا، كما في `Button` هنا (بدون شيفرة برمجية لمكتبة GUI فعلية كما ذكرنا سابقًا). قد يحتوي النوع `Button` على سبيل المثـال كتلـة `impl` إضـافية تحتـوي على توابـع مرتبطـة بمـا يحدث عندما يضغط مستخدم الزر، ولا تنطبق هذه الأنواع من التوابع على أنواع مثل `TextField`.

إذا قرر شخص ما استعمال مكتبتنا لتطبيق هيكل `SelectBox` الـذي يحتـوي الحقـول`width` و `height` و `options`، فسينفّذ سمة `Draw` على النوع `SelectBox` أيضًا كما هو موضح بالشيفرة 8.

اسم الملف: src/main.rs

```
use gui::Draw;

struct SelectBox {
```

```
    width: u32,
    height: u32,
    options: Vec<String>,
}

impl Draw for SelectBox {
    fn draw(&self) {
        // الشيفرة البرمجية المسؤولة عن رسم صندوق الاختيار
    }
}
```

[الشيفرة 8: وحدة مصرفة أخرى تستعمل `gui` وتنفذ سمة `Draw` على هيكل `SelectBox`]

يمكن لمستخدم مكتبتنا الآن كتابة الدالة `main` ليُنشئ نسخةً من `Screen`، ثم إضافة كل من `SelectBox` و `Button` لنسخة `Screen` بوضع كل واحدة منها في `Box<T>` لتصبح سمة كائن، ويمكنه بعد ذلك استدعاء التابع `run` على نسخة`Screen` التي ستستدعي `draw` على كل من المكونات، وتوضح الشيفرة 9 التطبيق المذكور.

اسم الملف: src/main.rs

```
use gui::{Button, Screen};

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    String::from("Yes"),
                    String::from("Maybe"),
                    String::from("No"),
                ],
            }),
            Box::new(Button {
                width: 50,
                height: 10,
```

```
                label: String::from("OK"),
            }),
        ],
    };

    screen.run();
}
```

[الشيفرة 9: استخدام كائنات السمة لتخزين قيم لأنواع مختلفة تنفذ السمة ذاتها]

لم نفترض عند كتابتنا للمكتبة بأن شخصًا ما قد يضيف النوع `SelectBox`، إلا أن تطبيق `Screen` لدينا قادرٌ على العمل مع النوع الجديد ورسمه، وذلك لأن `SelectBox` ينفّذ سمة `Draw`، ما يعني أنه ينفّذ تابع `draw`.

هذا المفهوم - المتمثل بالاهتمام فقط بالرسائل التي تستجيب لها القيمة بدلًا من النوع الحقيقي للقيمة - مشابهٌ لمفهوم كتابة البطة duck typing في اللغات البرمجية المكتوبة ديناميكيًا؛ بمعنى أنه إذا كان شيء ما يسير مثل البطة ويصدر صوتًا مثل البطة، فيجب أن يكون بطة لا محالة. لا يحتاج `run` إلى معرفة النوع الحقيقي لكل مكون عند تنفيذ `run` على `Screen` في الشيفرة 5، فهو لا يتحقق ما إذا كان المكوِّن نسخةً من النوع `Button` أو `SelectBox` بل يستدعي فقط التابع `draw` على المكوّن. عرّفنا `Screen` لتحتاج إلى قيم يمكننا استدعاء تابع `draw` عليها من خلال تحديد `Box<dyn Draw>` على أنه نوع القيم في الشعاع `components`.

ميزة استخدام كائنات السمة ونظام نوع رست لكتابة شيفرة برمجية مشابهة للشيفرة البرمجية التي تستعمل كتابة البطة هي أننا لا نضطر أبدًا إلى التحقّق ما إذا كانت القيمة تنفذ تابعًا معينًا وقت التنفيذ، أو القلق بشأن حدوث أخطاء إذا كانت القيمة لا تنفّذ التابع، لكننا نستدعيه بغضّ النظر عن ذلك. لن تصرّف رست الشيفرة البرمجية الخاصة بنا إذا كانت القيم لا تنفّذ السمات التي تحتاجها كائنات السمة.

تُظهر الشيفرة 10 على سبيل المثال ما يحدث إذا حاولنا إنشاء `Screen` مع `String` مثل مكوِّن:

اسم الملف: src/main.rs

```
use gui::Screen;

fn main() {
    let screen = Screen {
        components: vec![Box::new(String::from("Hi"))],
    };
```

```
    screen.run();
}
```

[الشيفرة 10: محاولة استخدام نوع لا ينفّذ سمة كائن السمة]

سنحصل على الخطأ التالي، وذلك لأن `String` لا ينفّذ السمة `Draw`:

```
$ cargo run
   Compiling gui v0.1.0 (file:///projects/gui)
error[E0277]: the trait bound `String: Draw` is not satisfied
 --> src/main.rs:5:26
  |
| 5 |         components: vec![Box::new(String::from("Hi"))],
  |                          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait
`Draw` is not implemented for `String`
  |
  = help: the trait `Draw` is implemented for `Button`
  = note: required for the cast from `String` to the object type `dyn
Draw`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `gui` due to previous error
```

يتيح لنا هذا الخطأ معرفة ما إذا كنّا نمرّر شيئًا ما إلى `Screen` لم نقصد تمريره، ولذا يجب أن نمرّر نوعًا مختلفًا أو يجب أن نطبّق `Draw` على `String` حتى تتمكن `Screen` من استدعاء `draw` عليه.

17.2.3 الإرسال الديناميكي لكائنات السمة

باستذكار حديثنا عن عملية توحيد الشكل monomorphization المنفذة بواسطة المصرف عندما نستخدم حدود السمة على الأنواع المعممّة وذلك في قسم (أداء الشيفرة باستعمال الأنواع المعممة) في الفصل 10: يولّد المصرّف تطبيقات غير معممّة للدوال والتوابع لكل نوع حقيقي نستخدمه بدلًا من معامل نوع مُحدّد. تنجز الشيفرة البرمجية الناتجة من عملية توحيد الشكل إيفادًا ساكنًا static dispatch، والذي يحدث عندما يعرِف المصرّف التابع الذي تستدعيه وقت التصريف. يتعارض هذا مع الإيفاد الديناميكي الذي يحدث عندما يتعذر على المصرف أن يخبرك بالتابع الذي تستدعيه وقت التصريف. يرسل المصرّف في حالات الإيفاد الديناميكي شيفرة برمجية تُحدّد في وقت التنفيذ التابع الذي يجب استدعاؤه.

يجب أن تستخدم رست الإيفاد الديناميكي عندما نستخدم كائنات السمة، إذ لا يعرف المصرّف جميع الأنواع الممكن استعمالها مع الشيفرة البرمجية التي تستخدم كائنات السمة، لذا فهو لا يعرف التابع الذي يُنفّذ على النوع المطلوب استدعاؤه. وتستخدم رست بدلًا من ذلك المؤشرات داخل كائن السمة في وقت التنفيذ لمعرفة التابع الذي يجب استدعاؤه، ويتسبب هذا البحث بزيادة في وقت التنفيذ مقارنةً بالإيفاد الثابت. يمنع الإيفاد الديناميكي أيضًا المصرف من اختيار تضمين شيفرة التابع البرمجية التي تمنع بدورها بعض التحسينات، ومع ذلك فقد حصلنا على مزيد من المرونة في الشيفرة البرمجية التي كتبناها في الشيفرة 5 وتمكنا من دعمها في الشيفرة 9، لذا فهي مقايضة يجب أخذها بالحسبان.

17.3 تنفيذ نمط تصميمي Design Pattern كائني التوجه

نمط الحالة state pattern هو نمط تصميم كائني التوجه Object-Oriented، والمغزى منه هو أننا نعرّف مجموعةً من الحالات التي يمكن للقيمة أن تمتلكها داخليًا، وتُمثَّل الحالات من خلال مجموعة من كائنات الحالة state objects، ويتغير سلوك القيمة بناءً على حالتها. سنعمل من خلال مثال لهيكل منشور مدونة يحتوي على حقل للاحتفاظ بحالته، والتي ستكون كائن حالة من مجموعة القيم "مسودة draft" أو " قيد المراجعة review" أو "منشور published".

تتشارك كائنات الحالة بالوظيفة، ونستخدم الهياكل structs والسمات traits بدلًا من الكائنات objects والوراثة inheritance في لغة رست. كل كائن حالة مسؤول عن سلوكه الخاص وعن تحديد متى يجب عليه أن يتغير من حالة إلى أخرى. لا تعرف القيمة التي تخزّن كائن الحالة شيئًا عن السلوك المختلف للحالات أو متى تنتقل بينها.

ميزة استخدام نمط الحالة هي أنه لن نحتاج إلى تغيير شيفرة القيمة البرمجية التي تحتفظ بالحالة أو الشيفرة البرمجية التي تستخدم القيمة عندما تتغير متطلبات العمل للبرنامج، إذ سنحتاج فقط إلى تعديل الشيفرة البرمجية داخل أحد كائنات الحالة لتغيير قواعدها أو ربما إضافة المزيد من كائنات الحالة.

سننّفذ بدايةً نمط الحالة بطريقة تقليديّة كائنية التوجه، ثم سنستعمل نهجًا أكثر شيوعًا في رست. لننفّذ تدريجيًا طريقةً لتنظيم سير عمل منشور المدونة باستعمال نمط الحالة.

ستكون النتيجة النهائية كما يلي:

1. يبدأ منشور المدونة مثل مسودة فارغة.
2. يُطلب مراجعة المنشور عند الانتهاء من المسودة.
3. يُنشر المنشور عندما يُوافَق عليه.
4. منشورات المدونة التي هي بحالة "منشور published" هي المنشورات الوحيدة التي تعيد محتوًى ليُطبع، بحيث لا يمكن نشر المنشورات غير الموافق عليها عن طريق الخطأ.

يجب ألا يكون لأي تعديلات أخرى أُجريت على إحدى المنشورات أي تأثير، فعلى سبيل المثال إذا حاولنا الموافقة على مسودة منشور مدونة قبل أن نطلب المراجعة، فيجب أن يبقى المنشور مسودةً غير منشورة.

تظهر الشيفرة 11 سير العمل هذا على هيئة شيفرة برمجية، وهذا مثال عن استعمال الواجهة البرمجية API التي سننفّذها في وحدة مكتبة مصرّفة library crate تسمى `blog`. لن تُصرَّف الشيفرة البرمجية التالية بنجاح، لأننا لم ننفّذ الوحدة المصرفة `blog` بعد.

اسم الملف: src/main.rs

```
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());

    post.request_review();
    assert_eq!("", post.content());

    post.approve();
    assert_eq!("I ate a salad for lunch today", post.content());
}
```

[الشيفرة 11: الشيفرة التي تُظهر السلوك المرغوب الذي نريد لوحدتنا المصرفة `blog` أن تحتويه]

نريد السماح للمستخدم بإنشاء مسودة منشور مدونة جديد باستعمال `Post::new`، كما نريد السماح بإضافة نص إلى منشور المدونة، إذ يجب ألّا نحصل على أي نص إذا حاولنا الحصول على محتوى المنشور فورًا قبل الموافقة وذلك لأن المنشور لا يزال في وضع المسودة. أضفنا `assert_eq!` في الشيفرة البرمجية لأغراض توضيحية. قد يكون اختبار الوحدة unit test المناسب لهذا هو التأكيد على أن مسودة منشور مدونة تعرض سلسلةً نصيةً string فارغة من تابع `content` لكننا لن نكتب أي اختبارات لهذا المثال حاليًا.

نريد بعد ذلك تمكين طلب مراجعة المنشور ونريد أن يُعيد `content` سلسلةً نصيةً فارغة أثناء انتظار المراجعة، ويُنشر المنشور عندما يتلقى الموافقة، مما يعني أنه سيُعاد نص المنشور عندما نستدعي `content`.

لاحظ أن النوع الوحيد الذي نتفاعل معه من الوحدة المصرّفة هو النوع `Post`، إذ سيستعمل هذا النوع نمط الحالة وسيحتوي على قيمة واحدة من ثلاث قيم هي كائنات حالة تمثل الحالات المختلفة التي يمكن أن يكون المنشور فيها، ألا وهي مسودة، أو انتظار المراجعة، أو النشر. يجري التحكم بالتغيير من حالة إلى أخرى داخليًا

ضمن نوع Post، إذ تتغير الحالات استجابةً للتوابع التي يستدعيها مستخدمو مكتبتنا في نسخة Post، لكن لا يتعين عليهم إدارة تغييرات الحالة مباشرةً، كما لا يمكن للمستخدمين ارتكاب خطأ في الحالات مثل نشر منشور قبل مراجعته.

17.3.1 تعريف المنشور وإنشاء نسخة جديدة في حالة المسودة

لنبدأ بتنفيذ المكتبة؛ نعلم أننا بحاجة إلى هيكل عام public struct يدعى Post ليخزّن محتوى المنشور، لذا سنبدأ بتعريف الهيكل ودالة عامة مرتبطة associated تدعى new لإنشاء نسخة من Post كما هو موضح في الشيفرة 12. سننشئ أيضًا سمةً خاصة تدعى State تعرّف السلوك الذي يجب أن تتمتع به جميع كائنات حالة Post.

سيخزّن هيكل Post بعد ذلك كائن السمة Box<dyn State> داخل Option<T> في حقل خاص يسمى state ليخزّن كائن الحالة، وسترى سبب ضرورة Option<T> بعد قليل.

اسم الملف: src/lib.rs

```
pub struct Post {
    state: Option<Box<dyn State>>,
    content: String,
}

impl Post {
    pub fn new() -> Post {
        Post {
            state: Some(Box::new(Draft {})),
            content: String::new(),
        }
    }
}

trait State {}

struct Draft {}

impl State for Draft {}
```

[الشيفرة 12: تعريف هيكل Post ودالة new التي تنشئ نسخة من Post وسمة State وهيكل Draft جدد]

تعـرّف السـمة `State` السـلوك المشـترك بين حـالات النشـر المختلفـة. كائنـات الحالـة هي `Draft` و `PendingReview` و `Published` وسوف تنفِّذ جميعها سمة `State`. لا تحتـوي السـمة في الـوقت الحـالي على أي توابع وسنبدأ بتعريف حالة `Draft` فقط لأن هذه هي الحالة التي نريد أن يبدأ المنشور فيها.

عندما ننشئ `Post` جديد نعيّن حقل `state` الخاص به بقيمـة `Some` الـتي تحتـوي على `Box`، وتشـير `Box` هنا إلى نسخة جديدة للهيكل `Draft`، أي أنه عندما ننشئ نسـخةً جديـدة من `Post` فإنهـا سـتبدأ مثـل مسـودة. نظرًا لأن حقل `state` للهيكل `Post`هو خاص ولا توجد طريقةٌ لإنشـاء `Post` في أي حالـة أخـرى. عيّننـا سلسـلة نصية `String` فارغة جديدة للحقل `content` في الدالة `Post::new`.

17.3.2 تخزين نص محتوى المنشور

كنا في الشيفرة 11 قادرين على استدعاء تابع يسمى `add_text` وتمرير`&str` إليه ليُضـاف لاحقًـا على أنـه محتوى نص منشور المدونة. ننفّذ هذا مثل تابع بدلًا من عرض حقل `content` على أنه `pub` حتى نتمكن لاحقًـا من تنفيذ تابع يتحكم بكيفية قراءة بيانات حقل `content`. تابع `add_text` واضح جـدًا، لـذا سنضـيف التنفيـذ في الشيفرة 13 إلى كتلة `impl Post`.

اسم الملف: src/lib.rs

```
impl Post {
    // --snip--
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

[الشيفرة 13: تنفيذ تابع `add_text` لإضافة نص لمنشور `content`]

يأخـذ تـابع `add_text` مرجعًـا متغيّـرًا mutable إلى `self` لأننـا نعـدّل نسـخة `Post` الـتي نسـتدعي `add_text` عليها، ثم نستدعي `push_str` على `String` في `content` ونمـرّر الوسـيط `text` لإضـافتها إلى `content` المحفوظ. لا يعتمد هذا السلوك على حالة المنشور لذا فهو ليس جزءًا من نمط الحالة. لا يتفاعل تـابع `add_text` مع حقل `state` إطلاقًا، لكنه جزءٌ من السلوك الذي نريد دعمه.

17.3.3 ضمان أن محتوى مسودة المنشور فارغ

ما زلنا بحاجة تابع `content` حتى بعد استدعائنا `add_text` وإضـافة بعض المحتـوى إلى منشـورنا وذلـك لإعادة شريحة سلسلة نصية slice string فارغة لأن المنشـور لا يـزال في حالـة المسـودة كمـا هـو موضـح في السطر 7 من الشيفرة 11. لننفّذ حاليًا تابع `content` بأبسط شيء يستوفي هـذا المتطلب؛ وهـو إعـادة شـريحة سلسلة نصية فارغة دائمًا، إلا أننا سنغير هذا لاحقًا بمجـرد تقـديم القـدرة على تغيـير حالـة المنشـور حـتى يمكن

نشره. حتى الآن يمكن أن تكون المنشورات في حالة المسودة فقط لذلك يجب أن يكون محتوى المنشـور فارغًـا دائمًا. تُظهر الشيفرة 14 تنفيذ الموضع المؤقت هذا.

اسم الملف: src/lib.rs

```
impl Post {
    // --snip--
    pub fn content(&self) -> &str {
        ""
    }
}
```

[الشيفرة 14: إضافة تنفيذ موضع مؤقت للتابع `content` على `Post` يُعيد دائمًا شريحة سلسلة فارغة]

يعمل الآن كل شيء كما خُطّط له مع إضافة تابع `content` في الشيفرة 11 حتى السطر 7.

17.3.4 طلب مراجعة للمنشور يغير حالته

نحتاج بعد ذلك إلى إضافة إمكانية طلب مراجعة منشور وتغيير حالتــه من `Draft` إلى `PendingReview`، وتوضّح الشيفرة 15 هذا الأمر.

اسم الملف: src/lib.rs

```
impl Post {
    // --snip--
    pub fn request_review(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.request_review())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<dyn State>;
}

struct Draft {}

impl State for Draft {
```

```
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        Box::new(PendingReview {})
    }
}

struct PendingReview {}

impl State for PendingReview {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        self
    }
}
```

[الشيفرة 15: تنفيذ توابع `request_review` على `Post` وسمة `State`]

نعطي `Post` تابعًا عام يدعى `request_review`، والذي سيأخذ مرجعًا متغيّرًا يشير إلى `self`، نستدعي بعد ذلك التابع `request_review` الداخلي على الحالة `Post` الحالية، إذ يستخدم `request_review` الثاني الحالة الحالية ويعيد حالةً جديدة.

نضيف التابع `request_review` إلى السمة `State`؛ إذ ستحتاج جميع الأنواع التي تطبق السمة الآن إلى تنفيذ تابع `request_review`. لاحظ أن لدينا `self: Box<Self>` بدلًا من `self`، أو `&self`، أو `mut &self` بمثابة معامل أول للتابع. تعني هذه الصيغة أن التابع صالحٌ فقط عندما يُستدعى على `Box` يحتوي النوع. تأخذ هذه الطريقة بالصياغة ملكية `Box<Self>` مما يبطل الحالة القديمة بحيث يمكن أن تتحول قيمة حالة `Post` إلى حالة جديدة.

يجب أن يأخذ تابع `request_review` ملكية قيمة الحالة القديمة لاستخدامها، وهذه هي فائدة استخدام `Option` في حقل `state` الخاص بالمنشور `Post`؛ إذ نستدعي تابع `take` لأخذ قيمة `Some` من حقل `state` وترك `None` في مكانها لأن رست لا تسمح لنا بامتلاك حقول غير مأهولة في الهياكل. يتيح لنا ذلك نقل قيمة `state` خارج `Post` بدلًا من استعارتها. ثم نعيّن قيمة `state` للمنشور بنتيجة هذه العملية.

نحتاج إلى جعل `state` مساوية للقيمة `None` مؤقتًا بدلًا من تعيينها مباشرةً باستعمال شيفرة برمجية مثل:

```
    self.state = self.state.request_review();
```

للحصول على ملكية قيمة `state`، ويضمن لنا ذلك أن `Post` لا يمكنه استخدام قيمة `state` القديمة بعد أن حوّلناها إلى حالة جديدة.

يُعيد التابع `request_review` الموجود في `Draft` نسخةً جديدةً موضوعة في صندوق لهيكل `PendingReview` جديد يمثل الحالة التي يكون فيها المنشور في انتظار المراجعة. يطبّق هيكل `PendingReview` أيضًا تابع `request_review` ولكنه لا يجري أي تحويلات، ويُعيد بدلًا من ذلك قيمًا لنفسه لأنه يجب أن يبقى المنشور على حالته إذا كانت `PendingReview` بعد طلبنا لمراجعته.

يمكننا الآن البدء في رؤية مزايا نمط الحالة، فتابع `request_review` على `Post` هو نفسه بغض النظر عن قيمة `state`، فكل حالة مسؤولة عن قواعدها الخاصة.

سنترك تابع `content` على `Post` كما هو ونعيد شريحة سلسلة نصية string slice فارغة. يمكننا الآن الحصول على `Post` في حالة `PendingReview` وكذلك في حالة `Draft` إلا أننا نريد السلوك ذاته في حالة `PendingReview`. تعمل الشيفرة 11 الآن بنجاح وصولًا للسطر 10.

17.3.5 إضافة approve لتغيير سلوك التابع content

سيكون تابع `approve` مشابهًا لتابع `request_review`، إذ سيضبط قيمة `state` على القيمة التي تقول الحالة الحالية أنها يجب أن تكون عليها عند الموافقة على تلك الحالة كما هو موضح في الشيفرة 16.

اسم الملف: src/lib.rs

```
impl Post {
    // --snip--
    pub fn approve(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.approve())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<dyn State>;
    fn approve(self: Box<Self>) -> Box<dyn State>;
}

struct Draft {}

impl State for Draft {
    // --snip--
    fn approve(self: Box<Self>) -> Box<dyn State> {
```

```
        self
    }
}

struct PendingReview {}

impl State for PendingReview {
    // --snip--
    fn approve(self: Box<Self>) -> Box<dyn State> {
        Box::new(Published {})
    }
}

struct Published {}

impl State for Published {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        self
    }

    fn approve(self: Box<Self>) -> Box<dyn State> {
        self
    }
}
```

[الشيفرة 16: تنفيذ تابع `approve` على `Post` وسمة `State`]

نضيف تابع `approve` إلى سمة `State` ونضيف هيكلًا جديدًا ينفّذ السمة `State`، والحالة `Published`.

لن يكـون للتـابع `approve` على `Draft` عنـد اسـتدعائه أي تـأثير على غـرار الطريقـة الـتي يعمـل بهـا `request_review` في `PendingReview`، وذلـك لأن التـابع `approve` سـيعيد `self`، بينمـا يعيـد التـابع `approve` عندما نستدعيه على `PendingReview` نسخةً جديدةً ضمن صندوق boxed لهيكـل `Published`. ينفّـذ هيكـل `Published`السـمة `State` وتعيـد نفسـها من أجـل كـل من تـابع `request_review` وتـابع `approve` لأن المنشور يجب أن يبقى في حالة `Published` في تلك الحالات.

نحتاج الآن إلى تحديث تـابع `content` على `Post`، إذ نريـد أن تعتمـد القيمـة المُعـادة من `content` على حالة `Post` الحالية، لذلك نعرّف `Post` مفوض للتابع `content` المعرّف على قيمـة الحقـل `state` الخـاص بـه كما هو موضح في الشيفرة 17.

اسم الملف: src/lib.rs

```
impl Post {
    // --snip--
    pub fn content(&self) -> &str {
        self.state.as_ref().unwrap().content(self)
    }
    // --snip--
}
```

[الشيفرة 17: تحديث تابع `content` على `Post` للتفويض لتابع `content` على `State`]

نظرًا لأن الهدف هو الاحتفاظ بكل هذه القواعد داخل الهياكل التي تنفّذ السمة `State` فإننـا نسـتدعي تـابع `content` على قيمة `state` ونمرر نسخة المنشور (في هذه الحالة `self`) مثل وسـيط، ثم نعيـد القيمـة الـتي أُعيدَت من استعمال تابع `content` إلى قيمة `state`.

نستدعي تابع `as_ref` على `Option` لأننا نريد مرجعًا للقيمة داخل `Option` بدلًا من الحصـول على ملكيـة القيمة. بما أن `state` هو `Option<Box<dyn State>>`، تكون القيمة المُعادة عنـد اسـتدعاء `as_ref` هي `Option<&Box<dyn State>>`. نحصل على خطأ إذا لم نستدعي `as_ref` لأننا لا نستطيع نقل `state` من `&self` المستعارة إلى معامل الدالة.

نستدعي بعد ذلك التابع `unwrap` الـذي نعلم أنـه لن يهلـع أبـدًا لأننـا نعلم أن التوابـع الموجـودة على `Post` تضمن أن `state` سيحتوي دائمًا على القيمة `Some` عند الانتهاء من هـذه التوابـع، وهـذه إحـدى الحـالات الـتي تحدثنا عنها سابقًا في قسم "الحالات التي تعرف فيها معلومات أكثر من المصرف" من الفصـل 9، وهي عنـدما نعلم أن قيمة `None` غير ممكنة أبدًا على الرغم من أن المصرف غير قادر على فهم ذلك.

سيكون تأثير التحصـيل القسـري deref coercion في هـذه المرحلـة سـاريًا على كـل من `&` و `Box` عنـدما نستدعي `content` على `&Box<dyn State>`، لذلك يُسـتدعى تـابع `content` في النهايـة على النـوع الـذي ينفذ سمة `State`. هذا يعني أننا بحاجة إلى إضـافة `content` إلى تعريـف سـمة `State` وهنـا سنضـع منطـق المحتوى الذي سيُعاد اعتمادًا على الحالة التي لدينا كما هو موضح في الشيفرة 18.

اسم الملف: src/lib.rs

```
trait State {
```

```
        // --snip--
        fn content<'a>(&self, post: &'a Post) -> &'a str {
            ""
        }
    }

    // --snip--
    struct Published {}

    impl State for Published {
        // --snip--
        fn content<'a>(&self, post: &'a Post) -> &'a str {
            &post.content
        }
    }
```

[الشيفرة 18: إضافة تابع `content` إلى سمة `State`]

نضيف تنفيذًا مبدئيًا للتابع `content` الذي يُعيد شريحة سلسلة نصية فارغة، ويعني هذا أننا لسنا بحاجة إلى تنفيذ `content` في هيكلي `Draft` و `PendingReview`، إذ سيُعيد هيكل `Published` تعريف التابع `content` ويعيد القيمة في `post.content`.

لاحظ أننا نحتاج إلى توصيف لدورة الحياة lifetime على هذا التابع كما ناقشنا سابقًا في الفصل 10، إذ نأخذ هنا مرجعًا إلى `post` مثل وسيط ونعيد مرجعًا إلى جزء من `post` وبالتالي ترتبط دورة حياة المرجع المُعاد بدورة حياة وسيط `post`.

تعمل الشيفرة 11 الآن كاملةً بعد أن طبّقنا نمط الحالة state pattern مع قواعد سير عمل منشور المدونة. المنطق المتعلق بالقواعد موجود في كائنات الحالة بدلًا من بعثرته في جميع أنحاء `Post`.

لماذا لم نستخدم تعدادًا؟

لربما كنت تتساءل عن سبب عدم استخدامنا `enum` مع حالات المنشورات المختلفة الممكنة مثل متغايرات variants؛ هذا بالتأكيد حل ممكن، جرّبه وقارن النتائج النهائية لترى أيهما تفضل. أحد عيوب استعمال التعداد هو أن كل مكان يتحقق من قيمة التعداد سيحتاج إلى تعبير `match` أو ما شابه للتعامل مع كل متغاير ممكن، ويمكن أن يتطلب هذا الحل شيفرةً برمجيةً مكررة أكثر مقارنةً مع حل كائن السمة هذا.

17.3.6 سلبيات استخدام نمط الحالة

وضّحنا أن رست قادرة على تنفيذ نمط الحالة كائنية التوجه لتغليف أنواع مختلفة من السلوك التي يجب أن يتمتع بها المنشور في كل حالة. لا تعرف التوابع في `Post` شيئًا عن السلوكيات المختلفة. تسمح لنا الطريقة التي نظّمنا بها الشيفرة البرمجية -تنفيذ سمة `State` على الهيكل `Published`- أن ننظر في مكان واحد فقط لمعرفة الطرق المختلفة التي يمكن أن يتصرف بها المنشور المقبول للنشر.

إذا أردنا إنشاء تنفيذ بديل لا يستخدم نمط الحالة فقد نستخدم بدلًا من ذلك تعبيرات `match` في التوابع على `Post` أو حتى في الشيفرة `main` التي تتحقق من حالة المنشور وتغيِّر السلوك في تلك الأماكن، هذا يعني أنه يتعين علينا البحث في عدة أماكن لفهم جميع الآثار المترتبة على المنشور في الحالة المنشورة، وسيؤدي هذا إلى زيادة عدد الحالات التي أضفناها، إذ سيحتاج كل تعبير من تعبيرات `match` هذه إلى ذراع أخرى.

لا تحتاج توابع `Post` أو الأماكن التي نستخدم فيها `Post` إلى تعبيرات `match` مع نمط الحالة، وسنحتاج من أجل إضافة حالة جديدة إلى إضافة هيكل جديد فقط وتنفيذ توابع السمة على هذا الهيكل الواحد.

من السهل توسيع التنفيذ باستعمال نمط الحالة لإضافة المزيد من الوظائف. لمعرفة بساطة الحفاظ على الشيفرة البرمجية التي تستخدم نمط الحالة، جرّب بعضًا من هذه الاقتراحات:

- أضف تابع `reject` الذي يغيّر شكل حالة المنشور من `PendingReview` إلى `Draft`.
- استدعِ `approve` مرّتين قبل أن تتغير الحالة إلى `Published`.
- اسمح للمستخدمين بإضافة محتوى النص فقط عندما يكون المنشور في حالة `Draft`. تلميح: اجعل كائن الحالة مسؤولًا عما قد يتغير بشأن المحتوى ولكن ليس مسؤولًا عن تعديل `Post`.

يتمثل أحد الجوانب السلبية لنمط الحالة في أنه نظرًا لأن الحالات تنفّذ التحول بين الحالات، تكون بعض الحالات مقترنة ببعضها، وإذا أضفنا حالةً أخرى بين `PendingReview` و `Published` مثل `Scheduled`، سيتعين علينا تغيير الشيفرة البرمجية في `PendingReview` للانتقال إلى `Scheduled` بدلًا من ذلك. سيقلّ العمل المطلوب إذا لم تكن `PendingReview` بحاجة إلى التغيير مع إضافة حالة جديدة ولكن هذا يعني التبديل إلى نمط تصميم آخر.

السلبية الأخرى هو أننا كرّرنا بعض المنطق، ويمكن إزالة بعض حالات التكرار من خلال إجراء عمليات تنفيذ مبدئية لتوابع `request_review` و `approve` على سمة `State` التي تعيد `self`، ومع ذلك فإن هذا من شأنه أن ينتهك سلامة الكائن لأن السمة لا تعرف فعلًا ما ستكون عليه `self` الحقيقية. نريد أن نكون قادرين على استعمال `State` مثل كائن سمة لذلك نحتاج إلى أن تكون توابعها آمنة من الكائنات.

تتضمن التكرارات الأخرى عمليات تنفيذ مماثلة لتوابع `request_review` و `approve` على `Post`، يفوّض كلا التابعين تنفيذ التابع ذاته على القيمة في حقل `state` للقيمة `Option` وتعيين القيمة الجديدة

لحقل `state` إلى النتيجة. إذا كان لدينا الكثير من التوابع في `Post` التي اتبعت هذا النمط، فقد نفكر في تعريف ماكرو لإزالة التكرار، راجع قسم وحدات الماكرو في الفصل 19.

لا نستفيد استفادة كاملة من نقاط قوة رست بقدر الإمكان عبر تنفيذ نمط الحالة تمامًا كما هو معرّف في اللغات البرمجية كائنية التوجه الأخرى. دعنا نلقي نظرةً على بعض التغييرات الممكن إجراؤها على الوحدة المصرفة `blog`، والتي من شأنها أن تجعل الحالات غير الصالحة والانتقالات transitions أخطاءً تظهر وقت التصريف.

17.3.7 ترميز الحالات والسلوك مثل أنواع

سنوضّح كيفية إعادة التفكير بنمط الحالة للحصول على مجموعة مختلفة من المقايضات، وذلك بدلًا من تغليف الحالات والانتقالات بحيث لا يكون لدى الشيفرة البرمجية الخارجية أي معرفة بها. نرمّز الحالات إلى أنواع مختلفة، وبالتالي سيمنع نظام فحص النوع في رست محاولات استخدام مسودات المنشورات، بحيث لا يُسمح إلا بالمنشورات المنشورة وذلك عن طريق إصدار خطأ في المصرّف.

لننظر إلى الجزء الأول من دالة `main` في الشيفرة 11.

اسم الملف: src/main.rs

```
fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());
}
```

ما زلنا نسمح بإنشاء منشورات جديدة في حالة المسودة باستخدام `Post::new` والقدرة على إضافة نص إلى محتوى المنشور، ولكن بدلًا من وجود تابع `content` في مسودة المنشور التي تعيد سلسلةً نصيةً فارغة، سنعمل على تعديلها بحيث لا تحتوي مسودة المنشورات على تابع `content` إطلاقًا؛ وستحصل بهذه الطريقة على خطأ في المصرف إذا حاولنا الحصول على محتوى مسودة منشور يخبرنا أن التابع غير موجود، ونتيجةً لذلك سيكون من المستحيل بالنسبة لنا عرض محتوى مسودة المنشور عن طريق الخطأ في مرحلة الإنتاج production لأن هذه الشيفرة البرمجية لن تصرف. تُظهر الشيفرة 19 تعريف هيكلي `Post` و `DraftPost` إضافةً إلى التوابع الخاصة بكل منهما.

اسم الملف: src/lib.rs

```
pub struct Post {
    content: String,
```

```
}

pub struct DraftPost {
    content: String,
}

impl Post {
    pub fn new() -> DraftPost {
        DraftPost {
            content: String::new(),
        }
    }

    pub fn content(&self) -> &str {
        &self.content
    }
}

impl DraftPost {
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

[الشيفرة 19: `Post` مع تابع `content` و `DraftPost` بدون تابع `content`]

يحتـوي كـل من هيكلي `Post` و `DraftPost` على حقـل `content` خـاص يحتـوي على النص الخـاص بمنشور المدونة. لم يعد للهياكل حقل `state` لأننا ننقل ترميز الحالة إلى أنواع الهياكـل، وسـيمثل هيكـل `Post` منشورًا قد نُشر وله تابع `content` يُعيد `content`.

لا تزال لدينا دالـة `Post::new` ولكن بـدلًا من إعـادة نسـخة من `Post`، سـتُعيد نسـخةً من `DraftPost`، وذلك نظرًا لأن `content` خاص ولا وجود لأي دوال تُعيد `Post`، وبالتالي لا يمكن إنشاء نسخة عن `Post` حاليًا.

يحتوي هيكل `DraftPost` على تابع `add_text` لذا يمكننا إضافة نص إلى `content` كما كـان من قبـل، لكن لاحظ أن `DraftPost` لا يحتوي على تابع `content` معرّف لذا يضمن البرنامج الآن بدء جميع المنشـورات مثل مسودات منشورات وعدم إتاحة محتوى مسودات المنشـورات للعـرض. سـتؤدي أي محاولـة للتحايـل على هذه القيود إلى حدوث خطأ في المصرّف.

17.3.8 تنفيذ الانتقالات مثل تحولات إلى أنواع مختلفة

كيف نحصل على منشور قد نُشر؟ نريد فرض القاعدة التي تنص على وجوب مراجعة مسودة المنشور والموافقة عليها قبل نشرها. ينبغي عدم عرض أي منشور في حالة "قيد المراجعة" أي محتوى. لنطبّق هذه القيود عن طريق إضافة هيكل آخر باسم `PendingReviewPost` وتعريف التابع `request_review` في `DraftPost` لإعادة `PendingReviewPost` وتعريف تابع `approve` على `PendingReviewPost` لإعادة `Post` كما هو موضح في الشيفرة 20.

اسم الملف: src/lib.rs

```
impl DraftPost {
    // --snip--
    pub fn request_review(self) -> PendingReviewPost {
        PendingReviewPost {
            content: self.content,
        }
    }
}

pub struct PendingReviewPost {
    content: String,
}

impl PendingReviewPost {
    pub fn approve(self) -> Post {
        Post {
            content: self.content,
        }
    }
}
```

[الشيفرة 20: هيكل `PendingReviewPost` مُنشأ عن طريق استدعاء `request_review` على `DraftPost` وتابع `approve` الذي يرجع `PendingReviewPost` إلى `Post` منشور]

يأخذ التابعان `request_review` و `approve` ملكية `self` وبالتالي تستخدم نُسَخ `DraftPost` و `PendingReviewPost` وتحوّلهما إلى `PendingReviewPost` و `Post` منشور published على التوالي،

وبهذه الطريقة لن يكون لدينا أي نسخ متبقية من `DraftPost` بعد أن استدعينا `request_review` عليها وما إلى ذلك.

لا يحتوي هيكل `PendingReviewPost` على تابع `content` معرف عليه لذلك تؤدي محاولة قراءة محتواها إلى حدوث خطأ في المصرّف كما هو الحال مع `DraftPost`، لأن الطريقة الوحيدة للحصول على نسخة `Post` قد جرى نشره وله تابع `content` معرّف هي استدعاء تابع `approve` على `PendingReviewPost` والطريقة الوحيدة للحصول على `PendingReviewPost` هي استدعاء تابع `request_review` على `DraftPost`، إذ رمّزنا الآن سير عمل منشور المدونة إلى نظام النوع.

يتعين علينا أيضًا إجراء بعض التغييرات الصغيرة على `main`، إذ يُعيد التابعان `request_review` و `approve` حاليًا نسخًا جديدة بدلًا من تعديل الهيكل الذي استدعيت عليهما، لذلك نحتاج إلى إضافة المزيد من الإسنادات الخفية `let post =` لحفظ الأمثلة المُعادة. لا يمكن أيضًا أن تكون لدينا تأكيدات بسلاسل نصية فارغة حول محتويات المسودة ومنشورات بانتظار المراجعة، فنحن لسنا بحاجتها. لا يمكننا تصريف الشيفرة البرمجية التي تحاول استعمال محتوى المنشورات في تلك الحالات بعد الآن. تظهر الشيفرة البرمجية الجديدة ضمن الدالة `main` في الشيفرة 21.

اسم الملف: src/main.rs

```
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");

    let post = post.request_review();

    let post = post.approve();

    assert_eq!("I ate a salad for lunch today", post.content());
}
```

[الشيفرة 21: تعديل `main` لاستعمال التنفيذ الجديد لسير عمل منشور مدونة]

تعني التغييرات التي احتجنا لإجرائها على `main` من أجل إعادة تعيين `post` أن هذا التنفيذ لم يعد يتبع نمط الحالة كائنية التوجه بعد الآن، إذ لم تعد كامل التحوّلات بين الحالات مغلّفة في تنفيذ `Post`، ومع ذلك فإن النقطة التي بصالحنا هنا هي أن الحالات غير الصالحة أصبحت الآن مستحيلة بسبب نظام النوع والتحقق من

النوع الذي يحدث في وقت التصريف، إذ يضمن ذلك اكتشاف أخطاء معينة مثل عرض محتوى منشور لم يُنشر قبل الوصول لمرحلة الإنتاج.

جرّب المهام المقترحة في بداية القسم على وحدة `blog` المصرفة كما هي بعد الشيفرة 21 لمعرفة ما هو رأيك في تصميم هذا الإصدار من الشيفرة البرمجية. لاحظ أن بعض المهام قد تكون مكتملة فعلًا في هذا التصميم.

رأينا أنه على الرغم من أن رست قادرة على تنفيذ أنماط تصميم كائنية التوجه، إلا أن أنماطًا أخرى مثل ترميز الحالة في نظام النوع متاحة أيضًا في رست. هذه الأنماط لها مقايضات مختلفة. يمكن أن تكون على دراية كبيرة بالأنماط كائنية التوجه لكن يمكن أن توفّر إعادة التفكير في المشكلة للاستفادة من ميزات رست عدّة فوائد مثل منع بعض الأخطاء في وقت التصريف. لن تكون الأنماط كائنية التوجه هي الحل الأفضل دائمًا في رست نظرًا لوجود ميزات معينة مثل الملكية التي لا تمتلكها اللغات كائنية التوجه.

17.4 خاتمة

بعد قراءة هذا الفصل، تعلم الآن أنه يمكنك استخدام كائنات السمة للحصول على بعض الميزات كائنية التوجه في رست بغض النظر عما إذا كنت تعتقد أن رست هي لغة كائنية التوجه. يمكن أن يمنح الإيفاد الديناميكي للشيفرة البرمجية الخاصة بك مرونةً مقابل خسارة ضئيلة من سرعة وقت التنفيذ. يمكنك استخدام هذه المرونة لتنفيذ أنماط كائنية التوجه يمكن أن تساعد في صيانة الشيفرات البرمجية الخاصة بك. تحتوي رست أيضًا على ميزات أخرى مثل الملكية التي لا تتوفر في اللغات كائنية التوجه. لن يكون النمط كائني التوجه أفضل طريقة دائمًا للاستفادة من نقاط قوة رست ولكنه خيار متاح.

سنلقي نظرةً في الفصل التالي على الأنماط patterns وهي إحدى ميزات رست الأخرى التي تتيح قدرًا كبيرًا من المرونة، وعلى الرغم من أننا ألقينا نظرةً على المفهوم بإيجاز في جميع أنحاء الكتاب إلا أننا لم نر إمكاناتها الكاملة حتى الآن.

18. الأنماط والمطابقات

الأنماط Patterns هي صيغة خاصة في رست للمطابقة مع هيكل الأنـواع المعقـدة والبسـيطة، إذ يمنحـك استخدام الأنماط مع تعبير `match` وبُنى أخرى تحكمًا أكبر على سير البرنامج، ويتألف النمط من خليط مما يلي:

- قيم مُجرّدة Literals.
- مصفوفات Arrays أو معدّدات Enums أو هياكل structs أو صفوف Tuples مفكّكة.
- متغيرات Variables.
- محارف بدل Wildcards.
- مواضع مؤقتة Placeholders.

تتضمن بعض الأمثلة عن الأنماط `x` أو `(a, 3)` أو `Some(Color::Red)`، وتصف هذه المكونات شـكل البيانات في الحالات التي تكون فيها الأنماط محققة. يطابق البرنامج بعدها القيم مـع الأنمـاط للتحقـق فيمـا إذا كان لديها الشكل الصحيح من البيانات لاستمرار تنفيذ قسم معين من الشيفرة البرمجية.

لاستخدام النمط نقارنه مع قيمة ما، فإذا كان النمـط يطـابق القيمـة نسـتخدم أقسـام القيمـة في الشـيفرة. لنتذكر تعبير `match` المذكور سابقًا في الفصل 6، والذي كـان يسـتخدم الأنمـاط في مثـال آلـة تـرتيب العملات المعدنية؛ فإذا كانت القيمة تطابق شكل النمط يمكن استخدام القطع المسماة؛ وإذا لم تطابق القطعة نمطًا مـا، لن تُنفَّذ الشيفرة البرمجية الخاصة به.

يعدّ هذا الفصل مرجعًا لكل الأمـور المرتبطـة بالأنمـاط، إذ سـنغطي الحـالات المناسـبة لاسـتخدام الأنمـاط والفرق بين الأنماط القابلة للجدل refutable وغير القابلة للجدل irrefutable إضافةً إلى أنواع التعابير المختلفة

للأنماط التي قد تستخدمها، وسنتعرف في نهاية الفصل على كيفية استخدام الأنماط للتعبير عن مفاهيم متعددة بطريقة واضحة.

18.1 الأنماط Patterns واستخداماتها

تظهر الأنماط في العديد من الأماكن في رست، وقد استخدمتها سابقًا دون ملاحظتها غالبًا. سنتحدّث في هذا القسم عن جميع الحالات التي يكون فيها استخدام الأنماط صالحًا، إضافةً إلى الحالات التي تكون فيها قابلة للدحض Refutable.

18.1.1 أذرع تعبير match

كما تحدثنا سابقًا في الفصل 6، يمكننا استخدام الأنماط في أذرع arms تعبير `match`، ويُعرَّف التعبير `match` بالكلمة المفتاحية `match`، تليها قيمة للتطابق معها وواحد أو أكثر من أذرع المطابقة التي تتألف من نمط وتعبير يُنفذ إذا طابقت القيمة نمط الذراع على النحو التالي:

```
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
}
```

على سبيل المثال إليك تعبير `match` من الشيفرة 5 (من الفصل المذكور آنفًا) الذي يطابق القيمة `Option<i32>` في المتغير `x`:

```
match x {
    None => None,
    Some(i) => Some(i + 1),
}
```

الأنماط في تعبير `match` السابق، هما: `None` و `Some(i)` على يسار كل سهم.

أحد مُتطلبات تعبير `match` هو ضرورة كونه شاملًا لجميع الحالات، أي ينبغي أخذ جميع القيم المُحتملة بالحسبان في تعبير `match`. أحد الطرق لضمان تغطية شاملة الاحتمالات هي وجود نمط مطابقة مع الكل Catchall pattern للذراع الأخير، فلا يمكن مثلًا فشل تطابق اسم متغير لأي قيمة إطلاقًا وبذلك يغطي كل حالة متبقية.

يطابق النمط المحدد _ أي قيمة، لكن لا يرتبط بمتغير، لذلك يُستخدم غالبًا في ذراع المطابقة الأخـير. يمكن أن يكون النمط _ مفيدًا عندما نريد تجاهل أي قيمة غير محددة مثلًا، وسنتحدث لاحقًا بتفصيل أكثر عن النمط _ في قسم "تجاهل القيم في النمط".

18.1.2 تعابير if let الشرطية

تحدثنا في الفصل 6 عن كيفية استخدام تعابير `if let` بصورةٍ أساسية مثل طريقة مختصرة لكتابة مكافئ التعبير `match`، بحيث يطابق حالةً واحدةً فقط، ويمكن أن يكون التعبير `if let` مترافقًـا مـع `else` اختياريًـا، بحيث يحتوي على شيفرة برمجية تُنفّذ في حال لم يتطابق النمط في `if let`.

تبيّن الشيفرة 1 أنه من الممكن مزج ومطابقة تعـابير `if let` و `else if` و `else if let`، لإعطـاء مرونة أكثر من استخدام التعبير `match` الذي يقارن قيمةً واحدة مع الأنماط. إضـافةً إلى ذلـك، لا تتطلب رسـت أن تكون الأذرع في سلسلة `if let` أو `else if` أو `else if let` متعلقة ببعضها بعضًا.

تحدد الشيفرة 1 لون الخلفية اعتمادًا على تحقّق عدد من الشروط، إذ أنشأنا في مثالنا هذا متغيرات مع قيم مضمّنة في الشيفرة بحيث يمكن لبرنامج حقيقي استقبالها مثل مدخلات من المستخدم.

اسم الملف: src/main.rs

```
fn main() {
    let favorite_color: Option<&str> = None;
    let is_tuesday = false;
    let age: Result<u8, _> = "34".parse();

    if let Some(color) = favorite_color {
        println!("Using your favorite color, {color}, as the
background");
    } else if is_tuesday {
        println!("Tuesday is green day!");
    } else if let Ok(age) = age {
        if age > 30 {
            println!("Using purple as the background color");
        } else {
            println!("Using orange as the background color");
        }
    } else {
        println!("Using blue as the background color");
    }
```

```
}
```

[الشيفرة 1: استخدام `if let` و `else if` و `else if let` و `else` بنفس الوقت]

يُستخدم اللون المفضّل للمستخدم لونًا للخلفية إذا حدده المستخدم، وفي حال لم يُحدد لونه المفضل وكان اليوم خميس فسيكون لون الخلفية أخضر؛ وإذا حدّد المستخدم عمره في سلسلة نصية string، يمكننا تحليلها إلى رقم بنجاح، ويكون اللون إما برتقاليًا أو بنفسجيًا اعتمادًا على قيمة هذا الرقم؛ وأخيرًا إذا لم تنطبق أي من هذه الشروط سيكون لون الخلفية أزرق.

يسمح لنا هذا الهيكل الشرطي بدعم المتطلبات المعقدة، إذ نطبع في هذا المثال باستخدام القيم المضمّنة في الشيفرة ما يلي:

```
Using purple as the background color
```

يمكنك ملاحظة أن التعبير `if let` تسبب بحصولنا على متغير مخفي shadowed variable بالطريقة ذاتها التي تسببت بها أذرع التعبير `match`؛ إذ يُنشئ السطر `if let Ok(age) = age` متغيرًا مخفيًا جديد يدعى `age` يحتوي على القيمة داخل المتغاير `Ok`، وهذا يعني أنه يجب إضافة الشرط `if age > 30` داخل الكتلة؛ إذ لا يمكننا جمع الشَرطين في `if let ok(age) = age && age > 30`، والقيمة الخفية `age` التي نريد مقارنتها مع 30 غير صالحة حتى يبدأ النطاق scope الجديد بالقوس المعقوص curly bracket.

الجانب السلبي في استخدامنا لتعبير `if let` هو أن المصرّف لا يتحقق من شمولية الحالات مثلما يفعل تعبير `match`. إذا تخلّينا عن كتلة `else` الأخيرة وبالتالي فوّتنا معالجة بعض الحالات، لن ينبهنا المصرّف على احتمالية وجود خطأ منطقي.

18.1.3 حلقات while let الشرطية

تسمح الحلقة الشرطية `while let` بتنفيذ حلقة `while` طالما لا يزال النمط مُطابقًا على نحوٍ مشابه لبُنية `if let`. كتبنا في الشيفرة 2 حلقة `while let` تستخدم شعاعًا مثل مكدّس stack وتطبع القيم في الشعاع بالترتيب العكسي لإدخالها.

```
fn main() {
    let mut stack = Vec::new();

    stack.push(1);
    stack.push(2);
    stack.push(3);
```

```
    while let Some(top) = stack.pop() {
        println!("{}", top);
    }
}
```

[الشيفرة 2: استخدام حلقة `while let` لطباعة القيم طالما يُعيد استدعاء `stack.pop()` المتغاير `Some`]

يطبع المثال السابق القيمة 3 ثم 2 ثم 1، إذ يأخذ التابع `pop` آخر عنصر في الشعاع vector ويُعيد `Some(value)`، ويعيد القيمة `None` إذا كان الشعاع فارغًا. يستمر تنفيذ الحلقة `while` والشيفرة البرمجية داخل كتلتها طالما يُعيد استدعاء `pop` القيمة `Some`، وعندما يعيد استدعاء `pop` القيمة `None` تتوقف الحلقة، ويمكننا استخدام `while let` لإزالة pop off كل عنصر خارج المكدّس.

18.1.4 حلقات for التكرارية

القيمة التي تتبع الكلمة المفتاحية `for` في حلقة `for` هي النمط، فعلى سبيل المثال النمط في `for x in y` هو `x`. توضح الشيفرة 3 كيفية استخدام النمط في حلقة `for` لتفكيك destructure أو تجزئة الصف tuple إلى جزء من حلقة `for`.

```
fn main() {
    let v = vec!['a', 'b', 'c'];

    for (index, value) in v.iter().enumerate() {
        println!("{} is at index {}", value, index);
    }
}
```

[الشيفرة 3: استخدام نمط في حلقة `for` لتفكيك صف]

تطبع الشيفرة 3 ما يلي:

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
    Finished dev [unoptimized + debuginfo] target(s) in 0.52s
     Running `target/debug/patterns`
a is at index 0
b is at index 1
c is at index 2
```

نُعدل مكرّرًا iterator باستخدام التابع `enumerate` بحيث يعطينا قيمةً ودليلًا index لهذه القيمة ضمن صف tuple، بحيث تكون القيمة الأولى هي الصف `(0, 'a')`. عندما تطابق هذه القيمة النمط `(index, value)` ستكون `index` هي `0` وستكون `value` هي `a`، مما سيتسبب بطباعة السطر الأول من الخرج.

18.1.5 تعليمات let

تحدثنا سابقًا عن استخدام الأنماط مع `match` و `if let` فقط، إلا أننا استخدمنا الأنماط في أماكن أُخرى أيضًا مثل تعليمة `let`. ألقِ نظرةً على عملية إسناد المتغير التالية باستخدام `let`:

```
#![allow(unused)]
fn main() {
let x = 5;
}
```

اِستخدمت الأنماط في كلّ مرة استخدمت فيها تعليمة `let` بالشكل السابق دون معرفتك لذلك. تكون تعليمة `let` بالشكل التالي:

```
let PATTERN = EXPRESSION;
```

يشكّل اسم المتغير في التعليمات المشابهة لتعليمة `let x = 5;` -مع اسم المتغير في فتحة PATTERN- نوعًا بسيطًا من الأنماط. تقارن رست التعابير مع الأنماط وتُسند أي اسم تجده، أي تتألف التعليمة `let x = 5;` من نمط هو `x` يعني "اربط ما يتطابق هنا مع المتغير `x`"، ولأن الاسم `x` يمثّل كامل النمط، فإن هذا النمط يعني "اربط كل شيء مع المتغير `x` مهما تكُن القيمة".

لملاحظة مطابقة النمط في `let` بوضوح أكبر، جرّب الشيفرة 4 التي تستخدم نمطًا مع `let` لتفكيك الصف.

```
    let (x, y, z) = (1, 2, 3);
```

[الشيفرة 4: استخدام نمط لتفكيك الصف وإنشاء ثلاثة متغيرات بخطوة واحدة]

طابقنا الصف مع النمط هنا، إذ تُقارن رست القيمة `(1, 2, 3)` مع النمط `(x, y, z)` وترى أن القيمة تطابق النمط فعلًا، لذلك تربط رست `1` إلى `x` و `2` إلى `y` و `3` إلى `z`. يمكن عدّ نمط الصف هذا بمثابة تضمين ثلاثة أنماط متغيرات مفردة داخله.

إذا كان عدد العناصر في النمط لا يطابق عدد العناصر في الصف، لن يُطابَق النوع بأكمله وسنحصل على خطأ تصريفي. تبيّن الشيفرة 5 على سبيل المثال محاولة تفكيك صف بثلاثة عناصر إلى متغيرين وهي محاولة فاشلة.

```
    let (x, y) = (1, 2, 3);
```

[الشيفرة 5: بناء خاطئ لنمط لا تطابق متغيراته عدد العناصر الموجودة في الصف]

ستتسبب محاولة تصريف الشيفرة السابقة بالخطأ التالي:

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
error[E0308]: mismatched types
 --> src/main.rs:2:9
  |
|     let (x, y) = (1, 2, 3);
  |         ^^^^^^   --------- this expression has type `({integer},
{integer}, {integer})`
  |         |
  |         expected a tuple with 3 elements, found one with 2
elements
  |
  = note: expected tuple `({integer}, {integer}, {integer})`
             found tuple `(_, _)`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `patterns` due to previous error
```

يمكننا تجاهل قيمة أو أكثر في الصف لتصحيح الخطأ وذلك باستخدام _ أو .. كما سنرى لاحقًا في قسم "تجاهل القيم في النمط". إذا كانت المشكلة هي وجود عدة متغيرات في النمط فإن الحل يكمن بجعل الأنواع تتطابق عن طريق إزالة المتغيرات حتى يتساوى عدد المتغيرات وعدد العناصر في الصف.

18.1.6 معاملات الدالة Function Parameters

يمكن لمعاملات الدالة أن تمثّل أنماطًا. ينبغي أن تكون الشيفرة 6 مألوفةً بالنسبة لك، إذ نصرّح فيها عن دالة اسمها `foo` تقبل معاملًا واحدًا اسمه `x` من النوع `i32`.

```
fn foo(x: i32) {
    // تُكتب الشيفرة البرمجية هنا
}
```

[الشيفرة 6: بصمة دالة function signature تستخدم الأنماط في معاملاتها]

يشكّل الجزء `x` نمطًا. يمكننا مطابقة الصف في وسيط الدالة مع النمط كما فعلنا مع `let`. تجـزِّء الشـيفرة 7 القيم الموجودة في الصف عندما نمررها إلى الدالة.

اسم الملف: src/main.rs

```
fn print_coordinates(&(x, y): &(i32, i32)) {
    println!("Current location: ({}, {})", x, y);
}

fn main() {
    let point = (3, 5);
    print_coordinates(&point);
}
```

[الشيفرة 7: دالة تفكك الصف مع معاملاتها]

تطبع الشيفرة السابقة: `Current location: (3, 5)`، إذ تطابق القيم `&(3, 5)` النمط `&(x, y)`، لذا فإن قيمة `x` هي `3` وقيمة `y` هي `5`.

يمكننا أيضًا استخدام الأنمـاط في قـوائم معـاملات التغليـف parameter lists بطريقـة قـوائم closure معـاملات الدالـة parameter lists ذاتهـا function، لأن المغلفـات مشـابهة للـدالات كمـا رأينـا سـابقًا في الفصل 13.

رأينا بحلول هذه النقطة عدة طرق لاستخدام الأنماط، إلا أن الأنماط لا تعمـل بالطريقـة ذاتهـا في كـل مكـان تُسـتخدم فيـه، إذ يجب أن تكـون الأنمـاط غـير قابلـة للـدحض في بعض الأمـاكن وفي بعضـها الآخـر كـذلك، وسنتحدث عن هذين المفهومين تاليًا.

18.2 قابلية الدحض refutability: احتمالية فشل مطابقة النمط

تأتي الأنمـاط بشـكلين: قابلـة للـدحض أو النقض refutable وغـير قابلـة للـدحض irrefutable، إذ تُـدعى الأنمـاط الـتي تطـابق أي قيمـة تمـرر خلالهـا بالأنمـاط القابلـة للـدحض، ومثـال على ذلـك هـو `x` في التعليمة `let x = 5;` وذلك لأن المتغير `x` سيطابق أي شيء وبالتالي لا تفشل المطابقة؛ بينما تُـدعى الأنمـاط الـتي تفشـل في بعض القيم بالأنمـاط القابلـة للـدحض، ومثـال على ذلـك هـو `Some(x)` في التعبـير `if let Some(x) = a_value` لأن النمط `Some(x)` لن يُطابق إذا كانت القيمة في المتغير `a_value` هي `None` عوضًا عن `Some`.

تقبـل معـاملات الدالـة وتعليمـات `let` وحلقـات `for` فقـط الأنمـاط غـير القابلـة للـدحض لأن البرنـامج لا يستطيع عمل أي شيء مفيد عندما لا تتطابق القيم. يقبل التعبيران `if let` و `while let` الأنمـاط القابلـة

للدحض وغير القابلة للدحض، إلا أنّ المصرّف يحذّر من استخدام الأنماط غير القابلة للدحض، لأنها -بحسب تعريفها- ليست معدّة لتتعامل مع فشل محتمل، إذ تتمثّل الوظيفة الشرطية بقدرتها على التصرف بصورةٍ مختلفة اعتمادًا على النجاح أو الفشل.

عمومًا، لا يهم كثيرًا التمييز بين الأنماط القابلة للدحض وغير القابلة للدحض، ولكن يجب أن يكون مفهوم قابلية الدحض مألوفًا، وذلك لحل الأخطاء التي قد تحصل، إذ يجب في تلك الحالات تغيير إما النمط أو البنية construct المستخدمة مع النمط حسب السلوك المُراد من الشيفرة.

لنتابع مثالًا لما قد يحصل عندما نجرب استخدام نمط قابل للدحض عندما تتطلب رست نمطًا غير قابل للدحض -والعكس صحيح- إذ تبيّن الشيفرة 8 تعليمة `let` إلا أن النمط الذي حددناه هو `Some(x)` وهو نمط قابل للدحض، ولن تُصرَّف الشيفرة كما هو متوقع.

```
let Some(x) = some_option_value;
```

[الشيفرة 8: محاولة استخدام نمط قابل للدحض مع `let`]

ستفشل مطابقة النمط في `Some(x)` إذا كانت القيمة في `some_option_value` هي `None` أي أن النمط هو قابل للدحض، ولكن تقبل تعليمة `let` فقط الأنماط غير القابلة للدحض لأنه لا توجد قيمة صالحة تستطيع الشيفرة استخدامها مع قيمة `None`. تنبّهنا رست عند استخدام قيمة قابلة للدحض عندما يتطلب الأمر وجود قيمة غير قابلة للدحض وقت التصريف:

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
error[E0005]: refutable pattern in local binding: `None` not covered
 --> src/main.rs:3:9
  |
|     let Some(x) = some_option_value;
  |         ^^^^^^^ pattern `None` not covered
  |
  = note: `let` bindings require an "irrefutable pattern", like a
`struct` or an `enum` with only one variant
  = note: for more information, visit
https://doc.rust-lang.org/book/ch18-02-refutability.html
note: `Option<i32>` defined here
 --> 
/rustc/d5a82bbd26e1ad8b7401f6a718a9c57c96905483/library/core/src/optio
n.rs:518:1
  |
```

```
  = note:
/rustc/d5a82bbd26e1ad8b7401f6a718a9c57c96905483/library/core/src/
option.rs:522:5: not covered
  = note: the matched value is of type `Option<i32>`
help: you might want to use `if let` to ignore the variant that isn't
matched
  |
|     let x = if let Some(x) = some_option_value { x } else { todo!
() };
  |     ++++++++++                                  ++++++++++++++++++
+++
help: alternatively, you might want to use let else to handle the
variant that isn't matched
  |
|     let Some(x) = some_option_value else { todo!() };
  |                                   ++++++++++++++++

For more information about this error, try `rustc --explain E0005`.
error: could not compile `patterns` due to previous error
```

تُعطينا رست الخطأ التصريفي السابق، وذلك بسبب عدم تغطيتنا لكل القيم الممكنة مع النمط `Some(x)`، ولن نستطيع فعل ذلك حتى لو أردنا.

يمكننا إصلاح المشكلة في حال وجود نمط قابل للدحض يحل مكان نمط غير قابل للدحض عن طريق تغيير الشيفرة التي تستخدم النمط؛ فبدلًا من استخدام `let` نستخدم `if let`، وهكذا إذا لم يُطابق النمط تتخطى الشيفرة تلك الشيفرة الموجودة في القوسين المعقوصين وتستمر بذلك صلاحية الشيفرة. تبيّن الشيفرة 9 كيفية إصلاح الخطأ في الشيفرة 8.

```
    if let Some(x) = some_option_value {
        println!("{}", x);
    }
```

[الشيفرة 9: استخدام `if let` وكتلة تحتوي على أنماط قابلة للدحض بدلًا من `let`]

سُمح للشيفرة السابقة بالتصريف، فهذه الشيفرة صالحة، على الرغم من أنه لا يمكن استخدام نمط غير قابل للدحض دون رسالة خطأ. إذا أعطينا `if let` نمطًا يطابق دومًا مثل `x` كما في الشيفرة 10، سيمنحنا المصرّف تنبيهًا.

```
fn main() {
```

```
    if let x = 5 {
        println!("{}", x);
    };
}
```

[الشيفرة 10: محاولة استخدام نمط غير قابل للدحض مع `if let`]

تشتكي رست من عدم منطقيّة استخدام `if let` مع نمط غير قابل للدحض:

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
warning: irrefutable `if let` pattern
 --> src/main.rs:2:8
  |
|     if let x = 5 {
  |        ^^^^^^^^^
  |
  = note: this pattern will always match, so the `if let` is useless
  = help: consider replacing the `if let` with a `let`
  = note: `#[warn(irrefutable_let_patterns)]` on by default

warning: `patterns` (bin "patterns") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.39s
     Running `target/debug/patterns`
5
```

يجب أن تستخدم مطابقة الأذرع match arms الأنماط القابلة للدحض لهذا السبب ما عدا الذراع الأخير، الذي يجب أن يطابق أي قيمة متبقية من النمط غير القابل للدحض. تسمح رست باستخدام نمط غير قابل للدحض في `match` باستخدام ذراع واحد فقط، ولكن الصياغة هذه ليست مفيدة ويمكن استبدالها بتعليمة `let` أبسط.

بعدما عرفنا أماكن استخدام الأنماط والفرق بين الأنماط القابلة للدحض وغير القابلة للدحض، دعنا نكمل طريقة الصياغة syntax التي يمكن استخدامها لإنشاء الأنماط.

18.3 صياغة أنماط التصميم Pattern Syntax

سنجمع في هذا الفصل الصياغة الصالحة في الأنماط وسنتحدث عن مكان استخدام كل واحد منها.

18.3.1 مطابقة القيم المجردة Literals

يمكننا مطابقة الأنماط مباشرةً مع القيم المجرّدة كما رأينا سابقًا في الفصل 6. تمنحنا الشيفرة البرمجية التالية بعض الأمثلة:

```
    let x = 1;

    match x {
=> println!("one"),
=> println!("two"),
=> println!("three"),
        _ => println!("anything"),
    }
```

تطبع هذه الشيفرة one لأن القيمة في x هي 1. تُعد هذه الصياغة مفيدة عندما تريد من الشيفرة أن تنفّذ عملًا ما عندما تحصل على قيمة معينة واحدة.

18.3.2 مطابقة المتغيرات المسماة Named Variables

المتغيرات المُسمّاة هي أنماط غير قابلة للجدل تطابق أي قيمة، وقد استخدمناها مرات عديدة سابقًا، ولكن هناك تعقيدات عند استخدامها في تعابير match. ينشئ تعبير match نطاقًا scope جديدًا، وبالتالي ستُخفي المتغيرات المُصرّح عنها على أنها جزء من النمط داخل match المتغيرات التي تحمل الاسم ذاته خارج هيكل match كما هو الحال في جميع المتغيرات. صرّحنا ضمن الشيفرة 11 عن متغير مُسمى x قيمته Some(5) ومتغير y قيمته 10، ثم أنشأنا تعبير match على القيمة x. ألقِ نظرةً على الأنماط في أذرع المطابقة و println! وحاول اكتشاف ماذا ستطبع الشيفرة قبل تنفيذ هذه الشيفرة أو متابعة القراءة.

اسم الملف: src/main.rs

```
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(y) => println!("Matched, y = {y}"),
        _ => println!("Default case, x = {:?}", x),
    }

    println!("at the end: x = {:?}, y = {y}", x);
```

[الشيفرة 11: تعبير match مع ذراع يتسبب بظهور متغير خفي y]

لنرى ما سيحصل عند تنفيذ تعبير match؛ إذ لا تتطابق القيمة المُعرّفة x مع ذراع المطابقة الأول في النمط لذا يستمر تنفيذ الشيفرة.

يتسبب النمط الموجود في ذراع المطابقة الثاني بإنشاء متغير جديد باسم y وهو يطابق أي قيمة داخل قيمة Some، وذلك لأننا في نطاق جديد داخل تعبير match، هذا المتغير الجديد yهو ليس المتغير y ذاته الذي صرّحنا عنه في البداية بقيمة 10. يطابق الإسناد الجديد للمتغير y أي قيمة داخل Some وهي القيمة الموجودة في x، وبالتالي ترتبط y الجديدة بقيمة Some الداخلية في x، وتبلغ تلك القيمة 5، لذلك يُنَفَذ تعبير الذراع ويطبع Matched, y = 5.

لن تُطابق الأنماط في ذراعي النمط الأولين إذا كانت القيمة في x هي None بدلًا من Some(5)، لذا ستُطابق القيم مع الشرطة السفلية underscore. لم نُضِف المتغير x في نمط ذراع الشرطة السفلية، لذلك يبقى x في التعبير هو المتغير x الخارجي الذي لم يُخفى، وتطبع match في هذه الحالة الافتراضية Default case, x = None.

عندما ينتهي تعبير match ينتهي نطاقه أيضًا، وينتهي أيضًا نطاق المتغير y الداخلي. تطبع آخر تعليمة println! ما يلي:

```
at the end: x = Some(5), y = 10
```

يجب علينا استخدام ذ ع مطابقة شرطي match guard conditional لإنشاء تعبير match يقارن قيم x و y الخارجية عوضًا عن تقديم متغير خفي، وسنتحدث عن ذلك لاحقًا في قسم "الشرطيات الإضافية مع دروع المطابقة".

18.3.3 الأنماط المتعددة Multiple Patterns

يمكننا مطابقة عدة أنماط في تعبير match باستخدام الصياغة | التي يمكن أن تكون النمط أو المعامل. نطابق في المثال التالي قيمة x مع أذ ع المطابقة، بحيث يحتوي أول ذراع على الخَيَار "أو or"، بمعنى أنه إذا طابقت القيمة x أحد القيمتين في الذراع تُنفَّذ شيفرة الذراع كما يلي:

```
    let x = 1;

    match x {
| 2 => println!("one or two"),
=> println!("three"),
        _ => println!("anything"),
    }
```

تطبع الشيفرة السابقة ما يلي:

```
one or two
```

18.3.4 مطابقة مجالات القيم باستخدام الصيغة ..=

تسمح صيغة `..=` بمطابقة مجال شامل من القيم، إذ تُنفَّذ الشيفرة البرمجية الخاصة بالذراع عندما يطابق النمط أي قيمة في مجال ما كما في الشيفرة التالية:

```
let x = 5;

match x {
    1..=5 => println!("one through five"),
    _ => println!("something else"),
}
```

تتطابق الذراع الأولى إذا كانت قيمة `x` هي 1 أو 2 أو 3 أو 4 أو 5، هذه الصياغة ملائمة لمطابقة قيم متعددة بدلًا من استخدام المعامل `|` للتعبير عن الفكرة ذاتها؛ إذا أردنا استخدام `1 | 2 | 3 | 4 | 5`، وستكون طريقة تحديد المجال أقصر خاصةً إذا أردنا مطابقة أي رقم بين 1 و 1,000.

يتحقق المصرّف وقت التصريف من أن المجال غير فارغ لأن أنواع المجالات التي تستطيع رست تمييز ما إذا كانت فارغة أم لا هي `char` والقيم العددية، إذ يُسمح باستخدام المجالات فقط مع القيم العددية أو قيم `char`.

إليك مثالًا يستخدم مجال من قيم `char`:

```
let x = 'c';

match x {
    'a'..='j' => println!("early ASCII letter"),
    'k'..='z' => println!("late ASCII letter"),
    _ => println!("something else"),
}
```

تميز رست أن `'c'` تقع داخل مجال النمط الأول وتطبع `early ASCII letter`.

18.3.5 التفكيك Restructuring لتجزئة القيم

يمكننا استخدام الأنماط لتفكيك الهياكل أو المعدّدات enums أو الصفوف tuples لاستخدام الأجزاء المختلفة من القيم، لنستعرض كل حالة من الحالات السابقة.

ا. تفكيك الهياكل

تظهر الشيفرة 12 هيكل `Point` بحقلين `x` و `y` يُمكن تجزئتهما باستخدام نمط مع التعليمة `let`.

اسم الملف:src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x: a, y: b } = p;
    assert_eq!(0, a);
    assert_eq!(7, b);
}
```

[الشيفرة 12: تفكيك حقل هيكل إلى متغيرات منفصلة]

تُنشئ الشيفرة السابقة المتغيرين `a` و `b` اللذين يطابقان قيمتي الحقلين `x` و `y` في الهيكل `p`، ويوضح هذا المثال أنه ليس من الضروري لأسماء المتغيرات في النمط أن تُطابق أسماء الحقول في الهيكل، ولكن من الشائع مطابقة أسماء المتغيرات مع أسماء الحقول لتذكر ارتباط المتغير بحقل معين. بما أن كتابة ما يلي مثلًا:

```
let Point { x: x, y: y } = p;
```

تحتوي على الكثير من التكرار، لدى رست طريقةً مختصرة للأنماط التي تطابق حقول الهيكل؛ إذ عليك فقط أن تُضيف اسم حقل الهيكل مما يجعل المتغيرات المُنشأة من هذا النمط تحمل الاسم ذاته. تعمل الشيفرة 13 بطريقة عمل الشيفرة 12 ذاتها إلا أن المتغيرات المُنشأة في النمط `let` هي `x` و `y` بدلًا من `a` و `b`.

اسم الملف: src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
```

```
    let p = Point { x: 0, y: 7 };

    let Point { x, y } = p;
    assert_eq!(0, x);
    assert_eq!(7, y);
}
```

[الشيفرة 13: تفكيك حقول هيكل باستخدام طريقة حقل الهيكل المختصرة]

تُنشئ الشيفرة المتغيرين x و y اللذين يطابقان الحقلين x و y الخاصين بالهيكل p، والنتيجة هي احتواء المتغيرين x و y على القيم الموجودة في الهيكل p.

يمكننا إجراء عملية التفكيك باستخدام القيم المجردة مثل جزء من نمط الهيكل بدلًا من إنشاء متغيرات لكل الحقول، ويسمح لنا ذلك باختبار بعض الحقول لقيم معينة أثناء إنشاء متغيرات لتفكيك حقول أخرى.

لدينا في الشيفرة 14 تعبير `match` يقسم قيم `Point` إلى ثلاث حالات: نقاط تقع مباشرةً على محور x (الذي يُعدّ محققًا عندما `y = 0`)، ونقاط تقع على المحور y (أي `x = 0`)، ونقاط لا تقع على أي من المحورين.

اسم الملف: src/main.rs

```
fn main() {
    let p = Point { x: 0, y: 7 };

    match p {
        Point { x, y: 0 } => println!("On the x axis at {x}"),
        Point { x: 0, y } => println!("On the y axis at {y}"),
        Point { x, y } => {
            println!("On neither axis: ({x}, {y})");
        }
    }
}
```

[الشيفرة 14: تفكيك ومطابقة القيم المجردة في نمط واحد]

ستتطابق الذراع الأولى مع أي نقطة تقع على المحور x عن طريق تحديد أن الحقل y يقابل القيمة المجردة 0، ويُنشئ النمط متغير x يمكن استخدامه في شيفرة هذا الذراع؛ وبصورةٍ مشابهة، ستتطابق الذراع الثانية مع أي نقطة تقع على المحور y عن طريق تحديد أن الحقل x يقابل القيمة 0 ويُنشئ النمط متغير y للقيمة في الحقل y، ولا تحدد الذراع الثالثة أي قيمة مجرّدة لذا تُطابق أي قيمة `Point` أُخرى ويُنشأ متغيران لكل من

الحقلين `x` و `y`. تطابِق القيمة `p` الذراع الثانية في هذا المثال بفضل احتواء `x` على 0 وتطبع هذه الشيفرة ما يلي: `On the y axis at 7`.

تذكر أن تعبير `match` يتوقف عن التحقق من الأذرع عندما يجد أول نمط مطابق لذا حتى لو كانت `Point { x: 0, y: 0}` موجودةً على المحورين `x` و `y` ستطبع الشيفرة فقط `On the x axis at 0`.

ب. تفكيك المعددات

فكّكنا سابقًا المعدّدات (الشيفرة 5 في الفصل 6) إلا أننا لم نتحدث صراحةً أن النمط لتفكيك المعدّد يوافق طريقة تخزين البيانات داخله. تستخدم الشيفرة 15 معدّدًا يدعى `Message` من الشيفرة 2 من الفصل 6 وتُكتب `match` مع أنماط تفكك كل من القيم الداخلية الخاصة بذلك المعدّد.

اسم الملف: src/main.rs

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {
    let msg = Message::ChangeColor(0, 160, 255);

    match msg {
        Message::Quit => {
            println!("The Quit variant has no data to destructure.");
        }
        Message::Move { x, y } => {
            println!("Move in the x direction {x} and in the y
direction {y}");
        }
        Message::Write(text) => {
            println!("Text message: {text}");
        }
        Message::ChangeColor(r, g, b) => {
            println!("Change the color to red {r}, green {g}, and blue
{b}",)
```

```
        }
    }
}
```

[الشيفرة 15: تفكيك متغايرات المعدّد التي تحتوي على أنواع مختلفة من القيم]

تطبع الشيفرة السابقة ما يلي:

```
Change the color to red 0, green 160, and blue 255
```

حاول تغيير قيمة `msg` لملاحظة تنفيذ الشيفرة البرمجية للأذرع الأُخرى.

لا يمكننا تفكيك القيم أكثر من ذلك في متغيرات المعدّد التي لا تحتوي على أي بيانات مثل `Message::Quit`، إذ يمكننا فقط مطابقة القيمة المجرّدة `Message::Quit` ولا يوجد أي متغيرات في النمط.

يمكن استخدام أنماط مشابهة للنمط الذي نحدده لمطابقة الهياكل من أجل متغايرات المعدّدات التي تشبه الهيكل مثل `Message::Move`، إذ نضع أقواس معقوصة بعد اسم المتغير وبعدها نضع الحقول مع المتغيرات لتجزئتها واستخدامها في شيفرة الذراع، واستخدمنا هنا الطريقة المُختصرة كما فعلنا في الشيفرة 13.

يشابه النمط الذي نستخدمه لمتغيرات المعدّدات التي تشبه الصفوف مثل `Message::Write` الذي يحتوي على صف مع عنصر واحد و `Message::ChangeColor` الذي يحتوي صف مع ثلاثة عناصر للنمط الذي نحدده لمطابقة الصفوف، ويجب أن يطابق عدد المتغيرات في النمط عدد العناصر في المتغير المُراد مطابقته.

ج. تفكيك الهياكل والمعددات المتداخلة

كانت أمثلتنا حتى الآن مقتصرةً على مطابقة الهياكل structs والمعدّدات enums بعمق طبقة واحدة، إلا أن المطابقة تعمل على العناصر المتداخلة أيضًا، فعلى سبيل المثال يمكن إعادة بناء الشيفرة 15 لتدعم ألوان RGB و HSV في رسالة `ChangeColor` كما تبين الشيفرة 16.

```
enum Color {
    Rgb(i32, i32, i32),
    Hsv(i32, i32, i32),
}

enum Message {
    Quit,
    Move { x: i32, y: i32 },
```

```
    Write(String),
    ChangeColor(Color),
}

fn main() {
    let msg = Message::ChangeColor(Color::Hsv(0, 160, 255));

    match msg {
        Message::ChangeColor(Color::Rgb(r, g, b)) => {
            println!("Change color to red {r}, green {g}, and blue
{b}");
        }
        Message::ChangeColor(Color::Hsv(h, s, v)) => {
            println!("Change color to hue {h}, saturation {s}, value
{v}")
        }
        _ => (),
    }
}
```

[الشيفرة 16: مطابقة معدّدات متداخلة]

يتطابق النمط في الذراع الأولى في تعبير `match` مع متغاير معدد `Message::ChangeColor`، الذي يحتوي على متغاير معدد `Color::Rgb`، ويرتبط النمط مع قيم `i32` الداخلية الثلاث، بينما يتطابق نمط الذراع الثانية مع متغاير معدد `Message::ChangeColor` ويطابق المعدّد الداخلي `Color::Hsv`، ويمكن تحديد هذه الشروط المعقدة في تعبير `match` واحد حتى لو كان هناك معدّدان.

د. تفكيك الهياكل والصفوف

يمكننا مزج ومطابقة وتضمين الأنماط المفككة بطرق معقدة أكثر، ويبين المثال التالي تفكيك معقد، إذ نضمّن هياكل وصفوف داخل صف ونفكك جميع القيم الأولية primitive:

```
    let ((feet, inches), Point { x, y }) = ((3, 10), Point { x: 3, y:
-10 });
```

تسمح لنا الشيفرة السابقة بتجزئة الأنواع المعقدة إلى الأجزاء المكونة لها لاستخدام القيم التي نريدها على نحوٍ منفصل.

تُعد التجزئة مع الأنماط طريقةً مناسبة لاستخدام أجزاء من القيم مثل قيمة من كل حقل في هيكل منفصلين عن بعضهم.

18.3.6 تجاهل القيم في نمط

من المفيد أحيانًا تجاهل القيم في نمط ما كما هو الحال في ذراع `match` الأخير، للحصول على مطابقة مع الكل catchall التي لا تفعل أي شيء بدورها سوى تعريف كل القيم الممكنة المتبقية. هناك عدة طرق لتجاهل قيم كاملة أو جزء من قيم في نمط، منها: استخدام نمط _ (الذي تطرقنا له سابقًا)، أو استخدام نمط _ مع نمط أخر، أو استخدام اسم يبدأ بشرطة سفلية، أو استخدام `..` لتجاهل باقي أجزاء القيمة، وسنتعرف أكثر عن مكان وسبب استخدام كل نوع من هذه الأنماط.

ا. تجاهل قيمة كاملة باستخدام _

استخدمنا الشرطة السفلية مثل نمط محرف بدل wildcard يُطابق أي قيمة ولكن لا يرتبط بها، هذا مفيد لذراع أخير في تعبير `match` ولكن يمكن استخدامه أيضًا في أي نمط مثل معاملات الدالة كما تبين الشيفرة 17.

اسم الملف: src/main.rs

```
fn foo(_: i32, y: i32) {
    println!("This code only uses the y parameter: {}", y);
}

fn main() {
    foo(3, 4);
}
```

[الشيفرة 17: استخدام _ في بصمة الدالة]

تتجاهل الشيفرة السابقة القيمة 3 المُمَررة مثل معامل أول تمامًا وتطبع:

```
This code only uses the y parameter: 4
```

ستحتاج في معظم الحالات لتغيير الوسيط عندما لا توجد حاجة لمعامل دالة معين وذلك كي لا يُضمَّن المعامل غير المُستخدم، ويُعد تجاهل معامل الدالة مفيدًا في حالة تطبيق سمة trait بحاجة لنوع وسيط معين ولكن لا يحتاج متن الدالة في التطبيق إلى أحد الأنماط. يمكنك تفادي الحصول على تنبيه من المصرّف عن معاملات الدالة غير المُستخدمة كما تفعل لو استعضت عنها باسم آخر.

ب. تجاهل أجزاء من القيمة باستخدام _ متداخلة

يمكن استخدام _ داخل نمط آخر لتجاهل جزء من القيمة، إذ من الممكن مثلًا اختبار جزء فقط من القيمة وألا تكون هناك حاجة لباقي الأجزاء في الشيفرة المرافقة التي نريد تنفيذها. تمثّل الشيفرة 18 الشيفرة المسؤولة عن تنظيم إعدادات القيم، إذ لا تسمح متطلبات العمل للمستخدم الكتابة فوق تعديل لإعداد موجود سابقًا ولكن يمكن إزالة ضبط الإعدادات واعطائه قيمة إذا كان غير مضبوط بعد.

```
    let mut setting_value = Some(5);
    let new_setting_value = Some(10);

    match (setting_value, new_setting_value) {
        (Some(_), Some(_)) => {
            println!("Can't overwrite an existing customized value");
        }
        _ => {
            setting_value = new_setting_value;
        }
    }

    println!("setting is {:?}", setting_value);
```

[الشيفرة 18: استخدام شرطة سفلية داخل الأنماط التي تطابق متغايرات Some عندما لا توجد حاجة لاستخدام القيمة داخل Some]

تطبع الشيفرة ما يلي:

```
Can't overwrite an existing customized value
```

وبعدها:

```
setting is Some(5)
```

لا نحتاج لمطابقة أو استخدام القيمة في كِلا متغايري Some في ذراع المطابقة الأولى، لكننا بحاجة لاختبار الحالة عندما يكون متغايرا Some هما setting_value و new_setting_value، وفي تلك الحالة نطبع سبب عدم تغيير setting_value ولا تتغيّر؛ ومن أجل باقي الحالات (إذا كانت setting_value أو new_setting_value هي None) مُعبرة بالنمط _ في الذراع الثانية، سنسمح للقيمة new_setting_value بأن تصبح setting_value.

يمكننا استخدام الشرطة السفلية في أماكن متعددة داخل نمط واحد لتجاهل قيمة معينة، وتبين الشيفرة 19 مثالًا عن تجاهل القيمتين الثانية والرابعة في صف مكون من خمس قيم.

```
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, _, third, _, fifth) => {
            println!("Some numbers: {first}, {third}, {fifth}")
        }
    }
```

[الشيفرة 19: تجاهل قيم متعددة في صف]

تطبع الشيفرة `Some numbers: 2, 8, 32` متجاهلةً القيمتين 4 و16.

ج. تجاهل المتغيرات غير المستخدمة بكتابة _ بداية اسمها

تنبّهك رست إذا أنشأت متغيرًا ولم تستخدمه في أي مكان، إذ يمكن أن يكون عدم استخدام متغير خطأ برمجي، ولكن من المفيد إنشاء متغير لا تريد استخدامه حاليًا مثل عندما نريد كتابة نموذج أولي prototype أو عند بداية مشروع جديد، ففي هذه الحالات يمكن إخبار رست بعدم التنبيه عن المتغيرات غير المستخدمة بكتابة شرطة سفلية "_" قبل اسم المتغير. أنشأنا في الشيفرة 20 متغيرين غير مستخدمين ولكن عندما نصرّف الشيفرة هذه يجب أن نحصل على تنبيه بخصوص واحد منهما فقط.

اسم الملف: src/main.rs

```
fn main() {
    let _x = 5;
    let y = 10;
}
```

[الشيفرة 20: كتابة اسم المتغير مسبوقًا بشرطة سفلية لتجنب الحصول على تنبيه متغير غير مُستخدم]

نحصل على تنبيه عن عدم استخدام المتغير `y` ولكن لن نحصل على تنبيه لعدم استخدام المتغير `_x`.

لاحظ أن هناك اختلاف بسيط بين استخدام _ فقط أو استخدام اسم مسبوقًا بشرطة سفلية، إذ تُسنِد الصيغة `_x` القيمة بالمتغير ولكن لا تُسند الصيغة _ أي قيمة إطلاقًا، ولتوضيح أهمية الفرق إليك الشيفرة 21 التي تعطينا الخطأ التالي.

```
    let s = Some(String::from("Hello!"));
```

```
    if let Some(_s) = s {
        println!("found a string");
    }

    println!("{:?}", s);
```

[الشيفرة 21: يُسند متغير غير مستخدم يبدأ بشرطة سفلية إلى قيمة، مما قد يمنحه ملكية القيمة]

سنحصل على خطأ لأن قيمة `s` ستنتقل إلى `_s` التي تمنع استخدام `s` مجددًا، بينما لا يُسند استخدام الشرطة السفلية لوحدها القيمة أبدًا. تُصرّف الشيفرة 22 التالية دون أي أخطاء لأن `s` لا تنتقل إلى `_`.

```
    let s = Some(String::from("Hello!"));

    if let Some(_) = s {
        println!("found a string");
    }

    println!("{:?}", s);
```

[الشيفرة 22: استخدام الشرطة السفلية لا يُسند القيمة]

تعمل الشيفرة بصورةٍ صحيحة لأن `s` لا تُسند لأي قيمة ولا يتغير مكانها.

د. تجاهل الأجزاء المتبقية من القيمة باستخدام الرمز ..

يمكننا استخدام الصيغة `..` لاستخدام أجزاء معينة من القيمة وتجاهل الباقي وذلك مع القيم التي تحتوي على أجزاء متعددة، ودون الحاجة لاستخدام الشرطة السفلية لكل قيمة مُتَجَاهلة؛ إذ يتجاهل النمط `..` أي جزء لم يُطابق من القيمة صراحةً في باقي النمط. لدينا في الشيفرة 23 هيكل `Point` يحتوي إحداثيات في الفضاء ثلاثي الأبعاد، ونريد في تعبير `match` أن نعمل فقط على إحداثيات `x` وتتجاهل القيم الموجودة في الحقلين `y` و `z`.

```
    struct Point {
        x: i32,
        y: i32,
        z: i32,
    }
```

```
    let origin = Point { x: 0, y: 0, z: 0 };

    match origin {
        Point { x, .. } => println!("x is {}", x),
    }
```

[الشيفرة 23: تجاهل كل الحقول في `Point` عدا الحقل `x` باستخدام `..`]

نضع القيمة `x` في قائمة وبعدها نضيف النمط `..`، وتُعد هذه الطريقة أسرع من كتابة كل من `y: _` و `z: _` وتحديدًا عند العمل مع هياكل تحتوي على العديد من الحقول وتريد الحصول على حقل واحد أو اثنين.

يمكن زيادة الصيغة `..` إلى عدد كبير من القيم حسب الحاجة، وتظهر الشيفرة 24 كيفية استخدام `..` مع صف.

اسم الملف: src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, .., last) => {
            println!("Some numbers: {first}, {last}");
        }
    }
}
```

[الشيفرة 24: مطابقة القيمتين الأولى والأخيرة في الصف وتجاهل باقي القيم]

تتطابق القيمتين الأولى والأخيرة مع `first` و `last` في هذه الشيفرة، وتطابق `..` القيمتين وتتجاهل كل شيء في المنتصف.

يجب استخدام `..` بوضوح، إذ تعطي رست رسالة خطأ إذا كانت القيمة المُراد مطابقتها والقيم المُتجاهلة غير واضحة. تبين الشيفرة 25 مثالًا لاستخدام `..` غير واضح ونتيجة لذلك فإن الشيفرة لا تُصرَّف.

اسم الملف: src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);
```

```
    match numbers {
        (.., second, ..) => {
            println!("Some numbers: {}", second)
        },
    }
}
```

[الشيفرة 25: محاولة استخدام `..` بطريقة غير واضحة]

عندما نصرّف الشيفرة في هذا المثال نحصل على الخطأ التالي:

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
error: `..` can only be used once per tuple pattern
 --> src/main.rs:5:22
  |
5 |         (.., second, ..) => {
  |          --          ^^ can only be used once per tuple pattern
  |          |
  |          previously used here

error: could not compile `patterns` due to previous error
```

من المستحيل أن تحدد رست عدد القيم التي تتجاهلها في الصف قبل مطابقة قيمة مع `second` وكم قيمة ستتجاهل بعدها، قد تعني هذه الشيفرة أننا نريد تجاهل 2 وإسناد `second` إلى 4 وبعدها تجاهل 8 و 16 و 32 أو قد تعني أيضًا تجاهل 2 و 4 وإسناد `second` إلى 8 وبعدها تجاهل 16 و 32 وهكذا. لا يعني اسم المتغير `second` أي شيء مميز لرست لذا نحصل على خطأ تصريفي لأن استخدام `..` في مكانين يجعل النمط غير واضح.

18.3.7 تعابير شرطية إضافية مع دروع المطابقة

درع المطابقة هو شرط `if` إضافي يُحدّد بعد النمط في ذراع `match` ويجب أن يطابق الذراع حتى يجري اختياره، وتفيد دروع المقابلة للتعبير عن أفكار معقّدة لا يمكننا التعبير عنها بالنمط لوحده.

يمكن أن يستخدم الشرط متغيرات مُنشأة في النمط، وتبين الشيفرة 26 تعليمة `match` تحتوي الذراع الأولى فيها على النمط `Some(x)` وأيضًا درع مطابقة `if x % 2 == 0` (الذي يكون صحيحًا إذا كان العدد زوجي).

```
    let num = Some(4);

    match num {
        Some(x) if x % 2 == 0 => println!("The number {} is even", x),
        Some(x) => println!("The number {} is odd", x),
        None => (),
    }
```

[الشيفرة 26: إضافة درع مطابقة إلى نمط]

ستطبع الشيفرة السابقة `The number 4 is even`، وتتطابق عندما تُقارن `num` مع النمط الموجود في الذراع الأولى، لأن `Some(4)` تطابق `Some(x)`. بعد ذلك، يتحقق درع المطابقة إذا كان الباقي من عملية قسمة `x` على 2 يساوي 0 ولأن هذه الحالة محقّقة يقع الاختيار على الذراع الأولى.

سيكون درع المطابقة في الذراع الأولى خاطئًا إذا كان `num` هو `Some(5)`، وذلك لأن باقي قسمة 5 على 2 هو 1 وهو لا يساوي 0، وتنتقل بعدها رست إلى الذراع الثانية التي ليس فيها درع مطابقة وبالتالي تطابق أي متغير `Some`.

لا توجد طريقة للتعبير عن شرط `if x % 2 == 0` داخل النمط، لذا يسمح لنا درع المطابقة بالتعبير عن هذا المنطق. سلبية هذا التعبير الإضافي هي أن المصرّف لا يتحقق من الشمولية عند تواجد تعابير درع مطابقة.

ذكرنا في الشيفرة 11 أنه يمكننا استخدام دروع المطابقة لحل مشكلة إخفاء النمط pattern-shadowing. تذكر أننا أنشأنا متغيرًا جديدًا داخل النمط في التعبير `match` عوضًا عن استخدام المتغير خارج `match`، ويعني إنشاء هذا المتغير الجديد أنه لا يمكن اختبار القيم مع المتغير الخارجي. تبين الشيفرة 27 كيفية استخدام درع المطابقة لحل هذه المشكلة.

اسم الملف: src/main.rs

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(n) if n == y => println!("Matched, n = {n}"),
        _ => println!("Default case, x = {:?}", x),
    }
```

```
    println!("at the end: x = {:?}, y = {y}", x);
}
```

[الشيفرة 27: استخدام درع المطابقة لاختبار المساواة مع متغير خارجي]

ستطبع الشيفرة الآن:

```
Default case, x = Some(5)
```

لا يقدم النمط في الذراع الثاني متغير y جديد يخفي المتغير y الخارجي، وهذا يعني أنه يمكن استخدام y الخارجية في درع المطابقة، إذ نحدّد Some(n) عوضًا عن تحديد النمط Some(y) الذي كان سيخفي بدوره المتغير y الخارجي، وسينشئ ذلك متغيرًا جديدًا يدعى n لا يُخفي أي شيء لعدم وجود أي متغير بالاسم n خارج match.

ليس درع المطابقة y if n == y نمطًا وبالتالي لا يقدم أي متغيرات جديدة، إذ أن y هذه هي y الخارجية ذاتها وليست y مخفية جديدة، ويمكن البحث عن قيمة لديها نفس قيمة y الخارجية بمقارنة n مع y.

يمكن استخدام المعامل "أو" | في درع المطابقة لتحديد الأنماط المتعددة، وسيُطبق شرط درع المطابقة على كل الأنماط. تبين الشيفرة 28 الأسبقية عند جمع نمط يستخدم | مع درع مطابقة، والقسم الأهم من هذا المثال هو درع المطابقة y if الذي يطبق على 4 و 5 و 6 على الرغم من أن y if تبدو أنها مطبقةٌ فقط على 6.

```
    let x = 4;
    let y = false;

    match x {
| 5 | 6 if y => println!("yes"),
        _ => println!("no"),
    }
```

[الشيفرة 28: جمع عدة أنماط مع درع مطابقة]

ينصّ شرط المطابقة على أن الذراع تتطابق فقط إذا كانت قيمة x تساوي 4 أو 5 أو 6 وإذا كانت قيمة y هي true، وعندما تُنفذ الشيفرة يُطَابَق نمط الذراع الأول لأن x هي 4 ولكن درع المطابقة y if خاطئ، لذا لا يقع الاختيار على الذراع الأولى وتنتقل الشيفرة إلى الذراع الثانية التي تُطابَق ويطبع البرنامج no، والسبب وراء ذلك هو تطبيق شرط if لكل النمط 4 | 5 | 6، وليس فقط للقيمة الأخيرة 6، بمعنى أخر تتصرف أسبقية درع المطابقة مع النمط على النحو التالي:

```
(4 | 5 | 6) if y => ...
```

عوضًا عن:

```
| 5 | (6 if y) => ...
```

سلوك الأسبقية واضح بعد تنفيذ الشيفرة، إذ ستُطابق الذراع وسيطبع البرنامج `yes` إذا كـان ذر ع المطابقـة مطبقًا فقط على القيمة الأخيرة في قائمة القيم المحددة بالمعامل `|`.

18.3.8 ارتباطات @

يسمح لنا معامل at `@` بإنشاء متغيرات تحتوي قيمة واختبارها من أجل مطابقة نمـط بنفس الـوقت. نريـد في الشيفرة 29 اختبار حقل `id` في `Message::Hello` إذا كان ضمن المجال `3..=7`، ونريد أيضًا ربط القيمـة إلى المتغير `id_variable` لكي نستخدمها في الشيفرة المرتبطة مع الذراع. يمكن تسمية هـذا المتغـير باسـم الحقل `id`، لكننا استخدمنا اسمًا مختلفًا.

```
enum Message {
        Hello { id: i32 },
    }

    let msg = Message::Hello { id: 5 };

    match msg {
        Message::Hello {
            id: id_variable @ 3..=7,
        } => println!("Found an id in range: {}", id_variable),
        Message::Hello { id: 10..=12 } => {
            println!("Found an id in another range")
        }
        Message::Hello { id } => println!("Found some other id: {}",
id),
    }
```

[الشيفرة 29: استخدام @ لربط القيمة في النمط مع اختبارها أيضًا]

سيطبع المثال السابق ما يلي:

```
Found an id in range: 5
```

نلتقط أي قيمة تطابق المجال `3..=7` بتحديد `id_variable @` قبله، إضافةً إلى اختبار نمط تلك القيمة المطابقة.

لا تحتوي الشيفرة في الذراع متغيرًا يحتوي على قيمة حقيقية في حقل `id` في الذراع الثانية، إذ لدينا فقط مجالًا محددًا في النمط. يمكن أن تكون قيمة الحقل `id` مساوية إلى 10 أو 11 أو 12 لكن لا تعرف الشيفرة التي في ذلك النمط أي قيمة هي منهم، ولا يستطيع نمط الشيفرة استخدام القيمة من حقل `id` لأننا لم نحفظ قيمة `id` في المتغير.

ليس لدينا قيمةً متوفرةً لاستخدامها في شيفرة الذراع الأخيرة في المتغير المسمى `id`، إذ حددنا متغيرًا دون مجال، ويعود سبب ذلك إلى استخدام صيغة حقل الهيكل المختزلة، وعدم تطبيق أي اختبار على القيمة في حقل id` في هذا الذراع كما فعلنا في الذراعين الأوليين، فأي قيمة ستطابق هذا النمط.

يسمح لنا استخدام `@` باختبار القيمة وحفظها في متغير داخل نمط واحد.

18.4 خاتمة

أنماط رست مفيدة في التمييز بين أنواع البيانات المختلفة، إذ تتأكد رست أن النمط يغطي قيمةً ممكنةً وإلا فلن يُصرّف البرنامج. تجعل الأنماط في تعليمة `let` ومعاملات الدالة من الهياكل ذات فائدة أكبر، وذلك عن طريق السماح بتفكيك القيم إلى أجزاء أصغر وتعيينها إلى متغيرات في الوقت نفسه، كما يمكننا إنشاء أنماط بسيطة أو معقدة لتلائم الحاجة.

سنتحدث في الفصل ما قبل الأخير ضمن هذا الكتاب عن الخواص المتقدمة للغة رست.

دورة تطوير التطبيقات
باستخدام لغة بايثون

احترف البرمجة وتطوير التطبيقات مع أكاديمية حسوب
والتحق بسوق العمل فور انتهائك من الدورة

التحق بالدورة الآن

19. ميزات متقدمة

لقد تعلمت حتى الآن الأجزاء الأكثر استخدامًا من لغة البرمجة رست. قبل أن ننفذ مشروعًا آخـر في الفصـل 20، سنلقي نظرةً على بعض جوانب اللغة التي قد تصادفها من حينٍ لآخر، ولكن قد لا تستعملها كل يوم. يمكنك استخدام هذا الفصل بمثابة مرجع عندما تواجه أي شـيء مجهـول. المـيزات الـتي نتطـرق إليهـا هنـا مفيـدة في مواقف محددة جدًا، وعلى الرغم من أنك قد لا تواجه هـذه المواقـف، إلا أننـا نريـد التأكـد من أنـك تمتلـك فهمًـا لجميع الميزات التي تقدمها رست.

سغطي في هذا الفصل:

- رست غير الآمنـة Rust unsafe: كيفيـة تجاهـل بعض ضـمانات رسـت وتحمـل مسـؤولية دعم هـذه الضمانات يدويًا.
- السمات المتقدمة: الأنواع المرتبطة associated types ومعاملات النوع الافتراضـية default type parameters والصيغة المؤهلة كليًا fully qualified syntax والسمات الخارقة supertraits ونمط النوع الجديد newtype pattern وعلاقته بالسمات traits.
- أنـواع متقدمـة: المزيـد عن نمـط النـوع الجديـد newtype pattern وأسـماء النـوع المسـتعارة type aliases ونوع أبدًا never type والأنواع التي يتغير حجمها ديناميكيًا dynamically sized types.
- دوال ومغلفات متقدمة: مؤشرات الدالة والمغلفات المُعادة.
- ماكرو: طرق لتعريف شيفرة برمجية تعرف بدورها شيفرات برمجية أكثر في وقت التصريف.

إنها لائحة تحتوي على الكثير من ميزات رست مع شيء مناسب للجميع. لنتعمق بها.

19.1 لغة رست غير الآمنة Unsafe Rust

لدى كل الشيفرات البرمجيـة الـتي ناقشـناها حـتى الآن ضـمانات لأمـان الـذاكرة في رسـت وتُفـرض هـذه الضمانات وقت التصريف، ومع ذلك فإن رست تحتوى على لغة ثانية مخبـأة داخلهـا لا تفـرض ضـمانات أمـان الـذاكرة هـذه، ويطلـق عليهـا اسـم رسـت غـير الآمنـة وتعمـل تمامًـا مثـل رسـت العاديـة ولكنهـا تمنحنـا قـوى خارقة إضافية.

توجد رست غير الآمنة لأن التحليل الساكن بطبيعته متحفظ، أي عندما يحاول المصرّف تحديد ما إذا كـانت الشيفرة البرمجية تـدعم الضـمانات أم لا، فمن الأفضـل لـه رفض بعض الـبرامج الصـالحة بـدلًا من قبـول بعض البرامج غير الصالحة. يمكن أن تكون الشـيفرة البرمجيـة تكـون جيـدة، إلا أن مصـرف رسـت سـيرفض تصـريف الشيفرة البرمجية إن لم يكن لديه معلومات كافية ليكون واثقًا، ويمكنك في هذه الحالات اسـتعمال شـيفرة غـير آمنة لإخبار المصرف "صـدقني، أعـرف مـا أفعلـه"، ومـع ذلـك كن حـذرًا من اسـتعمال رسـت غـير الآمنـة على مسؤوليتك الخاصة: إذا استعملت شيفرة غير آمنة على نحوٍ غير صحيح فقد تحدث بعض المشاكل بسبب عـدم أمان الذاكرة مثل تحصيل dereferencing مؤشر فار غ.

السبب الآخر لوجود لغة رست غير آمنة هو أن عتاد الحاسب الأساسي غير آمن بطبيعته، فإذا لم تسمح لك رست بإنجاز عمليات غير آمنة فلن يمكنك إنجاز مهـام معينـة. تحتـاج رسـت إلى السـماح لـك ببرمجـة الأنظمـة منخفضة المستوى مثل التفاعل المباشر مع نظام التشغيل أو حتى كتابة نظام التشغيل الخاص بك. يُعد العمـل مع برمجة الأنظمة منخفضة المستوى أحد أهداف اللغة. لنكتشف ما يمكننا فعله مع رسـت غـير الآمنـة وكيفيـة إنجاز ذلك.

19.1.1 القوى الخارقة غير الآمنة unsafe superpowers

استخدم الكلمة المفتاحية `unsafe` للتبديل إلى رست غير الآمنة، ثم ابدأ كتلة جديـدة تحتـوي على الشـيفرة غير الآمنة. يمكنك اتخاذ خمسة إجراءات في رست غير الآمنة لا يمكنك فعلها في رست الآمنة ونسـميها القـوى الخارقة غير الآمنة؛ تتضمن هذه القوى الخارقة القدرة على:

- تحصيل مؤشر خام raw pointer.
- استدعاء تابع أو دالة غير آمنين.
- الوصول أو التعديل على متغير ساكن static متغيّر mutable.
- تطبيق سمة trait غير آمنة.
- حقول الوصول الخاصة بـ `union`.

من المهم أن نفهم أن `unsafe` لا توقف تشغيل مدقق الاستعارة أو تعطل أي من فحوصات أمان رست الأخرى؛ وإذا كنت تستخدم مرجعًا في شيفرة غير آمنة فسيظل التحقق منه جاريًا. تمنحك الكلمة المفتاحية `unsafe` فقط الوصول إلى هذه الميزات الخمس التي لم يجري التحقق منها بعد ذلك من المصرف من أجل سلامة الذاكرة، وستظل تتمتع بدرجة من الأمان داخل كتلة غير آمنة. لا تعني `unsafe` أن الشيفرة الموجودة داخل الكتلة هي بالضرورة خطيرة أو أنها ستواجه بالتأكيد مشكلات تتعلق بسلامة الذاكرة، القصد هو أنه بصفتك مبرمجًا ستضمن أن الشيفرة الموجودة داخل كتلة `unsafe` ستصل إلى الذاكرة بطريقة صالحة.

ليس البشر معصومين والأخطاء تحصل، ويمكنك من خلال إجبار وجود هذه العمليات الخمس غير الآمنة داخل كتل موصّفة `unsafe` أن تضمن بقاء أي أخطاء متعلقة بأمان الذاكرة داخل كتلة `unsafe`. اجعل كتل `unsafe` صغيرة، ستقدر ذلك لاحقًا عندما تفتش عن أخطاء الذاكرة.

لعزل الشيفرة غير الآمنة قدر الإمكان، يُفضّل تضمين الشيفرة غير الآمنة في عملية تجريد آمنة وتوفير واجهة برمجة تطبيقات آمنة التي سنناقشها لاحقًا في هذا الفصل عندما نتحدث عن الدوال والتوابع غير الآمنة. تُطبَّق أجزاء من المكتبة القياسية مثل تجريدات آمنة على الشيفرات البرمجية غير الآمنة التي قد جرى تدقيقها. يمنع تغليف الشيفرة غير الآمنة في عملية تجريد آمنة استخدامات `unsafe` من التسرب إلى جميع الأماكن التي قد ترغب أنت أو المستخدمين لديك في استخدام الوظيفة المطبقة بشيفرة `unsafe` لأن استخدام التجريد الآمن آمن.

لنلقي نظرةً على كل من القوى الخارقة الخمس غير الآمنة بالترتيب، سنلقي نظرةً أيضًا على بعض الأفكار المجردة التي تقدم واجهةً آمنةً للشيفرات البرمجية غير الآمنة.

19.1.2 تحصيل مرجع مؤشر خام

ذكرنا سابقًا في الفصل 4 في قسم "المراجع المعلقة" أن المصرّف يضمن صلاحية المراجع دائمًا. تحتوي رست غير الآمنة على نوعين جديدين يدعيان المؤشرات الخام التي تشبه المراجع، فكما هو الحال مع المراجع يمكن أن تكون المؤشرات الخام ثابتة immutable أو متغيّرة mmutable وتُكتب بالطريقة `*const T` و `mut *T` على التوالي. لا تمثّل علامة النجمة عامل التحصيل وإنما هي جزءٌ من اسم النوع. يُقصد بمصطلح الثابت في سياق المؤشرات الخام أنه لا يمكن تعيين المؤشر مباشرةً بعد تحصيله.

تختلف المؤشرات الأولية عن المراجع والمؤشرات الذكية بما يلي:

- يُسمح بتجاهل قواعد الاستعارة من خلال وجود مؤشرات ثابتة أو متغيّرة أو مؤشرات متعددة متغيّرة إلى الموقع ذاته.
- ليست مضمونة للإشارة إلى ذاكرة صالحة.
- من المسموح أن تكون فارغة.

- لا تطبق أي تحرير ذاكرة تلقائي.

يمكنك التخلي عن الأمان المضمون من خلال تجاهل الضمانات التي تقدمها رست وذلك مقابل أداء أفضل أو القدرة على التفاعل مع لغة أو عتاد آخر لا تنطبق عليه ضمانات رست.

توضح الشيفرة 1 كيفية إنشاء مؤشر خام ثابت ومتغيّر من المراجع.

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

[الشيفرة 1: إنشاء مؤشرات خام من المراجع]

لاحظ أننا لا نضمّن الكلمة المفتاحية `unsafe` في هذه الشيفرة. يمكننا إنشاء مؤشرات خام في شيفرة آمنة، ولا يمكننا تحصيل المؤشرات الخام خارج كتلة غير آمنة كما سترى بعد قليل.

أنشأنا مؤشرات خام باستعمال `as` لتحويل مرجع ثابت ومتغيّر إلى أنواع المؤشرات الخام الخاصة بهما، ونظرًا إلى أننا أنشأناها مباشرةً من مراجع مضمونة لتكون صالحة فنحن نعلم أن هذه المؤشرات الخام المعنيّة صالحة لكن لا يمكننا افتراض هذا حول أي مؤشر خام.

لإثبات ذلك سننشئ مؤشر خام لا يمكننا التأكد من صحته. تظهر الشيفرة 2 كيفية إنشاء مؤشر خام يشير إلى موقع عشوائي في الذاكرة. محاولة استخدام الذاكرة العشوائية هي عملية غير معرّفة، إذ قد توجد بيانات في ذلك العنوان أو قد لا تكون موجودة، ومن الممكن للمصرّف أن يحسن الشيفرة بحيث لا يكون هناك وصول للذاكرة أو قد يخطئ البرنامج في وجود خطأ في التجزئة segmentation. لا يوجد عادةً سببٌ جيد لكتابة شيفرة مثل هذه، لكن هذا ممكن.

```
let address = 0x012345usize;
let r = address as *const i32;
```

[الشيفرة 2: إنشاء مؤشر خام إلى عنوان ذاكرة عشوائي]

تذكر أنه يمكننا إنشاء مؤشرات خام في شيفرة آمنة لكن لا يمكننا تحصيل المؤشرات الخام وقراءة البيانات التي يُشار إليها، ونستخدم في الشيفرة 3 عامل التحصيل `*` على مؤشر خام يتطلب كتلة `unsafe`.

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

```
    unsafe {
        println!("r1 is: {}", *r1);
        println!("r2 is: {}", *r2);
    }
```

[الشيفرة 3: تحصيل المؤشرات الخام ضمن كتلة unsafe]

لا يضرّ إنشاء مؤشر؛ إذ يكمن الضرر فقط عندما نحاول الوصول إلى القيمة التي تشير إليها، فقد ينتهي بنا الأمر بالتعامل مع قيمة غير صالحة.

لاحظ أيضًا أننا أنشأنا في الشيفرة 1 و3 مؤشرات خام من النوع *const i32 و *mut i32 التي أشارت كلتاهما إلى موقع الذاكرة ذاته، حيث يُخزَّن num. إذا حاولنا إنشاء مرجع ثابت ومتغيّر إلى num بدلًا من ذلك، فلن تُصرّف الشيفرة لأن قواعد ملكية رست لا تسمح بمرجع متغيّر في الوقت ذاته كما هو الحال مع أي مراجع ثابتة. يمكننا باستخدام المؤشرات الخام إنشاء مؤشر متغيّر ومؤشر ثابت للموقع ذاته وتغيير البيانات من خلال المؤشر المتغيّر مما قد يؤدي إلى إنشاء سباق بيانات data race. كن حذرًا.

مع كل هذه المخاطر، لماذا قد تستخدم المؤشرات الخام؟ إحدى حالات الاستخدام الرئيسية هي عند التفاعل مع شيفرة سي C كما سترى لاحقًا في القسم "استدعاء دالة أو تابع غير آمنين"، وهناك حالة أخرى عند بناء تجريدات آمنة لا يفهمها مدقق الاستعارة. سنقدم دالات غير آمنة، ثم سنلقي نظرةً على مثال على التجريد الآمن الذي يستعمل شيفرةً غير آمنة.

19.1.3 استدعاء تابع أو دالة غير آمنين

النوع الثاني من العمليات التي يمكنك إجراؤها في كتلة غير آمنة هو استدعاء دالات غير آمنة، إذ تبدو الدالات والتوابع غير الآمنة تمامًا مثل الدالات والتوابع العادية ولكنها تحتوي على unsafe إضافية قبل بقية التعريف. تشير الكلمة المفتاحية unsafe في هذا السياق إلى أن الدالة لها متطلبات نحتاج إلى دعمها عند استدعاء هذه الدالة لأن رست لا يمكن أن تضمن أننا استوفينا هذه المتطلبات؛ فمن خلال استدعاء دالة غير آمنة داخل كتلة unsafe، فإننا نقول إننا قد قرأنا توثيق هذه الدالة ونتحمل مسؤولية دعم مواصفات الدالة هذه.

فيما يلي دالة غير آمنة تدعى dangerous لا تنفّذ أي شيء داخلها:

```
    unsafe fn dangerous() {}

    unsafe {
        dangerous();
    }
```

يجب علينا استدعاء دالة dangerous داخل كتلة unsafe منفصلة، وإذا حاولنا استدعاء dangerous دون unsafe سنحصل على خطأ:

```
$ cargo run
   Compiling unsafe-example v0.1.0 (file:///projects/unsafe-example)
error[E0133]: call to unsafe function is unsafe and requires unsafe
function or block
 --> src/main.rs:4:5
  |
|     dangerous();
  |     ^^^^^^^^^^ call to unsafe function
  |
  = note: consult the function's documentation for information on how
to avoid undefined behavior

For more information about this error, try `rustc --explain E0133`.
error: could not compile `unsafe-example` due to previous error
```

نؤكد لرست مع الكتلة unsafe أننا قرأنا توثيق الدالة ونفهم كيفية استخدامها صحيحًا وتحققنا من أننا نفي بمواصفات الدالة.

يعدّ محتوى الدالات غير الآمنة بمثابة كتل unsafe، لذا لا نحتاج إلى إضافة كتلة unsafe أخرى لأداء عمليات أخرى غير آمنة ضمن دالة غير آمنة.

١. إنشاء تجريد آمن على شيفرة غير آمنة

لا يعني احتواء الدالة على شيفرة غير آمنة أننا بحاجة إلى وضع علامة بأن كامل الدالة غير آمنة، إذ يُعد تغليف الشيفرات البرمجية غير الآمنة في دالة آمنة تجريدًا شائعًا. وكمثال دعنا ننظر إلى الدالة split_at_mut الموجودة في المكتبة القياسية التي تتطلب بعض الشيفرات البرمجية غير الآمنة. سنكتشف كيف يمكننا تنفيذها. يُعرَّف هذا التابع الآمن على الشرائح المتغيّرة، فهو يأخذ شريحةً واحدة ويحوّلها لشريحتين عن طريق تقسيم الشريحة في الدليل المعطى مثل وسيط. توضح الشيفرة 4 كيفية استخدام split_at_mut.

```
    let mut v = vec![1, 2, 3, 4, 5, 6];

    let r = &mut v[..];

    let (a, b) = r.split_at_mut(3);
```

```
    assert_eq!(a, &mut [1, 2, 3]);
    assert_eq!(b, &mut [4, 5, 6]);
```

[الشيفرة 4: استعمال الدالة الآمنة `split_at_mut`]

لا يمكننا تنفيذ هذه الدالة باستعمال رست الآمنة فقط، وقد تبدو المحاولة السابقة مثل الشيفرة 5 الـتي لن تصرف. سننفّذ `split_at_mut` مثل دالة للتبسيط بدلًا من تابع لشرائح قيم `i32` فقط بدلًا من النوع العام `T`.

```
fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut
[i32]) {
    let len = values.len();

    assert!(mid <= len);

    (&mut values[..mid], &mut values[mid..])
}
```

[الشيفرة 5: محاولة تنفيذ `split_at_mut` فقط باستعمال رست الآمنة]

تحصل هذه الدالة أولًا على الطول الكلي للشريحة، ثم تؤكد أن الدليل المعطى على أنه معامل موجود داخـل الشريحة عن طريق التحقق مما إذا كان أقل أو يساوي الطول. يعني هذا التأكيد أنه إذا مررنا دليلًا أكبر من الطـول لتقسيم الشريحة عنده، ستهلع الدالة قبل أن تحاول استعمال هذا الدليل.

نعيد بعد ذلك شريحتين متغيّرتين في الصف، واحدة من بداية الشريحة الأصـلية إلى الـدليل `mid` والأخـرى من `mid` إلى نهاية الشريحة.

عندما نحاول تصريف الشيفرة البرمجية في الشيفرة 5 سنحصل على خطأ.

```
$ cargo run
   Compiling unsafe-example v0.1.0 (file:///projects/unsafe-example)
error[E0499]: cannot borrow `*values` as mutable more than once at a
time
 --> src/main.rs:6:31
  |
| fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut
[i32]) {
  |                         - let's call the lifetime of this
reference `'1`
...
|     (&mut values[..mid], &mut values[mid..])
```

```
  |     --------------------------^^^^^^--------
  |     |     |                   |
  |     |     |                   second mutable borrow occurs here
  |     |     first mutable borrow occurs here
  |     returning this value requires that `*values` is borrowed for
`'1`

For more information about this error, try `rustc --explain E0499`.
error: could not compile `unsafe-example` due to previous error
```

لا يستطيع مدقق الاستعارة في رست أن يفهم أننا نستعير أجزاءً مختلفةً من الشريحة، إذ أنه يعرف فقط أننا نستعير من الشريحة نفسها مرتين. تعد عملية استعارة أجزاء مختلفة من الشريحة أمرًا مقبولًا بصورةٍ أساسية لأن الشريحتين غير متداخلتين لكن رست ليست ذكية بما يكفي لمعرفة ذلك. عندما نعلم أن الشيفرة على ما يرام لكن رست لا تعلم ذلك فهذا يعني أن الوقت قد حان لاستخدام شيفرة غير آمنة.

توضح الشيفرة 6 كيفية استخدام كتلة `unsafe` ومؤشر خام وبعض الاستدعاءات للدالات غير الآمنة لجعل تنفيذ `split_at_mut` يعمل.

```
use std::slice;

fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut
[i32]) {
    let len = values.len();
    let ptr = values.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (
            slice::from_raw_parts_mut(ptr, mid),
            slice::from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    }
}
```

[الشيفرة 6: استعمال شيفرة غير آمنة في تنفيذ دالة `split_at_mut`]

تـذكر سـابقًا من قسـم "نـوع الشـريحة" في الفصـل 4 أن الشـرائح هي مؤشـرات لبعض البيانـات وطـول الشريحة.

نستعمل تابع `len` للحصول على طول الشريحة وتابع `as_mut_ptr` للوصول إلى المؤشر الخـام للشـريحة، وفي هـذه الحالـة نظـرًا لأن لـدينا شـريحة متغيّـرة لقيم `i32` فـإن `as_mut_ptr` تُعيـد مؤشـرًا خامًـا من النـوع `*mut i32` وهو الذي خزّناه في المتغير `ptr`.

نحافظ على التأكيد على أن الدليل `mid` يقع داخل الشريحة، ثم نبدأ بكتابة الشيفرة غـير الآمنـة: تأخـذ الدالـة `slice::from_raw_parts_mut` مؤشرًا خامًا وطولًا وتنشئ شريحة. نستخدم هذه الدالة لإنشاء شريحة تبدأ من `ptr` وتكون عناصرها بطول `mid`، ثم نستدعي التابع `add` على `ptr` مع الوسـيط `mid` للحصـول على مؤشـر خـام يبـدأ من `mid` وننشـئ شـريحةً باسـتخدام هـذا المؤشـر والعـدد المتبقي من العناصـر بعـد `mid` ليكـون طول الشريحة.

الدالة `slice::from_raw_parts_mut` غير آمنـة لأنهـا تأخـذ مؤشـرًا خامًـا ويجب أن تثـق في أن هـذا المؤشر صالح، كما يعد التابع `add` في المؤشرات الخام غير آمن أيضًا لأنه يجب أن تثق في أن موقع الإزاحة هـو أيضًا مؤشر صالح، لذلك كان علينا وضع كتلة `unsafe` حول اسـتدعاءات `slice::from_raw_parts_mut` و `add`حتى نتمكن من استدعائها. من خلال النظر إلى الشيفرة وإضافة التأكيـد على أن `mid` يجب أن يكـون أقـل من أو يسـاوي `len` يمكننـا أن نقـول أن جميـع المؤشـرات الخـام المسـتخدمة داخـل الكتلـة `unsafe` سـتكون مؤشرات صالحة للبيانات داخل الشريحة، وهذا استخدام مقبول ومناسب للكتلة `unsafe`.

لاحظ أننـا لسـنا بحاجـة إلى وضـع علامـة على الدالـة `split_at_mut` الناتجـة بكونهـا `unsafe`، ويمكننـا استدعاء هذه الدالة من رست الآمنة. أنشأنا تجريدًا آمنًا للشيفرة غير الآمنة من خلال تنفيذ الدالة الـتي تسـتعمل شـيفرة `unsafe` بطريقـة آمنـة لأنهـا تُنشـئ مؤشـرات صـالحة فقـط من البيانـات الـتي يمكن لهـذه الدالة الوصول إليها.

في المقابل، من المحتمل أن يتعطـل اسـتخدام `slice::from_raw_parts_mut` في الشـيفرة 7 عنـد استعمال الشريحة. تأخذ هذه الشيفرة موقعًا عشوائيًا للذاكرة وتنشئ شريحة يبلغ طولها 10000 عنصر.

```
    use std::slice;

    let address = 0x01234usize;
    let r = address as *mut i32;

    let values: &[i32] = unsafe { slice::from_raw_parts_mut(r, 10000)
};
```

[الشيفرة 7: إنشاء شريحة من مكان ذاكرة عشوائي]

لا نمتلك الذاكرة في هذا الموقع العشوائي وليس هناك ما يضمن أن الشريحة التي تنشئها هذه الشيفرة تحتوي على قيم i32 صالحة، كما تؤدي محاولة استخدام values كما لو كانت شريحة صالحة إلى سلوك غير معرّف.

ب. استعمال دوال extern لاستدعاء شيفرة خارجية

قد تحتاج شيفرة رست الخاصة بك أحيانًا إلى التفاعل مع شيفرة مكتوبة بلغة برمجية أخرى، لهذا تحتوي رست على الكلمة المفتاحية extern التي تسهل إنشاء واستخدام واجهة الدالة الخارجية Foreign Function interface -أو اختصارًا FFI، وهي طريقة للغة البرمجة لتعريف الدوال وتمكين لغة برمجة (خارجية) مختلفة لاستدعاء هذه الدوال.

توضح الشيفرة 8 التكامل مع دالة abs من مكتبة سي القياسية، وغالبًا ما تكون الدوال المعلنة داخل الكتل extern غير آمنة لاستدعائها من شيفرة رست، والسبب هو أن اللغات الأخرى لا تفرض قواعد وضمانات رست ولا يمكن لرست التحقق منها لذلك تقع مسؤولية ضمان سلامتها على عاتق المبرمج.

اسم الملف: src/main.rs

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

[الشيفرة 8: التصريح عن الدالة extern واستدعاؤها في لغة أخرى]

نُدرج ضمن كتلة extern ''C'' أسماء وبصمات signature الدوال الخارجية من لغة أخرى نريد استدعائها، إذ يحدد الجزء "C" واجهة التطبيق الثنائية application binary interface -أو اختصارًا ABI- التي تستخدمها الدالة الخارجية. تعرّف واجهة التطبيق الثنائية ABI كيفية استدعاء الدالة على مستوى التجميع assembly، وتعد واجهة التطبيق الثنائية للغة ''C'' الأكثر شيوعًا وتتبع واجهة التطبيق الثنائية للغة البرمجة سي.

ج. استدعاء دوال رست من لغات أخرى

يمكننا أيضًا استخدام `extern` لإنشاء واجهة تسمح للغات الأخرى باستدعاء دوال رست، وبدلًا من إنشاء كتلة `extern` كاملة نضيف الكلمة المفتاحية `extern` ونحدد واجهة التطبيق الثنائية ABI لاستخدامها قبل الكلمة المفتاحية `fn` للدالة ذات الصلة. نحتاج أيضًا إلى إضافة تعليق توضيحي `#[no_mangle]` لإخبار مصرّف رست بعدم تشويه mangle اسم هذه الدالة؛ إذ يحدث التشويه عندما يغير المصرف الاسم الذي أعطيناه للدالة لاسم مختلف يحتوي على مزيد من المعلومات لأجزاء أخرى من عملية التصريف لاستهلاكها ولكنها أقل قابلية للقراءة من قبل الإنسان. يشكّل كل مصرف لغة برمجية الأسماء على نحوٍ مختلف قليلًا، لذلك لكي تكون دالة رست قابلة للتسمية من اللغات الأخرى، يجب علينا تعطيل تشويه الاسم في مصرف رست.

في المثال التالي نجعل دالة `call_from_c` قابلة للوصول من شيفرة مكتوبة بلغة سي بعد تصريفها في مكتبة مشتركة وربطها من لغة سي:

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
```

لا يتطلب استعمال `extern` الكتلة `unsafe`.

19.1.4 الوصول أو تعديل متغير ساكن قابل للتغيير mutable

لم نتحدث بعد عن المتغيرات العامة global التي تدعمها رست، إلا أنها قد تسبب مشكلةً مع قواعد ملكية رست. إذا كان هناك خيطان thread يصلان إلى نفس المتغير العام المتغيّر فقد يتسبب ذلك في حدوث سباق بيانات data race.

تسمى المتغيرات العامة في رست بالمتغيرات الساكنة، وتظهر الشيفرة 9 مثالًا للتصريح عن متغير ساكن واستخدامه مع شريحة سلسلة نصية مثل قيمة.

اسم الملف: src/main.rs

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("name is: {}", HELLO_WORLD);
}
```

[الشيفرة 9: تعريف واستعمال متغير ساكن ثابت]

تشبه المتغيرات الساكنة الثوابت التي ناقشناها سابقًا في الفصل 3. أسماء المتغيرات الثابتة موجـودة في SCREAMING_SNAKE_CASE اصطلاحًا، ويمكن للمتغيرات الساكنة فقط تخزين المراجـع مـع دورة حيـاة سـاكنة 'static، ما يعني أن مصرف رست يمكنه معرفـة دورة الحيـاة الخاصـة دون الحاجـة لتحديـده صـراحةً، ويعـد الوصول إلى متغير ساكن آمنًا.

الفرق الدقيق بين الثوابت والمتغيرات الساكنة الثابتة immutable هو أن القيم في متغير ساكن لها عنوان ثابت في الذاكرة، كما سيؤدي استعمال القيمة دائمًا إلى الوصول إلى البيانـات ذاتهـا. من ناحيـة أخـرى، يُسـمح للثوابت بتكرار بياناتها في أي وقت تُستخدم، الفـرق الآخـر هـو أن المتغـيرات السـاكنة يمكن أن تكـون متغيّـرة. الوصول إلى المتغيرات الساكنة القابلة للتغيير وتعديلها غير آمن. توضح الشيفرة 10 كيفيـة التصـريح عن متغـير ساكن قابل للتغيير يسمى COUNTER والوصول إليه وتعديله.

اسم الملف: src/main.rs

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```

[الشيفرة 10: القراءة من أو الكتابة على متغير ساكن قابل للتغيير غير آمن]

كما هو الحال مع المتغيرات العادية نحدد قابلية التغيير باستخدام الكلمـة المفتاحيـة mut، ويجب أن تكـون أي شيفرة تقرأ أو تكتب من COUNTER ضمن كتلة unsafe. تُصرَّف هذه الشـيفرة وتطبـع COUNTER: 3 كمـا نتوقـع لأنهـا تسـتخدم خيطًـا واحـدًا، إذ من المحتمـل أن يـؤدي وصـول خيـوط متعـددة إلى COUNTER إلى سباق البيانات.

يصعب ضمان عدم وجود سباقات بيانات مع البيانات المتغيّرة التي يمكن الوصول إليها بصورةٍ عامة، ولهذا السبب تنظر رست إلى المتغيرات الساكنة المتغيّرة بكونها غير آمنة. يُفضّل استخدام تقنيات التزامن والمؤشرات الذكية ذات الخيوط الآمنة التي ناقشناها سابقًا في الفصل 16 حيثما أمكن حتى يتحقق المصرف من أن البيانات التي يجري الوصول إليها من الخيوط المختلفة آمنة.

19.1.5 تنفيذ سمة غير آمنة

يمكننا استعمال `unsafe` لتطبيق سمة غير آمنة؛ وتكون السمة غير آمنة عندما يحتوي أحد توابعها على الأقل على بعض اللامتغايرات invariant التي لا يستطيع المصرف التحقق منها. نصرّح بأن السمة `unsafe` عن طريق إضافة الكلمة المفتاحية `unsafe` قبل `trait` ووضع علامة على أن تنفيذ السمة `unsafe` أيضًا كما هو موضح في الشيفرة 11.

```
unsafe trait Foo {
    // methods go here
}

unsafe impl Foo for i32 {
    // method implementations go here
}

fn main() {}
```

[الشيفرة 11: تعريف وتنفيذ سمة غير آمنة]

نعد بأننا سنلتزم باللامتغايرات التي لا يمكن للمصرف التحقق منها باستخدام `unsafe impl`.

على سبيل المثال، تذكر سمات العلامة `Sync` و `Send` التي ناقشناها سابقًا في قسم "التزامن الموسع مع السمة Sync والسمة Send" في الفصل 16، يطبّق المصرف هذه السمات تلقائيًا إذا كانت أنواعنا تتكون كاملًا من النوعين `Sync` و `Send`. إذا طبقنا نوعًا يحتوي على نوع ليس `Sync` و `Send` مثل المؤشرات الخام ونريد وضع علامة على هذا النوع على أنه `Sync` و `Send` فيجب علينا استخدام `unsafe`. لا تستطيع رست التحقق من أن النوع الخاص بنا يدعم الضمانات التي يمكن إرسالها بأمان عبر الخيوط أو الوصول إليها من خيوط متعددة، لذلك نحتاج إلى إجراء تلك الفحوصات يدويًا والإشارة إلى ذلك باستخدام `unsafe`.

19.1.6 الوصول لحقول الاتحاد Union

الإجراء الأخير الذي يعمل فقط مع `unsafe` هو الوصول إلى حقول الاتحاد؛ ويعد `union` مشابهًا للبنية `struct` ولكن يُستخدم فيه حقل مصرح واحد فقط في نسخة معينة في وقت واحد، وتُستخدم الاتحادات

بصورةٍ أساسية للتفاعل مع الاتحادات في شيفرة لغة سي. يعد الوصول إلى حقول الاتحاد غير آمن لأن رست لا يمكنها ضمان نوع البيانات المخزنة حاليًا في نسخة الاتحاد. يمكنك معرفة المزيد عن الاتحادات في توثيق رست Rust Reference.

19.1.7 متى تستعمل شيفرة غير آمنة؟

لا يُعد استعمال `unsafe` لفعل أحد الأفعال الخمسة (القوى الخارقة) التي ناقشناها للتو أمرًا خاطئًا أو غير مرغوب إلا أنه من الأصعب الحصول على شيفرة `unsafe` صحيحة لأن المصرف لا يستطيع المساعدة بدعم آمان الذاكرة. عندما يكون لديك سببًا لاستخدام شيفرة `unsafe` تستطيع ذلك، ويسهّل وجود تعليق توضيحي `unsafe` صريح تعقب مصدر المشكلات عند حدوثها.

19.2 السمات Trait المتقدمة

غطينا مفهوم السمات سابقًا في الفصل 10، إلا أننا لم نناقش التفاصيل الأكثر تقدمًا. سنخوض الآن بالتفاصيل الجوهرية بعد أن تعملت المزيد عن لغة رست حتى الآن.

19.2.1 تحديد أنواع الموضع المؤقت في تعريفات السمات مع الأنواع المرتبطة

تصل الأنواع المرتبطة associated types نوع موضع مؤقت placeholder بسمة بحيث يمكن لتعريفات تابع السمة استخدام أنواع المواضع المؤقتة هذه في بصماتها signature، وسيحدد منفّذ السمة النوع الحقيقي الذي سيُستخدم بدلًا من نوع الموضع المؤقت للتنفيذ المعين. يمكننا بهذه الطريقة تحديد سمة تستخدم بعض الأنواع دون الحاجة إلى معرفة ماهية هذه الأنواع تحديدًا حتى تُطبَّق السمة.

وصفنا معظم الميزات المتقدمة في هذا الفصل على أنها نادرًا ما تكون مطلوبة. توجد الأنواع المرتبطة في مكان ما في الوسط، إذ تُستخدَم نادرًا أكثر من الميزات الموضحة سابقًا في بقية الكتاب ولكنها أكثر شيوعًا من العديد من الميزات المتقدمة الأخرى التي نوقشت في هذا الفصل.

أحد الأمثلة على سمة ذات نوع مرتبط هي سمة `Iterator` التي توفرها المكتبة القياسية. يُطلق على النوع المرتبط اسم `Item` ويرمز إلى نوع القيم التي يمرّ عليها النوع الذي ينفّذ سمة `Iterator`. تُعرّف سمة `Iterator` كما هو موضح في الشيفرة 12.

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
}
```

[الشيفرة 12: تعريف سمة `Iterator` التي لديها نوع مرتبط `Item`]

النـــوع `Item` هـــو موضـــع مـــؤقت، ويوضـــح تعريـــف تـــابع `next` أنـــه ســـيعيد قيمًـــا من النـــوع `Option<Self::Item>`. سيحدد منفّذ سمة `Iterator` النوع الحقيقي للنـوع `Item` وسـيُعيد التـابع `next` النوع `Option` الذي يحتوي على قيمة من هذا النوع الحقيقي.

قد تبدو الأنواع المرتبطة مثل مفهوم مشابه للأنواع المعممـة من حيث أن الأخـير يسـمح لنـا بتعريـف دالـة دون تحديد الأنواع التي يمكنها التعامل معها، ولفحص الاختلاف بين المفهومين، سنلقي نظرةً على تنفيذ سـمة `Iterator` على نوع يسمى `Counter` الذي يحدد نوع `Item` هو `u32`:

اسم الملف: src/lib.rs

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        // --snip--
```

تبدو هذه الصيغة مشابهة لتلك الموجـودة في الأنـواع المعمّمـة generic، فلمـاذا لا نكتفي بتعريـف سـمة `Iterator` باستخدام الأنواع المعممة كما هو موضح في الشيفرة 13؟

```
pub trait Iterator<T> {
    fn next(&mut self) -> Option<T>;
}
```

[الشيفرة 13: تعريف افتراضي لسمة `Iterator` باستخدام الأنواع المعممة]

الفرق هو أنه علينا أن نوضح الأنواع في كل تنفيذ عند استخدام الأنواع المعمّمـة كمـا في الشـيفرة 13، لأنـه يمكننا أيضًا تنفيذ `Iterator<String> for Counter` أو أي نوع آخر، فقد يكون لدينا تنفيذات متعـددة للسمة `Iterator` من أجل `Counter`. بعبارة أخـرى: عنـدما تحتـوي السـمة على معامـل معمّم، يمكن تنفيـذه لنوع ما عدة مرات مع تغيير الأنواع الحقيقية لمعاملات النوع المعمم في كل مرة، وعندما نستخدم التـابع `next` على `Counter` سيتوجب علينـا توفـير التعليقـات التوضـيحية من النـوع للإشـارة إلى تنفيـذ `Iterator` الـذي نريد استخدامه.

لا نحتاج مع الأنواع المرتبطة إلى إضافة تعليقات توضيحية للأنواع، لأننـا لا نسـتطيع تنفيـذ سـمة على نـوع عدة مرات. يمكننا فقط اختيار نوع `Item` مرةً واحدةً في الشيفرة 12 مع التعريف الذي يستخدم الأنواع المرتبطـة لأنه لا يمكن أن يكون هناك سوى `impl Iterator for Counter` واحد. لا يتعين علينا تحديد أننـا نريـد مكررًا لقيم `u32` في كل مكان نستدعي `next` على `Counter`.

تصبح الأنواع المرتبطة أيضًا جزءًا من عقد contract السمة، إذ يجب أن يوفر منفّذو السمة نوعًا لملء الموضع المؤقت للنوع المرتبط. تمتلك الأنواع المرتبطة غالبًا اسمًا يصف كيفية استخدام النوع، كما يُعد توثيق النوع المرتبط في توثيق الواجهة البرمجية ممارسةً جيدة.

19.2.2 معاملات النوع المعمم الافتراضي وزيادة تحميل العامل

عندما نستخدم معاملات النوع المعمم يمكننا تحديد نوع حقيقي افتراضي للنوع المعمم، وهذا يلغي الحاجة إلى منفّذي السمة لتحديد نوع حقيقي إذا كان النوع الافتراضي يعمل. يمكنك تحديد نوع افتراضي عند التصريح عن نوع عام باستخدام الصيغة `<PlaceholderType=ConcreteType>`.

من الأمثلة الرائعة على الموقف الذي تكون فيه هذه التقنية مفيدة هو زيادة تحميل العامل operator overloading، إذ يمكنك تخصيص سلوك عامل (مثل `+`) في مواقف معينة.

لا تسمح لك رست بإنشاء عواملك الخاصة أو زيادة تحميل العوامل العشوائية، ولكن يمكنك زيادة تحميل العمليات والسمات المقابلة المدرجة في `std::ops` من خلال تنفيذ السمات المرتبطة بالعامل. على سبيل المثال في الشيفرة 14 زدنا تحميل العامل `+` لإضافة نسختين `Point` معًا عن طريق تنفيذ سمة `Add` على بنية `Point`.

اسم الملف: src/main.rs

```
use std::ops::Add;

#[derive(Debug, Copy, Clone, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
```

```
}

fn main() {
    assert_eq!(
        Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
        Point { x: 3, y: 3 }
    );
}
```

[الشيفرة 14: تنفيذ سمة Add لزيادة تحميل العامل + لنسخ Point]

يضيف التابع add قيم x لنسختَي Point وقيم y لنسختَي Point لإنشاء Point جديدة. للسمة Add نوع مرتبط يسمى Output الذي يحدد النوع الذي يُعاد من التابع add.

النوع المعمم الافتراضي في هذه الشيفرة موجودٌ ضمن سمة Add. إليك تعريفه:

```
trait Add<Rhs=Self> {
    type Output;

    fn add(self, rhs: Rhs) -> Self::Output;
}
```

يجب أن تبدو هذه الشيفرة مألوفة عمومًا: سمة ذات تابع واحد ونوع مرتبط بها. الجزء الجديد هو Rhs=Self، إذ يُطلق على الصيغة هذه معاملات النوع الافتراضية default type parameters. يعرّف معامل النوع المعمم Rhs (اختصار للجانب الأيمن right hand size) نوع المعامل rhs في تابع add. إذا لم نحدد نوعًا حقيقيًا للقيمة Rhs عند تنفيذ سمة Add، سيُعيّن نوع Rhs افتراضيًا إلى Self الذي سيكون النوع الذي ننفّذ السمة Add عليه.

عندما نفّذنا Add على Point استخدمنا الإعداد الافتراضي للنوع Rhs لأننا أردنا إضافة نسختين Point. لنلقي نظرةً على مثال لتنفيذ سمة Add، إذ نريد تخصيص نوع Rhs بدلًا من الاستخدام الافتراضي.

لدينا الهيكلان Millimeters و Meters اللذان يحملان قيمًا في وحدات مختلفة، يُعرف هذا الغلاف الرقيق لنوع موجود في هيكل آخر باسم نمط النوع الجديد newtype pattern الذي نصفه بمزيد من التفصيل لاحقًا في قسم "استخدام نمط النوع الجديد لتنفيذ السمات الخارجية على الأنواع الخارجية". نريد أن نضيف القيم بالمليمترات إلى القيم بالأمتار وأن نجعل تنفيذ Add يجري التحويل صحيحًا. يمكننا تنفيذ Add للنوع Millimeters مع Meters مثل Rhs كما هو موضح في الشيفرة 15.

اسم الملف: src/lib.rs

```
use std::ops::Add;

struct Millimeters(u32);
struct Meters(u32);

impl Add<Meters> for Millimeters {
    type Output = Millimeters;

    fn add(self, other: Meters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
}
```

[الشيفرة 15: تنفيذ سمة `Add` على `Millimeters` لإضافة `Millimeters` للنوع `Meters`]

لإضافة `Millimeters` و `Meters` نحدد `impl Add<Meters>` لتعيين قيمة محدد نـوع `Rhs` بـدلًا من استخدام الافتراضي `Self`.

ستستخدم معاملات النوع الافتراضية بطريقتين رئيسيتين:

- لتوسيع نوع دون تعطيل الشيفرة الموجودة.
- للسماح بالتخصيص في حالات معينة لن يحتاجها معظم المستخدمين.

تعد سمة المكتبة القياسية `Add` مثالًا على الغرض الثـاني: ستضـيف عـادةً نـوعين متشـابهين، بينمـا تـوفّر خاصية `Add` القدرة على التخصيص بعد ذلك. يعني استخدام معامل النوع الافتراضي في تعريف سمة `Add` أنك لست مضطرًا لتحديد المعامل الإضافي في معظم الأوقات. بعبـارة أخـرى ليسـت هنـاك حاجـة إلى القليـل من التنفيذ المعياري، وهذا ما يسهل استخدام السمة.

ا. صيغة مؤهلة كليا للتوضيح باستدعاء التوابع التي تحمل الاسم ذاته

لا شيء في رست يمنع سمةً ما من أن يكون لها تابع يحمل اسم تابع السمة الأخـرى ذاتهـا، كمـا لا تمنعـك رست من تنفيذ كلتا السمتين على نوع واحد، ومن الممكن أيضًا تنفيذ تابع مباشرةً على النوع الذي يحمـل اسـم التوابع نفسه من السمات.

عند استدعاء التوابع التي تحمل الاسم نفسه، ستحتاج إلى إخبار رسـت بالتـابع الـذي تريـد اسـتخدامه. ألـقِ نظرةً على الشيفرة 16، إذ عرّفنا سمتين `Pilot` و `Wizard` ولكل منهما تابع يسمى `fly`، ثم نفّذنا كلتا السمتين على نوع `Human` لديه فعلًا تابع يسمى `fly` منفّذ عليه، وكل تابع `fly` يفعل شيئًا مختلفًا.

اسم الملف: src/main.rs

```
trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain speaking.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("Up!");
    }
}

impl Human {
    fn fly(&self) {
        println!("*waving arms furiously*");
    }
}
```

[الشيفرة 16: تعريف سمتين بحيث تملكان تابع `fly` وتنفيذهما على النوع `Human` وتنفيذ تابع `fly` على `Human` مباشرةً]

عندما نستدعي `fly` على نسخة `Human` يتخلف المصرف عن استدعاء التابع الذي يُنفّذ مباشـرةً على النـوع كما هو موضح في الشيفرة 17.

اسم الملف: src/main.rs

```
fn main() {
    let person = Human;
    person.fly();
}
```

[الشيفرة 17: استدعاء `fly` على نسخة `Human`]

سيؤدي تنفيذ هذه الشيفرة إلى طباعة `*waving arms furiously*` مما يدل على أن رست استدعت تابع `fly` المنفّذ على `Human` مباشرةً.

نحتاج إلى استخدام صيغة أكثر وضوحًا عند استدعاء توابع `fly` من سمة `Pilot` أو `Wizard` لتحديد تابع `fly` الذي نعنيه، وتوضّح الشيفرة 18 هذه الصيغة.

اسم الملف: src/main.rs

```
fn main() {
    let person = Human;
    Pilot::fly(&person);
    Wizard::fly(&person);
    person.fly();
}
```

[الشيفرة 18: تحديد تابع السمة `fly` التي نريد استدعاءه]

يوضح تحديد اسم السمة قبل اسم التابع لرست أي تنفيذ للتابع `fly` نريد استدعاءه. يمكننا أيضًا كتابة `Human::fly(&person)` وهو ما يعادل `person.fly()` الذي استخدمناه في الشيفرة 18، ولكن هذا أطول قليلًا للكتابة إذا لم نكن بحاجة إلى توضيح.

يؤدي تنفيذ هذه الشيفرة إلى طباعة التالي:

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.46s
     Running `target/debug/traits-example`
This is your captain speaking.
Up!
*waving arms furiously*
```

إذا كان لدينا نوعان ينفذ كلاهما سمةً واحدةً، يمكن أن تكتشف رست أي تنفيذ للسمة يجب استخدامه بناءً على نوع `self` نظرًا لأن تابع `fly` يأخذ معامل `self`، ومع ذلك فإن الدوال functions المرتبطة التي ليست

بتوابع لا تحتوي على معامل `self`. عندما تكون هناك أنواع أو سمات متعددة تحدد دوالًا غير تابعية non-method بنفس اسم الدالة، لا تعرف رست دائمًا النوع الذي تقصده ما لم تستخدم صيغة مؤهلة كليًا fully qualified syntax. على سبيل المثال في الشيفرة 19 أنشأنا سمة لمأوى للحيوانات يسمّي جميع الجراء puppies الصغار Spot. ننشئ سمة `Animal` بدالة غير تابعية مرتبطة `baby_name`، تُنفّذ `Animal` لهيكل `Dog` الذي نوفّر عليه أيضًا دالةً غير تابعية مرتبطة `baby_name` مباشرةً.

اسم الملف: src/main.rs

```
trait Animal {
    fn baby_name() -> String;
}

struct Dog;

impl Dog {
    fn baby_name() -> String {
        String::from("Spot")
    }
}

impl Animal for Dog {
    fn baby_name() -> String {
        String::from("puppy")
    }
}

fn main() {
    println!("A baby dog is called a {}", Dog::baby_name());
}
```

[الشيفرة 19: سمة لها دالة مرتبطة ونوع له دالة مرتبطة بالاسم ذاته الذي ينفّذ السمة أيضًا]

ننفّذ الشيفرة الخاص بتسمية كل الجراء في الدالة المرتبطة `baby_name` والمُعرّفة في `Dog`. ينفّذ النوع `Dog` أيضًا سمة `Animal` التي تصف الخصائص التي تمتلكها جميع الحيوانات. يُطلق على أطفال الكلاب اسم الجراء ويُعبّر عن ذلك في تنفيذ سمة `Animal` على `Dog` في الدالة المرتبطة `baby_name` بالسمة `Animal`

نستدعي في الدالة `main` الدالة `Dog::baby_name` التي تستدعي الدالة المرتبطة المعرفة في `Dog` مباشرةً. تطبع هذه الشيفرة ما يلي:

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.54s
     Running `target/debug/traits-example`
A baby dog is called a Spot
```

لم نكن نتوقع هذه النتيجة، إذ نريد استدعاء دالة `baby_name` التي تعد جزءًا من سمة `Animal` التي نفّذناها على `Dog` حتى تطبع الشيفرة `A baby dog is called a puppy`. لا تساعد تقنية تحديد اسم السمة التي استخدمناها في الشيفرة 18 هنا، وإذا غيرنا `main` للشيفرة لما هو موجود في الشيفرة 20، سنحصل على خطأ عند التصريف.

اسم الملف: src/main.rs

```
fn main() {
    println!("A baby dog is called a {}", Animal::baby_name());
}
```

[الشيفرة 20: محاولة استدعاء الدالة `baby_name` من السمة `Animal` دون معرفة رست بأي تنفيذ ينبغي استخدامه]

لا تستطيع رست معرفة أي تنفيذ نريده للقيمة `Animal::baby_name` لأن `Animal::baby_name` لا تحتوي على معامل `self` ولأنه يمكن أن تكون هناك أنواع أخرى تنفّذ سمة `Animal`. سنحصل على خطأ المصرف هذا:

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
error[E0790]: cannot call associated function on trait without
specifying the corresponding `impl` type
  --> src/main.rs:20:43
   |
|     fn baby_name() -> String;
   |     ------------------------ `Animal::baby_name` defined here
...
|     println!("A baby dog is called a {}", Animal::baby_name());
   |                                           ^^^^^^^^^^^^^^^^
cannot call associated function of trait
   |
help: use the fully-qualified path to the only available
implementation
```

```
   |
|     println!("A baby dog is called a {}", <Dog as
Animal>::baby_name());
   |                                          +++++++       +


For more information about this error, try `rustc --explain E0790`.
error: could not compile `traits-example` due to previous error
```

لإزالة الغموض وإخبار رست أننا نريد استخدام تنفيذ `Animal` للقيمة `Dog` بدلًا من استخدام `Animal` لبعض الأنواع الأخرى، نحتاج إلى استخدام صيغة مؤهلة كليًا. توضح الشيفرة 21 كيفية استخدام صيغة مؤهلة كليًا.

اسم الملف: src/main.rs

```
fn main() {
    println!("A baby dog is called a {}", <Dog as
Animal>::baby_name());
}
```

[الشيفرة 21: استعمال صيغة مؤهلة كليًا لتحديد أننا نريد استدعاء دالة `baby_name` من السمة `Animal` كما هو منفّذ في `Dog`]

نقدم لرست تعليقًا توضيحيًا للنوع ضمن أقواس مثلثية angle brackets مما يشير إلى أننا نريد استدعاء تابع `baby_name` من سمة `Animal` كما هو منفّذ في `Dog` بالقول إننا نريد معاملة النوع `Dog` مثل النوع `Animal` لاستدعاء الدالة هذه. ستطبع الشيفرة البرمجية الآن ما نريد:

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/traits-example`
A baby dog is called a puppy
```

تعرف الصيغة المؤهلة كليًا عمومًا على النحو التالي:

```
<Type as Trait>::function(receiver_if_method, next_arg, ...);
```

بالنسبة للدوال المرتبطة التي ليست توابع، لن يكون هناك `receiver`، سيكون هناك فقط قائمة من الوسائط الأخرى. يمكنك استخدام صيغة مؤهلة كليًا في كل مكان تستدعي فيه دوالًا أو توابعًا، ومع ذلك يُسمح لك بحذف أي جزء من هذه الصيغة التي يمكن لرست اكتشافها من المعلومات الأخرى في البرنامج. ما عليك

سوى استخدام هذه الصيغة المطولة أكثر في الحالات التي توجد فيها العديد من التنفيذات التي تستخدم الاسم ذاته وتحتاج رست إلى المساعدة في تحديد التنفيذ الذي تريد استدعاءه.

19.2.3 استخدام سمات خارقة supertrait لطلب وظيفة سمة ضمن سمة أخرى

في بعض الأحيان قد تكتب تعريف سمة يعتمد على سمة أخرى: لكي ينفّذ النوع السمة الأولى فأنت تريد أن تطلب هذا النوع لتنفيذ السمة الثانية أيضًا. يمكنك فعل ذلك حتى يتمكن تعريف السمة الخاص بك من الاستفادة من العناصر المرتبطة للسمة الثانية. تسمى السمة التي يعتمد عليها تعريف السمة الخاص بك بالسمة الخارقة لسمتك.

على سبيل المثال لنفترض أننا نريد إنشاء سمة `OutlinePrint` باستخدام تابع `outline_print` الذي سيطبع قيمة معينة منسقة بحيث تكون مؤطرة بعلامات نجمية. بالنظر إلى هيكل `Point` الذي ينفّذ سمة المكتبة القياسية `Display` لتعطي النتيجة `(x, y)`، عندما نستدعي `outline_print` على نسخة `Point` التي تحتوي على 1 للقيمة `x` و 3 للقيمة `y`، يجب أن يطبع ما يلي:

```
**********
*        *
* (1, 3) *
*        *
**********
```

نريد استخدام وظيفة سمة `Display` في تنفيذ طريقة `outline_print`، لذلك نحتاج إلى تحديد أن سمة `OutlinePrint` ستعمل فقط مع الأنواع التي تنفّذ أيضًا `Display` وتوفر الوظائف التي تحتاجها `OutlinePrint`. يمكننا فعل ذلك في تعريف السمة عن طريق تحديد `OutlinePrint: Display`، وتشبه هذه التقنية إضافة سمة مرتبطة بالسمة. تُظهر الشيفرة 22 تنفيذ سمة `OutlinePrint`.

اسم الملف: src/main.rs

```
use std::fmt;

trait OutlinePrint: fmt::Display {
    fn outline_print(&self) {
        let output = self.to_string();
        let len = output.len();
        println!("{}", "*".repeat(len + 4));
        println!("*{}*", " ".repeat(len + 2));
        println!("* {} *", output);
```

```
        println!("*{}*", " ".repeat(len + 2));
        println!("{}", "*".repeat(len + 4));
    }
}
```

[الشيفرة 22: تنفيذ سمة `OutlinePrint` التي تتطلب الوظيفة من `Display`]

بما أننا حددنا أن `OutlinePrint` تتطلب سمة `Display`، يمكننا استخدام دالة `to_string` التي تُنفّذ تلقائيًا لأي نوع ينفّذ `Display`. إذا حاولنا استخدام `to_string` دون إضافة نقطتين وتحديد سمة `Display` بعد اسم السمة، سنحصل على خطأ يقول بأنه لم يُعثر على تابع باسم `to_string` للنوع `&Self` في النطاق الحالي.

لنرى ما يحدث عندما نحاول تنفيذ `OutlinePrint` على نوع لا ينفّذ `Display` مثل هيكل `Point`.

اسم الملف: src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

impl OutlinePrint for Point {}
```

نحصل على خطأ يقول أن `Display` مطلوبة ولكن غير منفّذة:

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
error[E0277]: `Point` doesn't implement `std::fmt::Display`
  --> src/main.rs:20:6
   |
| impl OutlinePrint for Point {}
   |      ^^^^^^^^^^^^ `Point` cannot be formatted with the default
formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for
`Point`
   = note: in format strings you may be able to use `{:?}` (or {:#?}
for pretty-print) instead
note: required by a bound in `OutlinePrint`
```

```
  --> src/main.rs:3:21
   |
| trait OutlinePrint: fmt::Display {
   |                     ^^^^^^^^^^^^ required by this bound in
`OutlinePrint`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `traits-example` due to previous error
```

لإصلاح هذا الأمر ننفّذ `Display` على `Point` ونلبي القيد الذي تتطلبه `OutlinePrint` مثل:

اسم الملف: src/main.rs

```
use std::fmt;

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

سيُصرَّف تنفيذ سمة `OutlinePrint` على `Point` بعدها بنجاح، ويمكننا استدعاء `outline_print` على نسخة `Point` لعرضها ضمن مخطط من العلامات النجمية.

19.2.4 استخدام نمط النوع الجديد لتنفيذ سمات الخارجية على الأنواع الخارجية

ذكرنا سابقًا في الفصل 10 في قسم "تطبيق السمة على نوع" القاعدة الوحيدة التي تنص على أنه لا يُسمح لنا إلا بتنفيذ سمة على نوع ما إذا كانت السمة أو النوع محليين بالنسبة للوحدة المصرفة الخاصة بنا، إلا أنه من الممكن التحايل على هذا القيد باستخدام نمط النوع الجديد Newtype pattern الذي يتضمن إنشاء نوع جديد في هيكل الصف (ناقشنا هياكل الصف سابقًا في قسم "استخدام هياكل الصفوف دون حقول مسماة لإنشاء أنواع مختلفة" من الفصل 5)، إذ سيكون لهيكل الصف حقلًا واحدًا وسيكون هناك غلافًا رفيعًا حول النوع الذي نريد تنفيذ سمة له، ثم يكون نوع الغلاف محليًا بالنسبة للوحدة المصرفة الخاصة بنا ويمكننا تنفيذ السمة على الغلاف. النوع الجديد هو مصطلح ينشأ من لغة البرمجة هاسكل Haskell. لا يوجد تأثير سلبي على وقت التنفيذ لاستخدام هذا النمط ويُستبعد نوع الغلاف في وقت التصريف.

على سبيل المثال، لنفترض أننا نريد تنفيذ `Display` على `Vec<T>` التي تمنعنا القاعدة الوحيدة من فعل ذلك مباشرةً لأن سمة `Display` ونوع `Vec<T>` مُعرّفان خارج الوحدة المصرفة الخاصة بنا. يمكننا عمل هيكل

`Wrapper` يحتوي على نسخة من `Vec<T>` ومن ثم تنفيذ `Display` على `Wrapper` واستخدام قيمة `Vec<T>` كما هو موضح في الشيفرة 23.

اسم الملف: src/main.rs

```
use std::fmt;

struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}

fn main() {
    let w = Wrapper(vec![String::from("hello"),
String::from("world")]);
    println!("w = {}", w);
}
```

[الشيفرة 23: إنشاء نوع `Wrapper` حول `Vec<String>` لتنفيذ `Display`]

تُستخدم `self.0` من قِبل تنفيذ `Display` للوصول إلى `Vec<T>` الداخلي، لأن `Wrapper` هيكل صف و `Vec<T>` هو العنصر في الدليل 0 في الصف. يمكننا بعد ذلك استخدام وظيفة نوع `Display` على `Wrapper`.

الجانب السلبي لاستخدام هذه التقنية هو أن `Wrapper` هو نوع جديد لذلك لا يحتوي على توابع القيمة التي يحملها. سيتوجب علينا تنفيذ جميع توابع `Vec<T>` مباشرةً على `Wrapper`، بحيث تفوض هذه التوابع إلى `self.0`، ليسمح لنا بالتعامل مع `Wrapper` تمامًا مثل `Vec<T>`.

إذا أردنا أن يحتوي النوع الجديد على كل تابع يمتلكه النوع الداخلي، سيكون تنفيذ سمة `Deref` (ناقشناها سابقًا في الفصل 15) على `Wrapper` لإرجاع النوع الداخلي حلًا مناسبًا. إذا كنا لا نريد أن يحتوي نوع `Wrapper` على جميع التوابع من النوع الداخلي -على سبيل المثال لتقييد سلوك نوع `Wrapper`- سيتعين علينا تنفيذ التوابع التي نريدها يدويًا فقط.

نموذج النمط الجديد هذا مفيد أيضًا حتى عندما لا تكون السمات متضمنة. لنغير تركيزنا الآن، ونلقي نظرةً على بعض الطرق المتقدمة للتفاعل مع نظام نوع رست.

19.3 الأنواع المتقدمة

يحتوي نظام نوع رست على بعض الميزات التي ذكرناها سابقًا إلا أننا لم نناقشها بالتفصيل بعد، وسنبدأ بمناقشة الأنواع الجديدة newtypes بصورةٍ عامة والنظر إلى فائدتها كأنواع، ثم ننتقل إلى كتابة الاختصارات وهي ميزة مشابهة للأنواع الجديدة ولكن بدلالات مختلفة قليلًا. سنناقش أيضًا النمط ! والأنواع ذات الحجم الديناميكي dynamically sized types.

19.3.1 استخدام نمط النوع الجديد لأمان النوع والتجريد

> يفترض هذا القسم أنك قرأت قسم 'استخدام نمط النوع الجديد لتنفيذ سمات الخارجية على الأنواع الخارجية' من القسم 19.2.4.

يُعد نمط النوع الجديد مفيدًا أيضًا للمهمات التي تتجاوز تلك التي ناقشناها حتى الآن بما في ذلك الفرض الصارم بعدم الخلط بين القيم وكذلك الإشارة إلى وحدات القيمة. رأيتَ مثالًا على استخدام أنواع جديدة للإشارة إلى الوحدات في الشيفرة 15، تذكر أن هياكل `Millimeters` و `Meters` تغلف قيم `u32` في نوع جديد. إذا كتبنا دالة بمحدد من النوع `Millimeters` فلن نتمكن من تصريف برنامج حاولَ عن طريق الخطأ استدعاء هذه الدالة بقيمة من النوع `Meters` أو `u32` عادي.

يمكننا أيضًا استخدام نمط النوع الجديد للتخلص من بعض تفاصيل التطبيق الخاصة بنوع ما، ويمكن أن يكشف النوع الجديد عن واجهة برمجية عامة API تختلف عن الواجهة البرمجية للنوع الداخلي الخاص.

يمكن أن تخفي الأنواع الجديدة أيضًا التطبيق الداخلي، إذ يمكننا على سبيل المثال يمكننا منح نوع `People` لتغليف `HashMap<i32, String>` الذي يخزن معرف الشخص المرتبط باسمه. تتفاعل الشيفرة التي تستخدم `People` فقط مع الواجهة البرمجية العامة التي نقدمها مثل تابع لإضافة سلسلة اسم إلى مجموعة `People`، ولن تحتاج هذه الشيفرة إلى معرفة أننا نعيِّن معرفًا `i32` للأسماء داخليًا. يعد نمط النوع الجديد طريقةً خفيفةً لتحقيق التغليف لإخفاء تفاصيل التطبيق التي ناقشناها سابقًا في قسم "التغليف وإخفاءه لتفاصيل التنفيذ" من الفصل 17.

19.3.2 إنشاء مرادفات للنوع بواسطة أسماء النوع البديلة

توفّر رست القدرة على التصريح عن اسم بديل للنوع type alias لمنح نوع موجود اسمًا آخر، ونستخدم لذلك الكلمة المفتاحية `type`. يمكننا على سبيل المثال منح الاسم البديل `Kilometers` للنوع `i32` على النحو التالي:

```
type Kilometers = i32;
```

يصبح الاسم المستعار `Kilometers` الآن مرادفًا للنوع `i32` على عكس أنواع `Millimeters` و `Meters` التي أنشأناها في الشيفرة 15، إذ أن `Kilometers` ليست نوعًا جديدًا منفصلًا. ستُعامل القيم ذات النوع `Kilometers` نفس معاملة قيم النوع `i32`:

```
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

println!("x + y = {}", x + y);
```

يمكننا إضافة قيم من كلا النوعين نظرًا لأن `Kilometers` و `i32` من النوع ذاته، كما يمكننا تمرير قيم `Kilometers` إلى الدوال التي تأخذ معاملات `i32`. لن نحصل على مزايا التحقق من النوع التي نحصل عليها من نمط النوع الجديد الذي ناقشناه سابقًا إذا استخدمنا هذا التابع، أي بعبارة أخرى إذا خلطنا قيم `Kilometers` و `i32` في مكان ما فلن يعطينا المصرف خطأ.

حالة الاستخدام الرئيسة لأسماء النوع البديلة هي تقليل التكرار، على سبيل المثال قد يكون لدينا نوع طويل مثل هذا:

```
Box<dyn Fn() + Send + 'static>
```

يمكن أن تكون كتابة هذا النوع المطول في بصمات الدوال ومثل تعليقات توضيحية للنوع في جميع أنحاء الشيفرة أمرًا مملًا وعرضةً للخطأ. تخيل وجود مشروع مليء بالسطر السابق كما توضح الشيفرة 24.

```
    let f: Box<dyn Fn() + Send + 'static> = Box::new(|| println!
("hi"));

    fn takes_long_type(f: Box<dyn Fn() + Send + 'static>) {
        // --snip--
    }

    fn returns_long_type() -> Box<dyn Fn() + Send + 'static> {
        // --snip--
    }
```

[الشيفرة 24: استعمال نوع طويل في أماكن كثيرة]

يجعل الاسم البديل للنوع هذه الشيفرة أكثر قابلية للإدارة عن طريق تقليل التكرار، إذ قدّمنا في الشيفرة 25 اسمًا بديلًا هو `Thunk` للنوع المطول ويمكننا استبدال جميع استخدامات النوع بالاسم البديل الأقصر `Thunk`.

```
type Thunk = Box<dyn Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("hi"));

fn takes_long_type(f: Thunk) {
    // --snip--
}

fn returns_long_type() -> Thunk {
    // --snip--
}
```

[الشيفرة 25: استخدام اسم بديل Thunk لتقليل التكرار]

هذه الشيفرة أسهل في القراءة والكتابة، ويمكن أن يساعد اختيار اسم ذي معنى لاسم بديل للنوع على إيصال نيتك أيضًا، إذ أن thunk هي كلمة لشيفرة تُقيَّم في وقت لاحق لذا فهو اسم مناسب للمغلّف closure الذي يُخزَّن.

تُستخدم الأسماء البديلة للنوع أيضًا كثيرًا مع نوع Result<T, E> لتقليل التكرار، خذ على سبيل المثال وحدة std::io في المكتبة القياسية، إذ غالبًا ما تُعيد عمليات الدخل والخرج النوع Result<T, E> للتعامل مع المواقف التي تفشل فيها العمليات هذه، وتحتوي هذه المكتبة على هيكل std::io::Error الذي يمثل جميع أخطاء الدخل والخرج المحتملة، وتعيد العديد من الدوال في std::io النوع Result<T, E> بحيث تكون قيمة E هي std::io::Error كما هو الأمر بالنسبة للدوال الموجودة في سمة Write:

```
use std::fmt;
use std::io::Error;

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize, Error>;
    fn flush(&mut self) -> Result<(), Error>;

    fn write_all(&mut self, buf: &[u8]) -> Result<(), Error>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<(), Error>;
}
```

تكررت Result<..., Error> كثيرًا، كما احتوت الوحدة std::io على هذا النوع من التصريح:

```
type Result<T> = std::result::Result<T, std::io::Error>;
```

يمكننا استخدام الاسم البديل المؤهل كليًا `std::io::Result<T>` لأن هذا التصريح موجـود في الوحـدة `std::io`، ويعني النوع السابق وجود النـوع `Result<T, E>` مـع ملء `E` بقيمـة `std::io::Error`. تبـدو بصمة السمة `Write` بنهاية المطاف على النحو التالي:

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<()>;
}
```

يساعد الاسم البديل للنوع بطريقتين، فهو يجعل كتابة الشيفرات أسهل ويعطينا واجهةً متسـقة عـبر جميـع أنواع `std::io`، إذ نظرًا لأن الاسم البديل هو `Result<T, E>` ببساطة فهذا يعني أننا نسـتطيع اسـتخدام أي تابع يعمل على `Result<T, E>` بالإضافة إلى صيغة خاصة مثل العامل `?`.

19.3.3 النوع Never الذي لا يعيد أي قيمة

تمتلك لغة رست نوعًا خاصًا يدعى `!`، وهذا النوع معروف في لغة نظرية النوع بالنوع الفـارغ empty type لأنه لا يحتوي على أي قيم، إلا أننا نفضل أن نطلق عليه اسم "أبدًا Never" لأنه يحلّ مكان النوع المُعاد عنـدما لا تُعيد الدالة أي قيمة، إليك مثالًا على ذلك:

```
fn bar() -> ! {
    // --snip--
}
```

تُقرأ الشيفرة السابقة على أنّ الدالة `bar` لا تُعيد أي قيمة، وتسـمى الـدوال الـتي لا تُعيـد أي قيمـة بالـدوال المتباينة diverging functions. لا يمكننا إنشاء قيم من النوع `!` لذلك لا يمكن للدالة `bar` أن تُعيد أي شيء.

لكن ما فائدة نوع لا يمكنك أبدًا إنشاء قيم له؟ تذكر الشيفرة 5 سابقًا من القسم 19.1 الـتي كـانت جـزءًا من لعبة التخمين بالأرقام، ولنعيد إنتاج جزء منها هنا في الشيفرة 26.

```
        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };
```

[الشيفرة 26: بنية match بذراع ينتهي بتعليمة continue]

تخطينا بعض التفاصيل في هذه الشيفرة، إذ ناقشنا سابقًا في الفصل 6 في قسم بنية match للتحكم بسير البرامج، أن أذرع match يجب أن تُعيد جميعها النوع ذاته، وهذا السبب وراء عدم عمل الشيفرة التالية:

```
let guess = match guess.trim().parse() {
    Ok(_) => 5,
    Err(_) => "hello",
};
```

يجب أن يكون النوع guess في هذه الشيفرة عددًا صحيحًا وسلسلة وتتطلب رست أن يكون guess نوعًا واحدًا فقط. إذًا، ماذا تُعيد continue؟ كيف سُمحَ لنا بإعادة u32 من ذراع مع وجود ذراع آخر ينتهي بالتعليمة continue في الشيفرة 26؟

لربّما خمّنت ذلك فعلًا، إذ للتعليمة continue قيمة !، وذلك يعني أنه عندما تحسب رست النوع guess فإنها تنظر إلى ذراعي التطابق، الأول بقيمة u32 والأخير بقيمة !، ولأنه ليس من الممكن للقيمة ! أن تكون لها قيمة أبدًا، تقرر رست أن النوع guess هو u32.

الطريقة الرسمية لوصف هذا السلوك هي أنه يمكن إجبار التعبيرات من النوع ! على أي نوع آخر. يُسمح لنا بإنهاء ذراع match هذا بالكلمة المفتاحية continue لأن continue لا تُعيد قيمة، ولكن بدلًا من ذلك يُنقل عنصر التحكم مرةً أخرى إلى أعلى الحلقة، لذلك في حالة Err لا نعيّن قيمة للمتغير guess إطلاقًا.

النوع "أبدًا never" مفيدٌ في ماكرو panic! أيضًا؛ تذكر دالة unwrap التي نستدعيها على قيم Option<T> لإنتاج قيمة أو هلع بهذا التعريف:

```
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None`
value"),
        }
    }
}
```

يحدث أمر مماثل في هذه الشيفرة للشيفرة 26 ضمن match، إذ ترى رست أن val لديه النوع T و panic! من النوع ! لذا فإن نتيجة تعبير match الكلي هي T. تعمل هذه الشيفرة لأن panic! لا تنتج قيمة تنهي البرنامج. لن نعيد قيمة من unwrap في حالة None لذا فإن هذه الشيفرة صالحة.

يحتوي تعبير أخير على النوع `!` ألا وهو الحلقة:

```
print!("forever ");

loop {
    print!("and ever ");
}
```

لا تنتهي هنا الحلقة أبدًا لذا فإن `!` هي قيمة التعبير، ومع ذلك لن يكون هذا صحيحًا إذا ضمّنا `break` لأن الحلقة ستنتهي عندما تصل إلى `break`.

19.3.4 الأنواع ذات الحجم الديناميكي والسمة Sized

تحتاج رست إلى معرفة تفاصيل معينة حول الأنواع المستخدمة، مثل مقدار المساحة المراد تخصيصها لقيمة من نوع معين، وهذا يجعل من أحد جوانب نظام النوع الخاص به مربكًا بعض الشيء في البداية، تحديدًا مفهوم الأنواع ذات الحجم الديناميكي Dynamically Sized Types، ويشار إليها أحيانًا باسم DST أو الأنواع غير محددة الحجم unsized types، إذ تتيح لنا هذه الأنواع كتابة الشيفرات باستخدام قيم لا يمكننا معرفة حجمها إلا وقت التنفيذ.

لنتعمق في تفاصيل النوع ذو الحجم الديناميكي المسمى `str` الذي استخدمناه سابقًا في جميع أنحاء الكتاب، لاحظ أننا لم نقل `&str` وإنما `str` بذاتها، إذ تُعدّ من الأنواع ذات الحجم الديناميكي. لا يمكننا معرفة طول السلسلة حتى وقت التنفيذ، مما يعني أنه لا يمكننا إنشاء متغير من النوع `str`، ولا يمكننا أخذ وسيط من النوع `str`. ألقِ نظرةً على الشيفرة التالية التي لا تعمل:

```
let s1: str = "Hello there!";
let s2: str = "How's it going?";
```

تحتاج رست أن تعرف مقدار الذاكرة المراد تخصيصها لأي قيمة من نوع معين ويجب أن تُستخدم جميع قيم النوع نفس المقدار من الذاكرة. إذا سمحت لنا رست بكتابة هذه الشيفرة فستحتاج قيمتي `str` هاتين إلى شغل المقدار ذاته من المساحة، إلا أن للقيمتين أطوال مختلفة، إذ يحتاج `s1` إلى 12 بايت من التخزين ويحتاج `s2` إلى 15، ولهذا السبب لا يمكن إنشاء متغير يحمل نوعًا محدد الحجم ديناميكيًا.

إذًا ماذا نفعل؟ يجب أن تعلم الإجابة مسبقًا في هذه الحالة، إذ أن الحلّ هو بإنشاء الأنواع `s1` و `s2` و `&str` بدلًا من `str`. تذكر سابقًا من قسم "شرائح السلاسل النصية" في الفصل 4 أن هيكل بيانات الشريحة يخزن فقط موضع البداية وطول الشريحة، لذلك على الرغم من أن `&T` هي قيمة واحدة تخزن عنوان الذاكرة الخاص بالمكان الذي يوجد فيه `T` إلا أن `&str` هي قيمتان، ألا وهما عنوان `str` وطولها، ويمكننا على هذا النحو معرفة حجم قيمة `&str` في وقت التصريف، وهي ضعف طول `usize`، أي أننا نعرف دائمًا حجم `&str` بغض النظر

عن طول السلسلة التي تشير إليها. هذه هي الطريقة التي تُستخدم بها الأنواع ذات الحجم الديناميكي عمومًا في رست، إذ لهذه الأنواع مقدار إضافي من البيانات الوصفية metadata التي تخزن حجم المعلومات الديناميكية. القاعدة الذهبية للأنواع ذات الحجم الديناميكي هي أنه يجب علينا دائمًا وضع قيم للأنواع ذات الحجم الديناميكي خلف مؤشر من نوع ما.

يمكننا دمج `str` مع جميع أنواع المؤشرات، على سبيل المثال `Box<str>` أو `Rc<str>`، وقد فعلنا ذلك سابقًا ولكن بنوع ذو حجم ديناميكي مختلف، ألا وهو السمات traits، فكل سمة هي نوع ذو حجم ديناميكي يمكننا الرجوع إليه باستخدام اسم السمة. ذكرنا سابقًا في الفصل 17 من قسم استخدام كائنات السمة Object Trait أنه يجب وضع السمات خلف مؤشر لاستخدامها مثل كائنات سمات، مثل `&dyn Trait` أو `Box<dyn Trait>` (يمكن استخدام `Rc<dyn Trait>` أيضًا).

توفر رست سمة `Sized` للعمل مع الأنواع ذات الأحجام الديناميكية لتحديد ما إذا كان حجم النوع معروفًا أم لا في وقت التصريف، إذ تُطبق هذه السمة تلقائيًا لكل شيء يُعرف حجمه في وقت التصريف، كما تضيف رست ضمنيًا تقييدًا على `Sized` لكل دالة عامة. يُعامل تعريف دالة عامة مثل هذه:

```
fn generic<T>(t: T) {
    // --snip--
}
```

كما لو أننا كتبنا هذا:

```
fn generic<T: Sized>(t: T) {
    // --snip--
}
```

ستعمل الدوال العامة افتراضيًا فقط على الأنواع التي لها حجم معروف في وقت التصريف، ومع ذلك يمكنك استخدام الصيغة الخاصة التالية لتخفيف هذا التقييد:

```
fn generic<T: ?Sized>(t: &T) {
    // --snip--
}
```

الصفة مرتبطة بـ `?Sized` تعني أن "`T` قد تكون أو لا تكون `Sized`" وهذا الترميز يلغي الافتراض الذي ينص على وجود حجم معروف للأنواع العامة وقت التصريف. صيغة `?Trait` بهذا المعنى متاحة فقط للسمة `Sized` وليس لأي سمات أخرى.

لاحظ أيضًا أننا بدّلنا نوع المعامل `t` من `T` إلى `&T`، نظرًا لأن النوع قد لا يكون `Sized` فنحن بحاجة إلى استخدامه خلف نوع من المؤشرات، وفي هذه الحالة اخترنا مرجعًا.

19.4 الدوال functions والمغلفات closures المتقدمة

حان الوقت للتحدث عن بعض الخصائص المتقدمة المتعلقة بالمغلّفات والدوال بما في ذلك مؤشرات الدوال والمغلفات الراجعة Returing Closures.

19.4.1 مؤشرات الدوال

تحدثنا سابقًا عن كيفية تمرير المغلفات للدوال، ويمكننا أيضًا تمرير الدوال العادية للدوال. تفيد هذه التقنية عندما نريد تمرير دالة عرّفناها مسبقًا بدلًا من تعريف مغلف جديد. تُجبَر الدوال بالنوع `fn` (بحرف f صغير) -لا تخلط بينه وبين مغلف السمة `Fn`- يسمى نوع `fn` مؤشر دالة function pointer، ويسمح لك تمرير الدوال بمؤشرات الدوال باستخدام الدوال مثل وسطاء لدوال أُخرى.

تشابه صياغة مؤشرات الدوال لتحديد معامل مثل مؤشر صياغتها في المغلفات كما تبين الشيفرة 27، إذ عرّفنا تابع `add_one` الذي يضيف واحد إلى معامله. تأخذ الدالة `do_twice` معاملين، هما مؤشر دالة لأي دالة تأخذ معامل `i32` وتعيد النوع `i32`، وقيمة `i32` واحدة. تستدعي دالة `do_twice` الدالة `f` مرتين وتمرر قيمة `arg` وتضيف نتيجتَي استدعاء الدالة معًا، بينما تستدعي الدالة `main` الدالة `do_twice` مع الوسيطين `add_one` و 5.

اسم الملف: src/main.rs

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {}", answer);
}
```

[الشيفرة 27: استخدام نوع `fn` لقبول مؤشر دالة مثل وسيط]

تطبع الشيفرة السابقة ما يلي:

```
The answer is: 12
```

حددنا أن المعامل `f` في `do_twice` هو `fn` الذي يأخذ معامل واحد من النوع `i32` ويُعيد `i32`، ويمكن بعدها استدعاء `f` من داخل الدالة `do_twice`. يمكننا في `main` تمرير اسم الدالة `add_one` على أنه الوسيط الأول إلى `do_twice`.

على عكس المغلفات، فإن `fn` هو نوع وليس سمة، لذا نحدد `fn` مثل نوع معامل مباشرة بدلًا من تصريح معامل نوع معمم generic مع واحدة من سمات `fn` على أنه قيد سمة trait bound.

تطبّق مؤشرات الدالة سمات المغلفة الثلاثة (`Fn` و `FnMut` و `FnOnce`). يعني ذلك أنه بإمكانك دائمًا تمرير مؤشر الدالة مثل وسيط لدالة تتوقع مغلفًا. هذه هي الطريقة الأفضل لكتابة الدوال باستخدام النوع المعمم وواحد من مغلف السمات بحيث يمكن للدوال الأخرى قبول دوال أو مغلفات. هناك مثال واحد تستطيع فيه قبول `fn` فقط وليس المغلفات وهو عندما نتعامل مع شيفرة خارجية لا تحتوي على مغلفات. يمكن لدوال لغة البرمجة سي أن تقبل الدوال مثل وسطاء، لكن ليس لديها مغلفات.

لنأخذ مثالًا عن مكان استخدام مغلف معرّف ضمنيًا أو دالة مسماة، ولنتابع كيفية استخدام تابع `map` مقدم بسمة `Iterator` في المكتبة القياسية. يمكننا استخدام المغلف لاستخدام دالة `map` لتحويل شعاع أرقام إلى شعاع سلاسل نصية على النحو التالي:

```
    let list_of_numbers = vec![1, 2, 3];
      let list_of_strings: Vec<String> =
          list_of_numbers.iter().map(|i| i.to_string()).collect();
```

أو بتسمية التابع مثل وسيط `map` بدلًا من المغلف على النحو التالي:

```
    let list_of_numbers = vec![1, 2, 3];
      let list_of_strings: Vec<String> =
          list_of_numbers.iter().map(ToString::to_string).collect();
```

لاحظ أنه يجب استخدام الصيغة المؤهلة كليًاالتي تحدثنا عنها سابقًا في قسم "السمات المتقدمة" لأنه يوجد دوال متعددة جاهزة اسمها `to_string`. استخدمنا هنا الدالة `to_string` المعرّفة في سمة `ToString` التي تطبّقها المكتبة القياسية لأي نوع يطبّق `Display`.

تذكر سابقًا من القسم "قيم التعداد" في الفصل 6 أن اسم كل متغاير variant في تعداد enum عرّفناه يصبح أيضًا دالة تهيئة. يمكننا استخدام دوال التهيئة هذه مثل مؤشرات دالة تطبّق مغلفات السمة، ما يعني أنه يمكننا تحديد دوال التهيئة مثل وسطاء للتوابع التي تقبل المغلفات على النحو التالي:

```
    enum Status {
```

```
        Value(u32),
        Stop,
    }

    let list_of_statuses: Vec<Status> =
(0u32..20).map(Status::Value).collect();
```

أنشأنا هنا نسخةً من `Status::Value` باستخدام كل قيمة من النوع `u32` ضمن المجال الذي استُدعي إليه `map` باستخدام دالة التهيئة `Status::Value`. يفضّل بعض الناس هذه الطريقة وآخرون يفضلون استخدام المغلفات، النتيجة بعد التصريف مماثلة للطريقتين لذا استخدم الطريقة الأوضح بالنسبة لك.

19.4.2 إعادة المغلفات

تُمثل المغلفات بسمات، ما يعني أنه لا يمكن إعادة المغلفات مباشرةً، إذ يمكنك استخدام النوع الحقيقي الذي ينفذ السمة مثل قيمة معادة للدالة في معظم الحالات عندما تريد إعادة سمة بدلًا من ذلك، ولكن لا يمكنك فعل ذلك في المغلفات لأنها لا تحتوي نوعًا حقيقيًا يمكن إعادته. على سبيل المثال، يُمنع استخدام مؤشرات الدالة `fn` مثل نوع مُعاد.

تحاول الشيفرة التالية إعادة مغلف مباشرةً، ولكنها لن تُصرّف.

```
fn returns_closure() -> dyn Fn(i32) -> i32 {
    |x| x + 1
}
```

يكون خطأ المصرّف على النحو التالي:

```
$ cargo build
   Compiling functions-example v0.1.0 (file:///projects/functions-
example)
error[E0746]: return type cannot have an unboxed trait object
 --> src/lib.rs:1:25
  |
| fn returns_closure() -> dyn Fn(i32) -> i32 {
  |                         ^^^^^^^^^^^^^^^^ doesn't have a size
known at compile-time
  |
  = note: for information on `impl Trait`, see <https://doc.rust-
lang.org/book/ch10-02-traits.html#returning-types-that-implement-
traits>
```

```
help: use `impl Fn(i32) -> i32` as the return type, as all return
paths are of type `[closure@src/lib.rs:2:5: 2:8]`, which implements
`Fn(i32) -> i32`
  |
| fn returns_closure() -> impl Fn(i32) -> i32 {
  |                         ~~~~~~~~~~~~~~~~~~

For more information about this error, try `rustc --explain E0746`.
error: could not compile `functions-example` due to previous error
```

يشير الخطأ إلى سمة `Sized` مجددًا. لا تعرف رست ما هي المساحة اللازمة لتخـزين المغلـف، وقـد عرفنـا حل هذه المشكلة سابقًا، إذ يمكننا استخدام كائن سمة.

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

تُصرّف الشيفرة بصـورةٍ اعتياديـة هنـا. راجـع الفصـل 17 القسـم اسـتخدام كائنـات السـمة Trait Object.

سنتحدث لاحقًا عن الماكرو.

19.5 الماكرو Macros

استخدمنا الماكرو مثل `println!` سابقًا في هذا الكتاب، إلا أننا لم نتحدث بالكامل عما هو الماكرو وكيفية عمله، إذ تشير كلمة ماكرو إلى مجموعة من الميّزات في رسـت، ألا وهي المـاكرو التصـريحية declarative مـع `macro_rules!`، إضافةً إلى ثلاثة أنواع من الماكرو الإجرائي procedural:

- مـاكرو `#[derive]` مخصـص يحـدد شـيفرة مضـافة بسـمة `derive` المسـتخدمة على الهياكـل والمعدّدات.
- ماكرو شبيه بالسمة attribute، الذي يعرف سمات معينة تُستخدم على أية عنصر.
- ماكرو يشبه الدالة ويشابه استدعاءات الدالة ولكن يعمل على المفاتيح المحددة مثل وسائطها.

سنتحدث عن كلٍّ مما سبق بدوره ولكن لنتحدث أولًا عن حاجتنا للماكرو بالرغم من وجود الدوال.

19.5.1 الفرق بين الماكرو والدوال

الماكرو هو طريقة لكتابة شيفرة تكتب شيفرة أُخرى والمعروف بالبرمجـة الوصـفية metaprogramming، وحـدثنا في الملحـق "ت" عن سـمة `derive` الـتي تنشـئ تنفيـذًا لسـمات متعـددة، واسـتخدمنا أيضًـا مـاكرو `println!` و `vec!` سابقًا. تتوسع كل هذه الماكرو لتضيف شيفرةً أكثر من الشيفرة التي كُتبت يدويًا.

تفيد البرمجة الوصفية في تقليل كمية الشيفرة التي يجب كتابتها والمحافظة عليها وهو أيضًا أحد أدوار الدوال، لكن لدى الماكرو بعض القوى الإضافية غير الموجودة في الدوال.

يجب أن تصرّح بصمة الدالة signature على عدد ونوع المعاملات الموجودة في الدالة، أما في حالة الماكرو فيمكن أن يأخذ عدد متغير من المعاملات، إذ يمكننا استدعاء `println!("hello")` بوسيط واحد أو `println!("hello {}", name)` بوسيطين. يتوسع أيضًا الماكرو قبل أن يفسر المصرف معنى الشيفرة لذا يمكن للماكرو مثلًا تنفيذ سمة على أي نوع مُعطى ولا يمكن للدالة فعل ذلك لأنها تُستدعى وقت التنفيذ وتحتاج لسمة لتُنفذ وقت التصريف.

من مساوئ تنفيذ الماكرو بدلًا من الدالة هو أن تعاريف الماكرو أكثر تعقيدًا من تعاريف الدالة لأننا نكتب شيفرة رست لتكتب شيفرة رست، بالتالي تكون تعاريف الماكرو أكثر تعقيدًا للقراءة والفهم والمحافظة عليها من تعاريف الدالة. هناك فرق آخر مهم بين الماكرو والدوال هو أنه يجب تعريف الماكرو أو جلبه إلى النطاق في ملف قبل استدعائه، على عكس الدوال التي يمكنك تعريفها واستدعائها في كل وقت ومكان.

19.5.2 الماكرو التصريحي مع macro_rules! للبرمجة الوصفية العامة

أكثر أنواع الماكرو استخدامًا في رست هو الماكرو التصريحي الذي يسمى أحيانًا "ماكرو بالمثال macros by example" أو "ماكرو `macro_rules!`" أو ببساطة "ماكرو". يسمح لك الماكرو التصريحي بكتابة شيء مشابه لتعبير `match` في رست بداخله. تعابير `match` -كما تحدثنا في الفصل السادس- هي هياكل تحكم تقبل تعبيرًا وتقارن القيمة الناتجة من التعبير مع النمط وبعدها تنفذ الشيفرة المرتبطة مع النمط المُطابق. يقارن الماكرو أيضًا قيمةً مع أنماط مرتبطة بشيفرة معينة، وتكون القيمة في هذه الحالة هي الشيفرة المصدرية لرست المُمَررة إلى الماكرو. تُقارن الأنماط مع هيكل الشيفرة المصدرية والشيفرة المرتبطة بكل نمط، وعند حدوث التطابق يستبدل الشيفرة المُمَررة إلى الماكرو، ويحصل كل ذلك وقت التصريف.

نستخدم بنية `macro_rules!` لتعريف الماكرو. دعنا نتحدث عن كيفية استخدام `macro_rules!` بالنظر إلى كيفية تعريف ماكرو `vec!`، إذ تحدثنا سابقًا في الفصل الثامن عن كيفية استخدام ماكرو `vec!` من أجل إنشاء شعاع جديد بقيم معينة. ينشئ الماكرو التالي مثلًا شعاع جديد يحتوي على ثلاثة أعداد صحيحة.

```
let v: Vec<u32> = vec![1, 2, 3];
```

يمكن استخدام الماكرو `vec!` لإنشاء شعاع بعددين صحيحين أو شعاع بخمس سلاسل شرائح نصية string slice، ولا يمكننا فعل ذلك باستخدام الدوال لأننا لا نعرف عدد أو نوع القيم مسبقًا.

تبين الشيفرة 28 تعريفًا مبسطًا لماكرو `vec!`.

اسم الملف: src/main.rs

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

[الشيفرة 28: نسخة مبسطة من تعريف ماكرو `vec!`]

> يتضمن التعريف الفعلي لماكرو `vec!` في المكتبة القياسية شيفرة للحجز الصحيح للذاكرة مسبقًا، وهذه الشيفرة هي تحسين لم نضفه هنا لجعل المثال أبسط.

يشير توصيف `#[macro_export]` إلى أن هذا الماكرو يجب أن يبقى متاحًا عندما يجري إحضار الوحدة المُصرّفة crate المعرّفة داخلها الماكرو إلى النطاق، ولا يمكن إضافة الماكرو إلى النطاق دون هذا التوصيف.

عندما نبدأ بتعريف الماكرو مع `macro_rules!` ويكون اسم الماكرو الذي نعرّفه بدون علامة التعجب، يكون الاسم في هذه الحالة `vec` متبوعًا بقوسين معقوصين تدل على متن تعريف الماكرو.

يشابه الهيكل في متن `vec!` الهيكل في تعبير `match`، إذ لدينا هنا ذراع واحد مع النمط `( $( $x:expr ),* )` متبوعةً بالعامل `=>` وكتلة الشيفرة المرتبطة في النمط، وستُرسل الكتلة المرتبطة إذا تطابق النمط. بما أن هذا هو النمط الوحيد في الماكرو، هناك طريقة وحيدة للمطابقة، وأي أنماط أُخرى ستسبب خطأ، ويكون لدى الماكرو الأكثر تعقيدًا أكثر من ذراع واحدة.

تختلف الصيغة الصحيحة في تعاريف الماكرو عن صيغة النمط المذكور سابقًا في الفصل 18 لأن أنماط الماكرو تُطابق مع هيكل شيفرة رست بدلًا من القيم. لنتحدث عن ماذا تعني أقسام النمط في الشيفرة 28. لقراءة صيغة نمط ماكرو الكاملة راجع مرجع رست.

استخدمنا أولًا مجموعة أقواس لتغليف كامل النمط، واستخدمنا علامة الدولار (`$`) للتصريح عن متغير في نظام الماكرو الذي يحتوي على شيفرة رست مطابقة للنمط، إذ توضح إشارة الدولار أن هذا متغير ماكرو وليس متغير رست عادي. تأتي بعد ذلك مجموعةٌ من الأقواس التي تلتقط القيم التي تطابق النمط داخل القوسين لاستخدامها في الشيفرة المُستبدلة. توجد `$x:expr` داخل `$()`، التي تطابق أي تعبير رست وتعطي التعبير

الاسم `$x`. تشير الفاصلة التي تلي `$()` أنه يمكن أن يظهر هناك محرف فاصلة بعد الشيفرة الذي يطابق الشيفرة في `$()`، وتشير `*` إلى أن هناك نمط يطابق صفر أو أكثر مما يسبق `*`.

عندما نستدعي هذا الماكرو باستخدام `vec![1, 2, 3];`، يُطابق النمط `$x` ثلاث مرات مع التعابير الثلاث 1 و 2 و 3.

لننظر إلى النمط الموجود في متن الشيفرة المرتبطة مع هذا الذراع، إذ تُنشَئ `temp_vec.push()` داخل `$()*` لكل جزء يطابق `$()` في النمط صفر مرة أو أكثر اعتمادًا على كم مرة طابق النمط. تُبَدل `$x` مع كل جزء مطابق، وعندما نستدعي الماكرو باستخدام `vec![1, 2, 3];`، ستكون الشيفرة المُنشأة التي تستبدل هذا الماكرو على النحو التالي:

```
{
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
}
```

عرّفنا الماكرو الذي يستطيع أن يأخذ أي عدد من الوسطاء من أي نوع ويستطيع إنشاء شيفرة لإنشاء شعاع يحتوي العناصر المحددة.

لتعرف أكثر عن كيفية كتابة الماكرو، راجع وثائق ومصادر أُخرى على الشبكة مثل "الكتاب الصغير لماكرو رست The Little Book of Rust Macros" الذي بدأ فيه دانيل كيب Daniel Keep وتابعه لوكاس ويرث Lukas Wirth.

19.5.3 الماكرو الإجرائي لإنشاء شيفرة من السمات

الشكل الثاني من الماكرو هو الماكرو الإجرائي الذي يعمل أكثر مثل دالة (وهي نوع من الإجراءات). يقبل الماكرو الإجرائي بعض الشيفرة مثل دخل ويعمل على الشيفرة ويُنتج بعض الشيفرة مثل خرج بدلًا من مطابقة الأنماط وتبديل الشيفرة بشيفرة أُخرى كما يعمل الماكرو التصريحي. أنواع الماكرو الإجرائي الثلاث، هي: مشتقة derive مخصصة custom، أو مشابهة للسمة attribute-like، أو مشابهة للدالة function-like وتعمل كلها بطريقة مشابهة.

عند إنشاء ماكرو إجرائي، يجب أن يبقى التعريف داخل الوحدة المصرّفة الخاصة به بنوع وحدة مصرّفة خاص، وذلك لأسباب تقنية معقدة نأمل أن نتخلص من وجودها مستقبلًا، تبين الشيفرة 29 كيفية تعريف الماكرو الإجرائي، إذ أن `some_attribute` هو عنصر مؤقت لاستخدام نوع ماكرو معين.

اسم الملف: src/lib.rs

```
use proc_macro;

#[some_attribute]
pub fn some_name(input: TokenStream) -> TokenStream {
}
```

[الشيفرة 29: مثال لتعريف ماكرو إجرائي]

تأخذ الدالة الـتي تعـرّف المـاكرو الإجـرائي `TokenStream` مثـل دخـل وتنتج `TokenStream` في الخـرج. يُعرّف نوع `TokenStream` بالوحدة المصرّفة `proc_macro` المتضمنة في رست وتمثّل سلسلة من المفـاتيح. هذا هو صلب الماكرو: تكون الشيفرة المصدرية التي يعمل فيها المـاكرو هي الـدخل `TokenStream` والشـيفرة التي ينتجها الماكرو هي الخرج `TokenStream`. لدى الدالة سمة مرتبطة بها تحدد أي نوع من المـاكرو الإجـرائي يجب أن نُنشئ، ويمكن أيضًا الحصول على العديد من الماكرو الإجرائي في الوحدة المصرّفة ذاتها.

لنتحدث عن الأشكال المختلفة من الماكرو الإجرائي. سـنبدأ بالمـاكرو المشـتق الخـاص ونفسـر الاختلافـات البسيطة التي تجعل باقي الأشكال مختلفة.

19.5.4 كيفية كتابة ماكرو derive مخصص

لننشئ وحدة مصرّفة اسمها `hello_macro` الـتي تعـرف سـمةً اسـمها `HelloMacro` مـع دالـة مرتبطـة associated اسمها `hello_macro`، وبدلًا من إجبار المستخدمين على تنفيذ السـمة `HelloMacro` لكـل من أنـــواعهم، ســـنؤمن مـــاكرو إجـــرائي لكي يتمكن المســـتخدمين من توصـــيف نـــوعهم باســـتخدام `#[derive(HelloMacro)]` للحصول على تنفيذ افتراضي للدالة `hello_macro`.

سيطبع التنفيذ الافتراضي:

```
Hello, Macro! My name is TypeName!
```

إذ أن `TypeName` هو اسم النوع المُعرّفة عليه السمة، بمعنى آخر سنكتب وحدة مصرّفة تسمح لمبرمج آخـر بكتابة الشيفرة باستخدام حزمتنا المصرفة كما في الشيفرة 30.

اسم الملف:src/main.rs

```
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
```

```
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

[الشيفرة 30: الشيفرة التي يستطيع مستخدم الوحدة المصرفة فيها الكتابة عند استخدام الماكرو الإجرائي الخاص بنا]

ستطبع الشيفرة عندما تنتهي ما يلي:

```
Hello, Macro! My name is Pancakes!
```

الخطوة الأولى هي إنشاء وحدة مكتبة مصرّفة على النحو التالي

```
$ cargo new hello_macro --lib
```

بعدها نعرّف سمة `HelloMacro` والدّالة التابعة لها

اسم الملف: src/lib.rs

```
pub trait HelloMacro {
    fn hello_macro();
}
```

لدينا السمة ودوالها، ويستطيع هنا مستخدم الوحدة المصرّفة تنفيذ السمة للحصول على الوظيفة المرغوبة على النحو التالي:

```
use hello_macro::HelloMacro;

struct Pancakes;

impl HelloMacro for Pancakes {
    fn hello_macro() {
        println!("Hello, Macro! My name is Pancakes!");
    }
}

fn main() {
    Pancakes::hello_macro();
```

```
}
```

ولكن سيحتاج المستخدم لكتابة كتلة التنفيذ لكل نوع يرغب باستخدامه مع `hello_macro`، ونريد إعفائهم من ذلك. إضافةً إلى ذلك، لا نستطيع أن نؤمّن للتابع `hello_macro` التنفيذ الافتراضي الذي سيطبع اسم نـوع السمة المُطبقة عليه، إذ ليس لدى رست قدرة على الفهم لذا لا تسـتطيع البحث عن اسـم النـوع وقت التنفيـذ، وفي هذه الحالة نحن بحاجة لماكرو لإنشاء شيفرة وقت التنفيذ.

الخطوة التالية هي تعريف الماكرو الإجرائي. يحتاج الماكرو الإجرائي حتى الآن إلى وحـدة مصـرّفة خاصـة بـه، ربما سيُرفع هذا التقييد بالنهاية. يأتي اصـطلاح الوحـدات المصـرّفة الهيكليـة والوحـدات المصـرّفة للمـاكرو على النحو التالي: يسمى الماكرو الإجرائي الخاص المشتق `foo_derive` لاسـم موحـدة مصـرفة `foo`. لنبـدأ بإنشـاء وحدة مصرّفة جديدة اسمها `hello_macro_derive` داخل المشروع `hello_macro`.

```
$ cargo new hello_macro_derive --lib
```

الوحدتان المصرّفتان مرتبطتان جدًا، لذلك سننشـئ وحـدةً مصـرّفةً للمـاكرو الإجـرائي داخـل مجلـد الوحـدة المصرّفة `hello_macro`. يجب علينا تغيـير تنفيـذ المـاكرو الإجـرائي في `hello_macro_derive` إذا غيرنـا تعريف السـمة في `hello_macro` أيضًـا. تحتـاج الوحـدتان المصـرّفتان أن تُنشَـرا بصـورةٍ منفصـلة ويجب أن يضيف مستخدمو هاتين الوحدتين المصرّفتين مثل اعتماديتين dependencies وجلبهما إلى النطـاق. يمكن - بدلًا من ذلك- جعل الحزمة المصرّفة `hello_macro` تستخدم `hello_macro_derive` مثل اعتمادية وتعيـد تصـدير شـيفرة المـاكرو الإجـرائي ولكن الطريقـة الـتي بنينـا فيهـا المشـروع تسـمح للمـبرمجين اسـتخدام `hello_macro` حتى لو كانوا لا يرغبون باستخدام وظيفة `derive`.

يجب علينا التصريح عن الوحـدة المصـرفة `hello_macro_derive` مثـل وحـدة مصـرفة لمـاكرو إجـرائي ونحتاج أيضًا إلى وظائف من الوحدات المصرّفة `syn` و `quote` كمـا سـنرى بعـد قليـل لـذا سـنحتاج لإضـافتهم كاعتماديات. أضِف التالي إلى ملف Cargo.toml من أجل `hello_macro_derive`:

اسم الملف: hello_macro_derive/Cargo.toml

```
[lib]
proc-macro = true

[dependencies]
syn = "1.0"
quote = "1.0"
```

لنبـدأ بتعريـف المـاكرو الإجـرائي. ضـع الشـيفرة 31 في ملـف src/lib.rs من أجـل الوحـدة المصـرّفة `hello_macro-derive`. لاحظ أن الشيفرة لن تصرّف حتى نضيف التعريف لدالة `impl_hello_macro`.

اسم الملف: hello_macro_derive/src/lib.rs

```
use proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // إنشاء تمثيل لشيفرة رست مثل شجرة صيغة يمكننا التلاعب بها
    let ast = syn::parse(input).unwrap();
    // بناء تنفيذ السمة
    impl_hello_macro(&ast)
}
```

[الشيفرة 31: الشيفرة التي تتطلبها معظم الوحدات المصرّفة للماكرو الإجرائي لكي تعالج شيفرة رست]

لاحظ أننا قسّمنا الشيفرة إلى دالة `hello_macro_derive` المسؤولة عن تحليل `TokenStream`، ودالة `Impl_hello_macro` المسؤولة عن تحويل شجرة الصيغة syntax tree التي تجعل كاتبة الماكرو الإجرائي لتكون أكثر ملائمة. ستكون الشيفرة في الدالة الخارجية (في هذه الحالة `hello_macro_derive`) هي نفسها لمعظم الوحدات المصرّفة للماكرو الإجرائي الذي تراه أو تنشئه، وستكون الشيفرة التي تحددها في محتوى الدالة الداخلية (في هذه الحالة `impl_hello_macro`) مختلفة اعتمادًا على غرض الماكرو الإجرائي.

أضفنا ثلاث وحدات مصرّفة هي `proc_macro` و `syn` و `quote`. لا نحتاج لإضافة الوحدة المصرفة `proc_macro` إلى الاعتماديات في Cargo.toml لأنها تأتي مع رست، وهذه الوحدة المصرفة هي واجهة برمجة التطبيق للمصرف التي تسمح بقراءة وتعديل شيفرة رست من شيفرتنا.

تحلّل الوحدة المصرّفة `syn` شيفرة رست من سلسلة نصية إلى هيكل بيانات يمكننا إجراء عمليات عليه. تحوّل الوحدة المصرّفة `quote` هيكل بيانات `syn` إلى شيفرة رست. تسهّل هذه الوحدات المصرّفة تحليل أي نوع من شيفرة رست يمكن أن نعمل عليه. تُعد كتابة محلل parser كامل لرست أمرًا صعبًا.

تُستدعى دالة `hello_macro_derive` عندما يحدد مستخدم مكتبتنا `#[derive(HelloMacro)]` على نوع، وهذا ممكن لأننا وصفّنا دالة `hello_macro_dervie` باستخدام `proc_macro_dervie` وحددنا اسم `HelloMacro` الذي يطابق اسم سِمتنا، وهذا هو الاصطلاح الذي يتبعه معظم الماكرو الإجرائي.

تحوّل دالة `hello_macro_derive` أولًا `input` من `TokenStream` إلى هيكل بيانات يمكن أن نفسره ونجري عمليات عليه. هنا يأتي دور `syn`. تأخذ دالة `parse` في `syn` القيمة `TokenStream` وتُعيد هيكل

`DeriveInput` يمثّل شيفرة رست المحلّلة. تظهر الشيفرة 32 الأجزاء المهمة من هيكل `DeriveInput` الـتي نحصل عليها من تحليل السلسلة النصية `struct Pancakes;`.

```
DeriveInput {
    // --snip--

    ident: Ident {
        ident: "Pancakes",
        span: #0 bytes(95..103)
    },
    data: Struct(
        DataStruct {
            struct_token: Struct,
            fields: Unit,
            semi_token: Some(
                Semi
            )
        }
    )
}
```

[الشيفرة 32: نسخة `DeriveInput` التي نحصل عليها من تحليل الشيفرة التي فيها سمة الماكرو في الشيفرة 30]

تظهر حقول هذا الهيكل بأن شيفرة رست التي حللناها هي هيكل وحدة مع `ident` (اختصـارًا للمعـرّف، أي الاسم) الخاصة بالاسم `Pancakes`. هناك حقول أخرى في هذا الهيكل لوصـف كـل أنـواع شـيفرة رسـت. راجـع وثائق syn من أجل `DeriveInput` لمعلومات أكثر.

سنعرِّف قريبًا دالة `impl_hello_macro`، التي سنبني فيها شيفرة رست الجديدة التي نريـد ضـمها، لكن قبل ذلك لاحظ أن الخرج من الماكرو المشتق الخاص بنا هو أيضًا `TokenStream`، إذ تُضـاف `TokenStream` المُعادة إلى الشيفرة التي كتبها مستخدمو حزمتنا المصرّفة، لذلك سيحصلون عند تصريف الوحدة المصرّفة على وظائف إضافية قدمناها في `TokenStream` المعدلة.

ربما لاحظت أننا استدعينا `unwrap` لتجعل الدالة `hello_macro_derive` تهلع إذا فشل استدعاء الدالـة `proc_macro_derive`. يجب أن يهلع الماكرو الإجرائي على الأخطاء، لأنه يجب أن تعيد الدالة `syn::parse` الـقيمة `TokenStream` بدلًا من `Result` لتتوافق مع واجهة برمجة التطبيقات للمـاكرو الإجـرائي. بسّـطنا هـذا المثال باستخدام `unwrap`، إلا أنه يجب تأمين رسالة خطأ محـددة أكـثر في شـيفرة الإنتـاج باسـتخدام `panic!` أو `expect`.

الآن لدينا الشيفرة لتحويل شيفرة رست الموصّفة من `TokenStream` إلى نسخة `DeriveInput` لننشئ الشيفرة التي تطبّق سمة `HelloMacro` على النوع الموصّف كما تظهر الشيفرة 33.

اسم الملف: hello_macro_derive/src/lib.rs

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}!", stringify!
(#name));
            }
        }
    };
    gen.into()
}
```

[الشيفرة 33: تنفيذ سمة `HelloMacro` باستخدام شيفرة رست المحلّلة]

نحصل على نسخة هيكل `Indent` يحتوي على الاسم (المُعرّف) على النوع الموصّف باستخدام `ast.ident`. يظهر الهيكل في الشيفرة 32 أنه عندما نفّذنا دالة `impl_hello_macro` على الشيفرة 30 سيكون لدى `ident` التي نحصل عليها حقل `ident` مع القيمة `"Pancakes"`، لذلك سيحتوي المتغير `name` في الشيفرة 33 نسخة هيكل `Ident`، الذي سيكون سلسلة نصية `"Pancakes"` عندما يُطبع، وهو اسم الهيكل في الشيفرة 30.

يسمح لنا ماكرو `quote!` بتعريف شيفرة رست التي نريد إعادتها. يتوقع المصرف شيئًا مختلفًا عن النتيجة المباشرة لتنفيذ ماكرو `quote!`، لذا نحتاج لتحويله إلى `TokenStream`، وذلك عن طريق استدعاء تابع `into` الذي يستهلك التعبير الوسطي ويعيد القيمة من النوع `TokenStream` المطلوب.

يؤمن ماكرو `quote!` تقنيات قولبة templating جيدة، إذ يمكننا إدخال `#name` ويبدّلها `quote!` بالقيمة الموجودة في المتغير `name`، ويمكنك أيضًا إجراء بعض التكرارات بطريقة مشابهة لكيفية عمل الماكرو العادي. راجع توثيق الوحدة المصرّفة quote لتعريف وافي عنها.

نريد أن يُنشئ الماكرو الإجرائي تنفيذًا لسمة `HelloMacro` للنوع الذي يريد توصيفه المستخدم، والذي نحصل عليه باستخدام `#name`. يحتوي تنفيذ السمة دالةً واحدةً `hello_macro` تحتوي على الوظيفة المراد تقديمها ألا وهي طباعة `Hello, Macro! My name is` وبعدها اسم النوع الموصَّف.

الماكرو `stringify!` المُستخدم هنا موجود داخل رست، إذ يأخذ تعبير رست مثل `1 + 2` ويحول التعبير إلى سلسلة نصية مجرّدة مثل `"1 + 2"`. هذا مختلف عن `format!` و `println!`، الماكرو الـذي يقيّم التعبـير ويحول القيمة إلى `String`. هناك احتمال أن يكون الدخل `#name` تعبيرًا للطباعة حرفيًا literally، لذا نسـتخدم `stringify!`، الذي يوفر مساحةً محجوزةً عن طريق تحويل `#name` إلى سلسلة نصية مجرّدة وقت التصريف.

الآن، يجب أن ينتهي `cargo build` بنجـاح في كـل من `hello_macro` و `hello_macro_derive`. لنربط هذه الوحدات المصرّفة مع الشيفرة في الشيفرة 30 لنرى كيفيـة عمـل المـاكرو الإجـرائي. أنشـئ مشـروعًا ثنائيًـا جديـدًا في مجلـد المشـاريع باسـتخدام `cargo new pancakes`. نحتـاج لإضـافة `hello_macro` و `hello_macro_derive` مثل اعتماديات في ملف Cargo.toml الخـاص بالوحـدة المصـرّفة `pancakes`. إذا نشـرت النسـخ الخاصـة بـك من `hello_macro` و `hello_macro_derive` إلى crates.io فسـتكون اعتماديات عادية، وإذا لم يكونوا كذلك فبإمكانك تحديدها مثل اعتماديات `path` على النحو التالي:

```
hello_macro = { path = "../hello_macro" }
hello_macro_derive = { path = "../hello_macro/hello_macro_derive" }
```

ضـع الشـيفرة 30 في الملـف src/main.rs ونفـذ `cargo run` يجب أن تطبـع العبـارة Hello, Macro! My name is Pancakes!. كان تنفيذ سمة `HelloMacro` من الماكرو الإجرائي متضمنًا دون أن تحتاج الوحدة المصرفة `pancakes` أن تنفّذه. أضاف `#[derive(HelloMacro)]` تنفيذ السمة.

سنتحدث تاليًا عن الاختلافات بين الأنواع الأُخرى من الماكرو الإجرائي من الماكرو المشتق الخاص.

19.5.5 الماكرو الشبيه بالسمة

يشابه الماكرو الشبيه بالسمة الماكرو المشتق الخاص لكن بدلًا من إنشاء شيفرة لسمة `derive` يسمح لـك بإنشاء سمات جديدة وهي أيضًا أكثر مرونة، تعمل `derive` فقط مـع الهياكـل والـتعدادات enums، يمكن أن تطبق السمات attributes على عناصر أُخرى أيضًا مثل الـدوال. فيمـا يلي مثـال عن اسـتخدام المـاكرو الشـبيه بالسمة: لنقل أن لديك سمة اسمها `route` توصّف الدوال عند استخدام إطار عمل تطبيق ويب:

```
#[route(GET, "/")]
fn index() {
```

تُعرَّف سمة `#[route]` بإطار العمل مثل ماكرو إجـرائي. سـتكون بصـمة دالـة تعريـف المـاكرو على النحـو التالي:

```
#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream {
```

لدينا هنا معاملان من النوع `TokenStream`، الأول هو من أجل محتوى السمة (جزء `GET, "/"`)، والثاني هو لمتن العنصر الذي ترتبط به السمة والذي هو `fn index() {}` في هذه الحالة والباقي هو متن الدالة.

عدا عن ذلك، تعمل الماكرو الشبيهة بالسمة بنفس طريقة الماكرو المشتق الخاص عن طريق إنشاء وحدة مصرفة مع نوع الوحدة المصرّفة `proc-macro` وتنفذ الدالة التي تنشئ الشيفرة المرغوبة.

19.5.6 الماكرو الشبيه بالدالة

يعرّف الماكرو الشبيه بالدالة الماكرو ليشبه استدعاءات الدوال، وعلى نحوٍ مشابه لماكرو `macro_rules!`، فهي أكثر مرونة من الدوال؛ إذ يستطيع الماكرو أخذ عدد غير معروف من الوسطاء، ولكن يمكن أن يعرّف ماكرو `macro_rules!` فقط باستخدام صيغة تشبه المطابقة التي تحدثنا عنها سابقًا في قسم "الماكرو التصريحي مع macro_rules! للبرمجة الوصفية العامة". يأخذ الماكرو الشبيه بالدالة معامل `TokenStream` ويعدل تعريفها القيمة `TokenStream` باستخدام شيفرة رست كما يفعل الماكرو الإجرائي السابق. إليك مثالًا عن ماكرو شبيه بالدالة هو ماكرو `sql!` التي يمكن استدعاؤه على النحو التالي:

```
#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream {
```

يحلل هذا الماكرو تعليمة SQL داخله ويتحقق إذا كانت صياغتها صحيحة، وهذه المعالجة أعقد مما يستطيع `macro_rules!` معالجته ويكون تعريف ماكرو `sql!` على النحو التالي:

```
#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
```

يشابه التعريف بصمة الماكرو المشتق الخاص، إذ أخذنا المفاتيح التي داخل القوسين وأعدنا الشيفرة التي نريد إنشاءها.

19.6 خاتمة

أصبح لديك الآن بعض من أدوات رست في جعبتك التي لن تستخدمها على نحوٍ متكرر، ولكن ستكون متاحة في بعض الأمور المحددة. عرضنا العديد من المواضيع المعقدة التي قد تواجهها في اقتراحات رسائل الخطأ أو في شيفرات الأخرين البرمجية التي ستكون قادرًا على ملاحظة هذه السياقات والصياغات فيها. استخدم هذا الفصل مثل مرجع للوصول إلى الحل.

سنضع تاليًا كل شيء تعلمناه في سياق هذا الكتاب للاستخدام العملي وسننفّذ مشروعًا جديدًا.

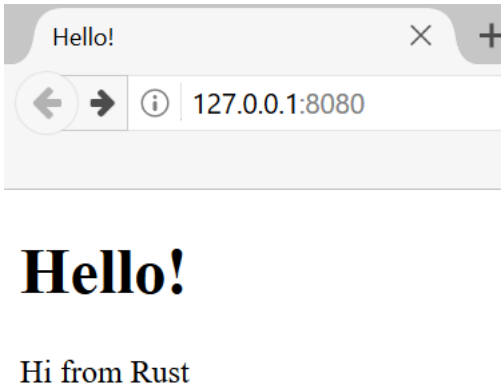
أكبر موقع توظيف عن بعد في العالم العربي

ابحث عن الوظيفة التي تحقق أهدافك وطموحاتك
المهنية في أكبر موقع توظيف عن بعد

تصفح الوظائف الآن

20. بناء خادم ويب متعدد مهام المعالجة

بعد رحلة طويلة وصلنا إلى نهاية الكتاب. سنبني في هـذا الفصـل مشـروعًا لتوضـيح بعض المفـاهيم الـتي تحدثنا عنها في الفصول السابقة. سنبني خادم ويب يعرض "hello" ويشبه الشكل 1 في متصفح الويب.



الشكل 15: مشروعنا الأخير المشترك

هذه هي خطة بناء خادم الويب:

1. مقدمة عن TCP و HTTP
2. الاستماع إلى اتصالات TCP على المقبس socket
3. تحليل عدد صغير من طلبات HTTP
4. إنشاء استجابة HTTP مناسبة
5. تطوير خرج الخادم بمجمع خيط thread pool

قبل البدء، يجب التنويه على أن هذه الطريقة ليست أفضل طريقة لبناء خادم ويب باستخدام رست، إذ نشر أعضاء المجتمع وحدات مصرفة جاهزة للتطبيق على creats.io، والتي تقدم خوادم ويب أكثر اكتمالًا وتطبيقات لمجمع خيط أفضل من الذي سنبنيه، ولكن هدفنا من هذا الفصل هو مساعدتك على التعلم وليس اختيار الطريق الأسهل. يمكننا اختيار مستوى التجريد الذي نريد العمل معه لأن رست هي لغة برمجية للأنظمة ويمكن الانتقال لمستوى أدنى مما هو ممكن أو عملي في بعض اللغات الأُخرى، لذلك سنكتب خادم HTTP بسيط ومجمع الخيط يدويًا لنتعلم الأفكار والتقنيات العامة الموجودة في الوحدات المصرفة التي يمكن أن تراها في المستقبل.

20.1 بناء خادم ويب أحادي الخيط

سنبدأ بإنشاء خادم ويب أحادي الخيط، ولكن قبل أن نبدأ دعنا نراجع البروتوكولات المستخدمة في إنشاء خوادم الويب. تفاصيل هذه البروتوكولات هي خارج نطاق موضوعنا هنا إلا أن مراجعة سريعة ستمنحك المعلومات الكافية.

البروتوكولان الأساسيان المعنيان في خوادم الويب هما بروتوكول نقل النصوص الفائقة Hypertext Transfer Protocol -أو اختصارًا HTTP- وبروتوكول تحكم النقل Transmission Control Protocol -أو اختصارًا TCP، وهما بروتوكولا طلب-استجابة؛ يعني أن العميل يبدأ الطلبات ويسمع الخادم الطلبات ويقدم استجابةً للعميل، ويُعرّف محتوى هذه الطلبات والاستجابات عبر هذه البروتوكولات.

يصف بروتوكول TCP تفاصيل انتقال المعلومات من خادم لآخر ولكن لا يحدد نوع المعلومات. يبني HTTP فوق TCP عن طريق تعريف محتوى الطلبات والاستجابات. يمكن تقنيًا استخدام HTTP مع بروتوكولات أُخرى لكن في معظم الحالات يرسل HTTP البيانات على بروتوكول TCP. سنعمل مع البايتات الخام في طلبات واستجابات TCP و HTTP.

20.1.1 الاستماع لاتصال TCP

يجب أن يستمع خادم الويب إلى اتصال TCP لذا سنعمل على هذا الجزء أولًا. تقدم المكتبة القياسية وحدة `std::net` التي تسمح لنا بذلك. لننشئ مشروعًا جديدًا بالطريقة الاعتيادية:

```
$ cargo new hello
     Created binary (application) `hello` project
$ cd hello
```

الآن اكتب الشيفرة 1 في الملف src/main.rs لنبدأ. ستسمع هذه الشيفرة إلى العنوان المحلي "127.0.0.1:7878" لمجرى TCP stream القادم، وعندما تستقبل مجرى قادم ستطبع `Connection established!`.

اسم الملف: src/main.rs

```
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        println!("Connection established!");
    }
}
```

[الشيفرة 1: الاستماع للمجاري القادمة وطباعة رسالة عند استقبال مجرى]

يمكننا الاستماع لاتصال TCP على هذا العنوان "127.0.0.1:7878" باستخدام `TcpListner`، إذ يمثّل القسم قبل النقطتين عنوان IP الذي يمثل الحاسوب (هذا العنوان هو نفسه لكل الحواسيب وليس لحاسوب المستخدم حصريًا)، ورقم المنفذ هو 7878. اخترنا هذا المنفذ لسببين: لا يُقبل HTTP على هذا المنفذ لذا لا يتعارض الخادم بأي خدمة ويب ربما تحتاجها على جهازك، و 7878 هي كلمة rust مكتوبة على لوحة أرقام الهاتف.

تعمل دالة `bind` في هذا الحالة مثل دالة `new` التي ترجع نسخة `TcpListner` جديدة. تسمى الدالة `bind` لأن الاتصال بمنفذ للاستماع إليه هي عملية تُعرف باسم الربط لمنفذ binding to a port. تعيد الدالة `bind` القيمة `Results<T, E>` التي تشير أنه من الممكن أن يفشل الربط. يتطلب الاتصال بالمنفذ 80 امتيازات المسؤول (يستطيع غير المسؤولين فقط الاستماع في المنافذ الأعلى من 1023)، لذا لا يعمل الارتباط إذا حاولت الاتصال بالمنفذ 80 بدون كونك مسؤول، ولا يعمل الارتباط أيضًا إذا نفذنا نسختين من برنامجنا أي لدينا برنامجين يستمعان لنفس المنفذ. لا يلزمنا أن نتعامل مع هكذا أخطاء لأننا نكتب خادم بسيط لأغراض تعليمية فقط. نستعمل `unwrap` لإيقاف البرنامج إذا حصلت أي أخطاء.

يعيد التابع `incoming` على `TcpListner` مكرّرًا iterator يعطي سلسلةً من المجاري (مجاري نوع `TcpStream` تحديدًا). يمثل المجرى الواحد اتصالًا مفتوحًا بين العميل والخادم، والاتصال هو الاسم الكامل لعملية الطلب والاستجابة التي يتصل فيها العميل إلى الخادم، وينشئ الخادم استجابةً ويغلق الاتصال. كذلك، سنقرأ من `TcpStream` لرؤية ماذا أرسل العميل وكتابة استجابتنا إلى المجرى لإرسال البيانات إلى العميل. ستعالج حلقة `for` عمومًا كل اتصال بدوره وتضيف سلسلة من المجاري لنتعامل معها.

تتألف حتى الآن طريقتنا للتعامل مع المجرى من استدعاء unwrap لينهي البرنامج إذا كان للمجرى أي أخطاء، وإذا لم يكن هناك أخطاء يطبع البرنامج رسالة، وسنضيف وظائفًا إضافية في حالة النجاح في الشيفرة التالية. سبب استقبال أخطاء من تابع incoming عندما يتصل عميل بالخادم هو أننا نكرّر زيادةً عن حد محاولات الاتصال بدلًا من أن نكرّر أعلى من حد الاتصالات؛ فقد تفشل محاولات الاتصال لعدد من الأسباب ويتعلق العديد منها بنظام التشغيل، فمثلًا تحدّد الكثير من أنظمة التشغيل عدد الاتصالات المفتوحة بالوقت الذي تدعمها، وستعطي أي اتصالات جديدة خطأ حتى تُغلق أي اتصالات مفتوحة.

لنحاول تنفيذ هذه الشيفرة، استدعِ cargo run في الطرفية وحمّل 127.0.0.1:7878 في متصفح الويب. يجب أن يظهر المتصفح رسالة الخطأ "إعادة ضبط الاتصال" لأن الخادم لا يرسل أي بيانات حاليًا، لكن عندما تنظر إلى الطرفية يجب أن ترى عدد من الرسائل المطبوعة عندما يتصل المتصفح بالخادم.

```
Running `target/debug/hello`
Connection established!
Connection established!
Connection established!
```

سنرى أحيانًا عددًا من الرسائل المطبوعة لطلب متصفح واحد، ويعود سبب ذلك إلى أن المتصفح أنشأ طلبًا الصفحة وكذلك لعدد من الموارد الأخرى مثل أيقونة favicon.ico التي تظهر على صفحة المتصفح. يمكن أن تعني أيضًا أن المتصفح يحاول الاتصال بالخادم مرات متعددة لأنه لا يتجاوب مع أي بيانات. يُغلق الاتصال كجزء من تنفيذ drop عندما تخرج stream عن النطاق وتُسقط في نهاية الحلقة. تتعامل المتصفحات أحيانًا مع الاتصالات المغلقة بإعادة المحاولة لأن هذه المشكلة يمكن أن تكون مؤقتة. العامل المهم أنه حصلنا على مقبض لاتصال TCP.

تذكر أن توقف البرنامج بالضغط على المفتاحين "ctrl-c" عندما تنتهي من تنفيذ نسخة معينة من الشيفرة، بعدها أعد تشغيل البرنامج باستدعاء أمر cargo run بعد إجراء أي تعديل على الشيفرة للتأكد من أنك تنفذ أحدث إصدار منها.

20.1.2 قراءة الطلب

دعنا ننفّذ وظيفةً لقراءة الطلب من المتصفح، إذ سنبدأ بدالة جديدة لمعالجة الاتصالات من أجل الفصل بين الحصول على اتصال وإجراء بعض الأعمال بالاتصال. سنقرأ دالة handle_connection البيانات من مجرى TCP وتطبعها لرؤية البيانات التي أُرسلت من المتصفح. غيّر الشيفرة لتصبح مثل الشيفرة 2.

اسم الملف: src/main.rs

```
use std::{
    io::{prelude::*, BufReader},
```

```
    net::{TcpListener, TcpStream},
};

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    println!("Request: {:#?}", http_request);
}
```

[الشيفرة 2: القراءة من `TcpStream` وطباعة البيانات]

نضيف `std::io::prelude` و `std::io::BufReader` إلى النطاق للحصول على سمات وأنواع تسمح لنا بالقراءة من والكتابة على المجرى. بدلًا من طباعة رسالة تقول أننا اتصلنا، نستدعي الدالة الجديدة `handle_connection` في حلقة `for` في الدالة `main` ونمرّر `stream` إليها.

أنشأنا نسخة `BufReader` في دالة `handle_connection` التي تغلف المرجع المتغيّر إلى `stream`. يضيف `BufReader` تخزينًا مؤقتًا عن طريق إدارة الاستدعاءات إلى توابع سمة `std::io::Read`.

أنشأنا متغيرًا اسمه `http_request` لجمع أسطر الطلب التي أرسله المتصفح إلى الخادم، ونشير أننا نريد جمع هذه الأسطر في شعاع عن طريق إضافة توصيف نوع `Vec<_>`.

ينفّذ `BufReader` سمة `std::io::BufRead` التي تؤمن التابع `lines`، الذي يعيد مكرّر `Result<String, std::io::Error>` عن طريق فصل مجرى البيانات أينما ترى بايت سطر جديد.

للحصول على كل `String`، نربط ونزيل تغليف `unwarp` كل `Result`. يمكن أن تكون `Result` خطأ إذا كانت البيانات ليست UTF-8 صالح أو كان هناك مشكلة في القراءة من المجرى. مجددًا، يمكن لبرنامج إنتاجي حل هذه المشكلات بسهولة ولكننا اخترنا إيقاف البرنامج في حالة الخطأ للتبسيط.

يشير المتصفح إلى نهاية طلب HTTP عن طريق إرسال محرفي سطر جديد على الترتيب، لذا للحصول على طلب من المجرى نأخذ الأسطر حتى نصل إلى سلسلة نصية فارغة. عندما نجمع الأسطر في الشعاع سنطبعهم باستخدام تنسيقات جذابة pretty لتنقيح الأخطاء لكي ننظر إلى التعليمات التي يرسلها المتصفح إلى الخادم.

لنجرب هذه الشيفرة. ابدأ البرنامج واطلب الصفحة في المتصفح مجددًا. لاحظ أنك ستحصل على صفحة خطأ في المتصفح، ولكن خرج البرنامج في الطرفية سيكون مشابهًا للتالي:

```
$ cargo run
   Compiling hello v0.1.0 (file:///projects/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target/debug/hello`
Request: [
    "GET / HTTP/1.1",
    "Host: 127.0.0.1:7878",
    "User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15;
rv:99.0) Gecko/20100101 Firefox/99.0",
    "Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image
/webp,*/*;q=0.8",
    "Accept-Language: en-US,en;q=0.5",
    "Accept-Encoding: gzip, deflate, br",
    "DNT: 1",
    "Connection: keep-alive",
    "Upgrade-Insecure-Requests: 1",
    "Sec-Fetch-Dest: document",
    "Sec-Fetch-Mode: navigate",
    "Sec-Fetch-Site: none",
    "Sec-Fetch-User: ?1",
    "Cache-Control: max-age=0",
]
```

اعتمادًا على المتصفح يمكن أن تحصل على خرج مختلف قليلًا. نطبع الآن طلبات البيانات ويمكن مشاهدة لماذا نحصل على اتصالات متعددة من طلب واحد من المتصفح بتتبع المسار الذي بعد `GET` في أول سطر من

الطلب. إذا كانت الاتصالات المتعددة كلها تطلب "/"، نعرف أن المتصفح يحاول إيجاد "/" باستمرار لأنه لم يحصل على استجابة من برنامجنا.

لنفصّل بيانات الطلب لفهم ما يطلبه المتصفح من برنامجنا.

20.1.3 نظرة أقرب على طلب HTTP

بروتوكول HTTP أساسه نصي وتكون طلباته على الشكل التالي:

```
Method Request-URI HTTP-Version CRLF
headers CRLF
message-body
```

السطر الأول هو سطر الطلب الذي يحتوي معلومات عما يطلبه العميل، إذ يدل القسم الأول من سطر الطلب على التابع المستخدم مثل `GET` أو `POST` الذي يصف كيفية إجراء العميل لهذا الطلب. استخدم عميلنا طلب `GET`، وهذا يعني أنه يطلب معلومات؛ بينما يشير القسم الثاني "/" من سطر الطلبات إلى معرّف الموارد الموحد Uniform Resource Identifier -أو اختصارًا URI- الذي يطلبه العميل. يشابه URI محدد الموارد الموحد Uniform Resource Locator -أو URL اختصارًا- ولكن ليس تمامًا، إذ أن الفرق بينهم ليس مهمًا لهذا المشروع، لكن تستخدم مواصفات HTTP المصطلح URI لذا نستبدل هنا URL بالمصطلح URI.

القسم الأخير هو نسخة HTTP التي يستخدمها العميل، وينتهي الطلب بسلسلة CRLF (تعني CRLF محرف العودة إلى أول السطر والانتقال سطر للأسفل Carriage Return and Line Feed وهما مصطلحان من أيام الآلة الكاتبة). يمكن كتابة سلسلة CRLF مثل `\r\n` إذ أن `\r` هي محرف العودة إلى أول السطر و `\n` هو الانتقال سطر للأسفل. تفصل سلسلة CRLF سطر الطلب من باقي بيانات الطلب. نلاحظ عندما تُطبع CRLF نرى بداية سطر جديد بدل `\r\n`.

عند ملاحظة سطر البيانات الذي استقبلناه من تنفيذ برنامجنا حتى الآن نرى أن التابع هو `GET` وطلب URI هو / والنسخة هي `HTTP/1.1`.

الأسطر الباقية بدءًا من `Host:` وبعد هي ترويسات. طلب `GET` لا يحتوي متن.

حاول عمل طلب من متصفح آخر أو طلب عنوان مختلف مثل 127.0.0.1:7878/test لترى كيف تتغير بيانات الطلب.

بعد أن عرفنا ماذا يريد المتصفح لنرسل بعض البيانات.

20.1.4 كتابة استجابة

سننفّذ إرسال بيانات مثل استجابة لطلب عميل. لدى الاستجابات التنسيق التالي:

```
HTTP-Version Status-Code Reason-Phrase CRLF
headers CRLF
message-body
```

يحتوي السطر الأول الذي هو سطر الحالة نسخة HTTP المستخدمة في الاستجابة ورمز حالة status code عددية تلخص نتيجة الطلب وعبارة سبب تقدم شرحًا نصيًا عن رمز الحالة. يوجد بعد سلسلة CRLF ترويسات وسلسلة CRLF أُخرى ومتن الاستجابة.

لدينا مثال عن استجابة تستخدم نسخة HTTP 1.1 ولديها رمز حالة 200 وعبارة سبب OK بلا ترويسة أو متن.

```
HTTP/1.1 200 OK\r\n\r\n
```

يُعد رمز الحالة 200 استجابة نجاح قياسية والنص هو استجابة نجاح HTTP صغيرة. لنكتب ذلك إلى المجرى مثل استجابة لطلب ناجح. أزل `println!` التي كانت تطبع طلب البيانات من الدالة `handle_connection` واستبدلها بالشيفرة 3.

اسم الملف: src/main.rs

```
fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    let response = "HTTP/1.1 200 OK\r\n\r\n";

    stream.write_all(response.as_bytes()).unwrap();
}
```

[الشيفرة 3: كتابة استجابة نجاح HTTP صغيرة إلى المجرى]

يعرّف أول سطر المتغير `response` الذي يحتوي بيانات رسالة النجاح، بعدها نستدعي `as_bytes` على `response` الخاص بنا لتحويل بيانات السلسلة النصية إلى بايتات. يأخذ تابع `write_all` على `stream` النوع `&[u8]` ويرسل هذه البايتات مباشرةً نحو الاتصال لأن عملية `write_all` قد تفشل. نستعمل `unwrap` على أي خطأ ناتج كما فعلنا سابقًا. مُجددًا، يجب أن تتعامل مع الأخطاء في التطبيقات الحقيقية.

لننفذ شيفرتنا بعد إجراء التعديلات ونرسل طلبًا. لا نطبع أي بيانات إلى الطرفية لذا لا نرى أي خرج ما عدا خرج Cargo. عند تحميل 127.0.0.1:7878 في متصفح الويب يجب أن يظهر صفحة فارغة بدلًا من خطأ، وبذلك تكون قد شفّرت يدويًا استقبال طلب HTTP وإرسال استجابة.

20.1.5 إعادة HTML حقيقي

لننفّذ وظيفة إعادة أكثر من صفحة فارغة. أنشئ الملف الجديد hello.html في جذر مسار مشروعك وليس في مسار src. يمكنك إدخال أي HTML تريده، تظهر الشيفرة 4 أحد الاحتمالات.

اسم الملف:hello.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Hi from Rust</p>
  </body>
</html>
```

[الشيفرة 4: مثال ملف HTML يعيد استجابة]

يمثّل هذا وثيقة HTML5 بسيطة مع ترويسة وبعض النصوص. سنعدّل الدالة handle_connection لإعادتها من الخادم عندما يُستقبل الطلب، كما في الشيفرة 5 وذلك لقراءة ملف HTML وإضافة الاستجابة مثل متن وإرساله.

اسم الملف: src/main.rs

```
use std::{
    fs,
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
};
// --snip--
```

```
fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    let status_line = "HTTP/1.1 200 OK";
    let contents = fs::read_to_string("hello.html").unwrap();
    let length = contents.len();

    let response =
        format!("{status_line}\r\nContent-Length: {length}\r\n\r\
n{contents}");

    stream.write_all(response.as_bytes()).unwrap();
}
```

[الشيفرة 5: إرسال محتوى hello.html مثل متن الاستجابة]

أضفنا `fs` إلى تعليمة `use` لجلب وحدة نظام ملفات المكتبة القياسية إلى النطاق. يجب أن تكون الشيفرة لقراءة محتوى الملف إلى سلسلةً نصيةً مألوفة، إذ استخدمناها سابقًا في الفصل 12 عندما قرأنا محتوى ملف مشروع I/O في الشيفرة 4.

استخدمنا `format!` لإضافة محتوى الملف على أنه متن استجابة النجاح، أضفنا الترويسة `Content-Length` التي تحدد حجم متن الاستجابة وفي حالتنا حجم `hello.html` لضمان استجابة HTTP صالحة لا.

نفذ هذه الشيفرة مع `cargo run` وحمّل 127.0.0.1:7878 في المتصفح، يجب أن ترى HTML الخاص بك معروضًا.

نتجاهل حاليًا طلب البيانات في `http_request` ونرسل فقط محتوى ملف HTML دون شروط، هذا يعني إذا جربنا طلب 127.0.0.1:7878/something-else في المتصفح سنحصل على نفس استجابة HTML. في هذه اللحظة الخادم محدود ولا يفعل ما يفعله خوادم الويب، ونريد تعديل استجابتنا اعتمادًا على الطلب وإرسال ملف HTML فقط لطلب منسق جيدًا إلى "/".

20.1.6 التحقق من صحة الطلب والاستجابة بصورة انتقائية

يعيد خادم الويب الخاص بنا ملف HTML مهما كان طلب العميل، لنضف وظيفة التحقق أن المتصفح يطلب "/" قبل إعادة ملف HTML وإعادة خطأ في حال طلب المتصفح شيئًا آخر، لذا نحتاج لتعديل `handle_connection` كما في الشيفرة 6. تتحقق هذه الشيفرة الجديدة محتوى الطلب المُستقبل مع ما يشبه طلب "/" وتضيف كتل `if` و `else` لمعالجة الطلبات على نحوٍ مختلف.

اسم الملف: src/main.rs

```
// --snip--

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let request_line = buf_reader.lines().next().unwrap().unwrap();

    if request_line == "GET / HTTP/1.1" {
        let status_line = "HTTP/1.1 200 OK";
        let contents = fs::read_to_string("hello.html").unwrap();
        let length = contents.len();

        let response = format!(
            "{status_line}\r\nContent-Length: {length}\r\n\r\
n{contents}"
        );

        stream.write_all(response.as_bytes()).unwrap();
    } else {
        // some other request
    }
}
```

[الشيفرة 6: معالجة الطلبات إلى / على نحوٍ مختلف عن الطلبات الأُخرى]

سننظر فقط إلى السطر الأول من طلب HTTP لذا بدلًا من قراءة كامل الطلب لشعاع، نستدعي `next` ليأخذ العنصر الأول من المكرّر. تتعامل `unwarp` الأولى مع `Option` وتوقف البرنامج إذا لم يكن للمكرّر أي عنصر؛ بينما تتعامل `unwarp` الثانية مع `Result` ولها نفس تأثير `unwarp` التي كان في `map` المضافة في الشيفرة 2.

نتحقق بعد ذلك من `request_line` لنرى إذا كانت تساوي سطر طلب `GET` إلى المسار"/"؛ فإذا ساوت تعيد كتلة `if` محتوى ملف HTML؛ وإذا لم تساوي، يعني ذلك أننا استقبلنا طلب آخر. سنضيف شيفرة إلى كتلة `else` بعد قليل لاستجابة الطلبات الأخرى.

نفذ هذه الشيفرة واطلب 127.0.0.1:7878، يجب أن تحصل على HTML في hello.html. إذا طلبت أي شيء آخر مثل 127.0.0.1:7878/something-else ستحصل على خطأ اتصال مثل الذي تراه عند تنفيذ الشيفرة 1 و2.

لنضيف الشيفرة في الشيفرة 7 إلى كتلة `else` لإعادة استجابة مع رمز الحالة 404 التي تشير إلى أن محتوى الطلب ليس موجودًا. سنعيد بعض HTML للصفحة لتصّير في المتصفح مشيرةً إلى جواب للمستخدم النهائي.

اسم الملف: src/main.rs

```
    // --snip--
    } else {
        let status_line = "HTTP/1.1 404 NOT FOUND";
        let contents = fs::read_to_string("404.html").unwrap();
        let length = contents.len();

        let response = format!(
            "{status_line}\r\nContent-Length: {length}\r\n\r\
n{contents}"
        );

        stream.write_all(response.as_bytes()).unwrap();
    }
```

[الشيفرة 7: الاستجابة برمز الحالة 404 وصفحة خطأ إذا كان أي شيء عدا / قد طُلب]

لدى استجابتنا سطر حالة مع رمز الحالة 404 وعبارة سبب `NOT FOUND`، يكون متن الاستجابة HTML في الملف 404.html. نحن بحاجة إنشاء ملف 404.html بجانب hello.html لصفحة الخطأ، ويمكنك استخدام أي HTML تريده أو استخدم مثال HTML في الشيفرة 8.

اسم الملف: 404.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
```

```
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Oops!</h1>
    <p>Sorry, I don't know what you're asking for.</p>
  </body>
</html>
```

[الشيفرة 8: محتوى معين لصفحة كي تُرسل مع أي استجابة 404]

شغّل الخادم مجددًا بعد هذه التغيرات. يجب أن يعيد محتوى hello.html عند طلب 127.0.0.1:7878 ويعيد خطأ HTML من 404.html في حال طلب آخر مثل 127.0.0.1:7878/foo.

20.1.7 القليل من إعادة بناء التعليمات البرمجية

في هذه اللحظة لدى كتلتي `if` و `else` الكثير من التكرار، فهما تقرأن الملفات وتكتبان محتوى الملفات إلى المجرى. الفرق الوحيد بينهما هو سطر الحالة واسم الملف. لنجعل الشيفرة أدق بسحب هذه الاختلافات إلى سطري `if` و `else` منفصلين، ليعينان القيم إلى المتغيرين سطر الحالة واسم الملف. يمكننا استخدام هذه المتغيرات دون قيود في الشيفرة لقراءة الملف وكتابة الاستجابة. تظهر الشيفرة 9 الشيفرة المُنتجة بعد استبدال كتل `if` و `else` الكبيرة.

اسم الملف: src/main.rs

```
// --snip--

fn handle_connection(mut stream: TcpStream) {
    // --snip--

    let (status_line, filename) = if request_line == "GET / HTTP/1.1"
{
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    let contents = fs::read_to_string(filename).unwrap();
    let length = contents.len();
```

```
    let response =
        format!("{status_line}\r\nContent-Length: {length}\r\n\r\
n{contents}");

    stream.write_all(response.as_bytes()).unwrap();
}
```

[الشيفرة 9: إعادة بناء التعليمات البرمجية لكتل `if` و `else` لتحتوي فقط على الشيفرة المختلفة بين الحالتين]

تعيد الآن كتلتا `if` و `else` فقط القيم المناسبة لسطر الحالة واسم الملف في الصف. نستخدم بعد ذلك التفكيك لتحديد هذه القيمتين إلى `status_line` و `filename` باستخدام الأنماط في تعليمة `let` كما تحدثنا في الفصل 18.

الشيفرة المتكررة الآن هي خارج كتلتي `if` و `else` وتستخدم المتغيران `status_line` و `filename`. يسهّل هذا مشاهدة الفرق بين الحالتين ولدينا فقط مكان واحد لتعديل الشيفرة إذا أردنا تغيير كيفية قراءة الملفات وكتابة الاستجابة. سيكون سلوك الشيفرة في الشيفرة 9 مثل ماهو في الشيفرة 8.

ممتاز، لديك الآن خادم ويب بسيط في حوالي 40 سطر من شيفرة رست الذي يستجيب لطلب واحد مع صفحة محتوى ويستجيب برمز حالة 404 لكل الطلبات الأُخرى.

ينفذ الخادم حاليًا خيطًا واحدًا، بمعنى أنه يُخدّم طلبًا واحدًا كل مرة. لنفحص تاليًا كيف يمكن لذلك أن يسبب مشكلةً بمحاكاة بعض الطلبات البطيئة، ثم سنعالج هذه المشكلة لكي يعالج الخادم طلبات متعددة بالوقت ذاته؟.

20.2 تحويل خادم ويب ذو خيط وحيد إلى خادم متعدد المهام

يعالج الآن خادم الويب كل طلب بدوره، يعني أنه لن يعالج اتصال ثاني حتى ينتهي من معالجة الطلب الأول. سيصبح التنفيذ التسلسلي أقل كفاءةً كلما زادت الطلبات على الخادم؛ فإذا استقبل الخادم طلبًا يتطلب وقتًا طويلًا لمعالجته ستنتظر الطلبات التالية وقتًا أطول حتى ينتهي الطلب الطويل حتى لو كانت الطلبات التالية تُنفذ بسرعة. يجب حل هذه المشكلة ولكن أولًا لنلاحظها أثناء العمل.

20.2.1 محاكاة طلب بطيء في تنفيذ الخادم الحالي

لنلاحظ كيف يؤثر طلب بطيء المعالجة على الطلبات الأخرى المقدمة إلى تنفيذ الخادم الحالي. تنفذ الشيفرة 10 طلب معالجة إلى /sleep بمحاكاة استجابة بطيئة التي تسبب سكون الخادم لخمس ثوانٍ قبل الاستجابة.

اسم الملف: src/main.rs

```
use std::{
    fs,
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
    thread,
    time::Duration,
};
// --snip--

fn handle_connection(mut stream: TcpStream) {
    // --snip--

    let (status_line, filename) = match &request_line[..] {
        "GET / HTTP/1.1" => ("HTTP/1.1 200 OK", "hello.html"),
        "GET /sleep HTTP/1.1" => {
            thread::sleep(Duration::from_secs(5));
            ("HTTP/1.1 200 OK", "hello.html")
        }
        _ => ("HTTP/1.1 404 NOT FOUND", "404.html"),
    };

    // --snip--
}
```

[الشيفرة 10: محاكاة استجابة بطيئة عن طريق سكون الخادم لخمس ثوان]

بدلنا من `if` إلى `match` إذ لـدينا ثلاث حـالات. يجب أن نطـابق صـراحةً مـع قطعـة من `request_line` لمطابقة النمط مع قيم السلسلة النصية المجردة. لا تُسند `match` ولا تُحصل تلقائيًا كما تفعل توابع المساواة.

تكون الذراع الأولى هي نفس كتلـة `if` من الشـيفرة 9، وتطـابق الـذراع الثانيـة الطلب إلى /sleep ويسـكن الخادم لخمس ثوان عندما يُستقبل الطلب قبل تصيير صـفحة HTML الناجحـة، والـذراع الثالثـة هي نفس كتلـة `else` من الشيفرة 9.

يمكن ملاحظة أن الخادم بدائي، لكن تعالج المكاتب الحقيقية طلبات متعددة بطريقة مختصرة أكثر.

شغل الخادم باستخدام `cargo run`، ثم افتح نافذتي متصفح واحدة من أجل http://127.0.0.1:7878 وأُخرى من أجل http://127.0.0.1:7878/sleep. إذا أدخلت URI / عدة مرات كما سابقًا سترى أنه يستجيب بسرعة، لكن إذا أدخلت sleep/ ومن ثم حمّلت "/" سترى أن "/" ينتظر حتى يسكن `sleep` خمس ثوان كاملة قبل أن يُحمّل.

هناك تقنيات متعددة لتفادي التراكم خلف طلب بطيء، والطريقة التي سنتبعها هي مجمع خيط thread pool.

20.2.2 تحسن الإنتاجية باستخدام مجمع خيط

مجمع خيط thread pool هو مجموعة من الخيوط المُنشأة التي تنتظر معالجة مهمة. عندما يستقبل البرنامج مهمةً، يُعيّن واحد من الخيوط في المجمع لأداء المهمة ومعالجتها وتبقى باقي الخيوط في المجمع متاحةً لمعالجة أي مهمة تأتي أثناء معالجة الخيط الأول للمهمة، وعندما ينتهي الخيط من معالجة المهمة يعود إلى مجمع الخيوط الخاملة جاهزًا لمعالجة أي مهمة جديدة. يسمح مجمع خيط معالجة الاتصالات بصورةٍ متزامنة ويزيد إنتاجية الخادم.

سنحدد عدد الخيوط في المجمع برقم صغير لحمايتنا من هجوم حجب الخدمة Denial of Service -أو اختصارًا DoS. إذا طلبنا من البرنامج إنشاء خيط لكل طلب قادم، يمكن لشخص إنشاء 10 مليون طلب مستهلكًا كل الموارد المتاحة وموقفًا معالجة الطلبات نهائيًا.

بدلًا من إنشاء عدد لا نهائي من الخيوط، سننشئ عددًا محددًا من الخيوط في المجمع، وستذهب الطلبات المرسلة إلى المجمع للمعالجة. يحافظ المجمع على ترتيب الطلبات القادمة في رتل، كل خيط يأخذ طلبًا من الرتل يعالجه ثم يطلب من الرتل طلبًا آخر. يمكن معالجة N طلب بهذه الطريقة، إذ تمثّل N عدد الخيوط. إذا كان كل خيط يعالج طلبًا طويل التنفيذ يمكن أن تتراكم الطلبات في الرتل ولكننا بذلك نكون قد زدنا عدد الطلبات طويلة التنفيذ التي يمكن معالجتها قبل أن نصل إلى تلك المرحلة.

هذه إحدى طرق زيادة إنتاجية خادم ويب، ويمكن استكشاف طرق أُخرى مثل نموذج اشتقاق/جمع fork/join أو نموذج الدخل والخرج للخيط الواحد غير المتزامن single-threaded async I/O أو نموذج الدخل والخرج للخيوط المتعددة غير المتزامن multi-threaded async I/O model. يمكنك قراءة وتنفيذ هذه الحلول إذا كنت مهتمًا بهذا الموضوع وكل هذه الخيارات ممكنة مع لغة برمجية ذات مستوى منخفض مثل رست.

قبل البدء بتنفيذ مجمع خيط، لنتحدث كيف يجب أن يكون استخدام المجمع. عند بداية تصميم الشيفرة، تساعدك كتابة واجهة المستخدم في التصميم. اكتب واجهة برمجة التطبيق API للشيفرة بهيكلية تشبه طريقة استدعائها، ثم نفذ الوظيفة داخل الهيكل بدلًا من تنفيذ الوظيفة ثم بناء واجهة برمجة التطبيق العامة.

سنستخدم طريقة تطوير مُقادة بالمصرف compiler-driven، هذه طريقة مشابهة لاستخدامنا التطوير المُقاد بالاختبار test-driven كما فعلنا في مشروع سابق في الفصل 12. سنكتب الشيفرة التي تستدعي الدالة المُرادة، ومن ثم ننظر إلى الأخطاء من المصرّف لتحديد ماذا يجب أن نغير حتى تعمل الشيفرة. يجب الحديث بدايةً عن التقنيات التي لن نستعملها.

20.2.3 إنشاء خيط لكل طلب

لنلاحظ بدايةً كيف ستكون الشيفرة في حال أنشأنا خيطًا لكل طلب. كما ذكرنا سابقًا، لن تكون هذه خطتنا النهائية وذلك لمشكلة إنشاء عدد لا نهائي من الخيوط ولكنها نقطة بداية لإنشاء خادم متعدد الخيوط قادر على العمل، ثم سنضيف مجمع الخيط مثل تحسين وستكون مقارنة الحلين أسهل. تظهر الشيفرة 11 التغييرات لجعل `main` تُنشئ خيط جديد لمعالجة كل مجرى داخل حلقة `for`.

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        thread::spawn(|| {
            handle_connection(stream);
        });
    }
}
```

[الشيفرة 11: إنشاء خيط جديد لكل مجرى]

كما تعلمنا سابقًا في الفصل 16 من قسم استخدام الخيوط Threads لتنفيذ شيفرات بصورة متزامنة آنيًا، يُنشئ `thread::spawn` خيطًا جديدًا وينفذ الشيفرة في المُغلف في الخيط الجديد. إذا نفذت الشيفرة وحملت "/sleep" في المتصفح ومن ثم "/" في نافذتي متصفح أُخريين ستلاحظ أن الطلبات إلى "/" لا تنتظر "/sleep" لينتهي ولكن كما ذكرنا سابقًا سيطغى هذا على النظام لأننا ننشئ خيوطًا دون حد.

20.2.4 إنشاء عدد محدد من الخيوط

نريد من مجمع الخيط أن يعمل بطريقة مشابهة ومألوفة حتى لا يحتاج التبديل من الخيوط لمجمع خيط أي تعديلات كبيرة للشيفرة التي تستخدمها واجهة برمجة التطبيق. تظهر الشيفرة 12 واجهة افتراضية لهيكل `ThreadPool` الذي نريد استخدامه بدلًا عن `thread::spawn`.

اسم الملف: src/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }
}
```

[الشيفرة 12: واجهة `ThreadPool` المثالية]

استخدمنا `ThreadPool::new` لإنشاء مجمع خيط جديد بعدد خيوط يمكن تعديله وفي حالتنا أربعة. لـدى `pool.execute` واجهة مماثلة للدالة `thread:spawn` في حلقة `for` إذ تأخذ مغلقًا يجب أن ينفـذه المجمـع لكل مجرى. نحتاج لتنفيذ `pool.execute` أن تأخذ مغلقًا وتعطيه لخيط في المجمـع لينفـذه. لن تُصـرّف هـذه الشيفرة ولكن سنجربها كي يدلنا المصرف عن كيفية إصلاحها.

20.2.5 إنشاء مجمع خيط باستخدام التطوير المقاد بالمصرف

أجرِ التغييرات في الشيفرة 12 على الملف src/main.rs واسـتخدم أخطـاء المصـرّف من `cargo check` لقيادة التطوير. هذه أول خطأ نحصل عليه:

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0433]: failed to resolve: use of undeclared type `ThreadPool`
  --> src/main.rs:11:16
   |
|     let pool = ThreadPool::new(4);
   |                ^^^^^^^^^^ use of undeclared type `ThreadPool`

For more information about this error, try `rustc --explain E0433`.
error: could not compile `hello` due to previous error
```

عظيم، يبين هذ الخطأ أننا بحاجة إلى نوع أو وحدة `ThreadPool`. سيكون تنفيذ `Threadpool` الخاص بنـا مستقل عن عمل خادم الويب، لذا لنبدّل الوحـدة المصـرفة `hello` من وحـدة ثنائيـة مصـرفة إلى وحـدة مكتبـة

مصرفة لاحتواء تنفيذ Threadpool. يمكننا بعد ذلك استخدام مكتبة مجمع الخيط المنفصلة لفعل أي عمل نريده باستخدام مجمع خيط وليس فقط لطلبات خادم الويب.

أنشئ src/lib.rs الذي يحتوي التالي، وهو أبسط تعريف لهيكل ThreadPool يمكن الحصول عليه.

اسم الملف: src/lib.rs

```
pub struct ThreadPool;
```

ثم عدل ملف main.rs لجلب ThreadPool من المكتبة إلى النطاق بإضافة الشيفرة التالية في مقدمة الملف src/main.rs.

اسم الملف: src/main.rs

```
use hello::ThreadPool;
```

لن تعمل هذه الشيفرة ولكن لننظر مجددًا إلى الخطأ التالي الذي نريد معالجته:

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no function or associated item named `new` found for
struct `ThreadPool` in the current scope
  --> src/main.rs:12:28
   |
|     let pool = ThreadPool::new(4);
   |                            ^^^ function or associated item not
found in `ThreadPool`

For more information about this error, try `rustc --explain E0599`.
error: could not compile `hello` due to previous error
```

يشير هذا الخطأ أننا نحتاج إلى إنتاج دالة مرتبطة اسمها new من أجل ThreadPool. يمكننا معرفة أن new تحتاج معامل يقبل 4 مثل وسيط ويجب أن يعيد نسخة ThreadPool. لننفذ أبسط دالة new التي تحتوي هذه الصفات characteristics.

اسم الملف: src/lib.rs

```
pub struct ThreadPool;

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
```

```
            ThreadPool
        }
    }
```

اخترنا نوع `usize` للمعامل `size` لأننا نعرف أن العدد السالب للطلبات غير منطقي ونعرف أيضًا أننا سنستخدم 4 ليمثّل عدد العناصر في مجموعة الخيوط وهذا هو عمل نوع `usize` كما تحدثنا سابقًا في قسم "أنواع الأعداد الصحيحة" في الفصل 3 من قسم أنواع البيانات Data Types في لغة رست.

لنتحقق من الشيفرة مجددًا:

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no method named `execute` found for struct `ThreadPool`
in the current scope
  --> src/main.rs:17:14
   |
|          pool.execute(|| {
   |               ^^^^^^^ method not found in `ThreadPool`

For more information about this error, try `rustc --explain E0599`.
error: could not compile `hello` due to previous error
```

يحدث الخطأ الآن لأنه ليس لدينا تابع `execute` على `ThreadPool`. تذكر من قسم "إنشاء عدد محدد من الخيوط" أننا قررنا أن مجمع الخيط يجب أن يكون له واجهة تشبه `thread::spawn`، وقررنا أيضًا أننا سننفذ التابع `execute` ليأخذ المغلف المعُطى له ويعطيه لخيط خامل في المجمع لينفذه.

سنعرّف تابع `execute` على `ThreadPool` ليأخذ المغلف مثل معامل. تذكر من القسم "نقل القيم خارج المغلف وسمات `Fn`" في الفصل 13 أننا بإمكاننا أخذ المغلفات مثل معاملات باستخدام ثلاث سمات هي `Fn` أو `FnMut` أو `FnOnce`. يجب أن نحدد أي نوع مغلف نريد استخدامه هنا، نحن نعرف أننا سنفعل شيئًا يشابه تنفيذ المكتبة القياسية لدالة`thread::spawn`، لذلك دعنا نرى ما هي القيود الموجودة لبصمة `thread::spawn` على معاملاتها. تظهر التوثيقات التالي:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where
        F: FnOnce() -> T,
        F: Send + 'static,
        T: Send + 'static,
```

المعامل F هو الـذي يهمنـا، والمعامـل T متعلـق بالقيمـة المُعـادة ولسـنا مهتمين بهـا. يمكننـا أن نـرى أن spawn تستخدم FnOnce مثل قيد سمة على F، وهذا ما نريده أيضًا لأننا نريـد تمريـر الوسـيط الـذي نأخـذه في execute إلى spawn. يمكننا التأكد أيضًا أن FnOnce هي السـمة المُـراد اسـتخدامها لأن خيـط تنفيـذ الطلب سينفِّذ فقط طلب المغلف مرةً واحدة، والذي يطابق Once في FnOnce.

لدى معامل نوع F أيضًـا قيـد سـمة Send وقيـد دورة حيـاة 'static المفيـدان في حالتنـا؛ فنحن بحاجـة Send لنقل المغلف من خيط لآخر، و 'static لأننا لا نعرف الوقت اللازم ليُنفذ الخيط. لننشئ تابع execute على ThreadPool التي تأخذ معامل معمم للنوع F مع هذه القيود.

اسم الملف: src/lib.rs

```
impl ThreadPool {
    // --snip--
    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
    }
}
```

استخدمنا () بعد FnOnce لأن FnOnce تمثل مغلفًا لا يأخذ معاملات ويعيد نوع الوحدة (). يمكن إهمـال النوع المُعاد من البصمة كما في تعريفات الدالة، ولكن حتى لو لم يوجد أي معاملات نحن بحاجة الأقواس.

هذا هو أبسط تنفيذ لدالة execute، فهي لا تعمل شيئًا، لكننا فقـط بحاجـة أن تُصـرّف شـيفرتنا، لنتحقـق منها مجددًا.

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.24s
```

إنها تُصرّف، لكن لاحظ إذا جربت cargo run وأجريت طلبًا في المتصفح، سترى الأخطـاء في المتصـفح نفسها التي رأيناها في بداية الفصل. لم تستدع المكتبة المغلف المُمرر إلى execute حتى الآن.

> هناك مقولة عن لغات البرمجة ذات المُصرّفات الحازمة مثل هاسكل Haskell ورست وهي "إذا صُرفت الشيفرة فإنها تعمل" ولكن هذه المقولة ليست صحيحة إجمالًا، إذ يُصرّف مشروعنا، لكنه لا يعمل شيئًا إطلاقًا. إذا كنا نريد إنشاء مشروع حقيقي ومكتمل، الآن هو الوقت المثالي لكتابة وحدات اختبار للتحقق من أن الشيفرة تُصرّف ولها السلوك المرغوب.

20.2.6 التحقق من صحة عدد الخيوط في new

لن نغيّر شيئًا للمعاملين new و parameter. لننفذ متن الدوال بالسلوك الذي نريده، ولنبدأ بالدالة new. اخترنا سابقًا نوع غير مؤشر للمعامل size لأن مجمع بعدد خيوط سلبي هو غير منطقي، ولكن مجمع بعدد خيوط صفر ليس منطقيًا أيضًا ولكن unsize صالح. سنضيف الشيفرة التي تتحقق من أن size أكبر من الصفر قبل إعادة نسخة من ThreadPool وجعل البرنامج يهلع إذا حصل على قيمة صفر باستخدام ماكرو assert! كما في الشيفرة 13.

اسم الملف: src/lib.rs

```
impl ThreadPool {
    /// Create a new ThreadPool.
    ///
    /// The size is the number of threads in the pool.
    ///
    /// ## Panics
    ///
    /// The `new` function will panic if the size is zero.
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        ThreadPool
    }

    // --snip--
}
```

[الشيفرة 13: تنفيذ Threadpool::new ليهلع إذا كان size صفر]

أضفنا بعض التوثيق إلى ThreadPool باستخدام تعليقات doc. لاحظ أننا اتبعنا خطوات التوثيق الجيدة بإضافة قسم يستدعي الحالات التي يمكن للدالة أن تهلع فيها كما تحدثنا في الفصل 14. جرب تنفيذ cargo run --open واضغط على هيكل ThreadPool لرؤية كيف تبدو المستندات المُنشأة للدالة new.

يمكننا تغيير new إلى build بدلًا من إضافة ماكرو assert!، ونعيد Result كما فعلنا في Config::build في مشروع الدخل والخرج في الشيفرة 9 في الفصل 12، لكننا قررنا في حالتنا أن إنشاء مجمع خيط بدون أي خيوط هو خطأ لا يمكن استرداده. إذا كنت طموحًا جرب كتابة دالة اسمها build مع البصمة التالية لمقارنته مع الدالة new.

```
pub fn build(size: usize) -> Result<ThreadPool, PoolCreationError> {
```

20.2.7 إنشاء مساحة لتخزين الخيوط

لدينا الآن طريقة لمعرفة أنه لدينا عدد صالح من الخيوط لتخزينها في المجمع، إذ يمكننا إنشاء هذه الخيوط وتخزينها في هيكل `ThreadPool` قبل إرجاعها إلى الهيكل، ولكن كيف نخزن الخيوط؟ لنلاحظ بصمة `thread::spawn`.

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where
        F: FnOnce() -> T,
        F: Send + 'static,
        T: Send + 'static,
```

يُعاد `JoinHandle<T>` من الدالة `spawn`، إذ تمثّل `T` النوع الذي يعيده المغلف. لنستعمل `JoinHandle` أيضًا لنرى ما سيحدث، إذ سيعالج المغلف الذي نمرره إلى مجمع الخيط الاتصال ولا يعيد أي شيء لذا `T` ستكون نوع وحدة `()`.

ستُصرّف الشيفرة في الشيفرة 14 ولكن لا تُنشئ أي خيوط. غيّرنا تعريف `ThreadPool` لتحتوي شعاعًا من نسخة `thread::JoinHandle<()>` وهيأنا الشعاع بسعة `size` وضبطنا حلقة `for` التي تعيد بعض الشيفرة لإنشاء الخيوط وتعيد نسخة `ThreadPool` تحتويهم.

اسم الملف: src/main.rs

```
use std::thread;

pub struct ThreadPool {
    threads: Vec<thread::JoinHandle<()>>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut threads = Vec::with_capacity(size);

        for _ in 0..size {
```

```
            // create some threads and store them in the vector
        }

        ThreadPool { threads }
    }
    // --snip--
}
```

[الشيفرة 14: إنشاء شعاع للهيكل `ThreadPool` الذي يحتوي الخيوط]

جلبنا `std::thread` إلى النطاق في وحدة المكتبة المصرفة لأننا نستخدم `Thread::JoinHandle` بمثابة نوع العنصر في الشعاع في `ThreadPool`. تُنشئ `ThreadPool` شعاعًا جديدًا يحتوي عناصر `size` عندما يُستقبل حجم صالح.

تعمل الدالة `with_capacity` نفس مهام `Vec::new` ولكن بفرق مهم هو أنها تحجز مسبقًا المساحة في الشعاع لأننا نريد تخزين عناصر `size` في الشعاع. إجراء هذا الحجز مسبقًا هو أكثر كفاءة من استخدام `Vec::new` الذي يغير حجمه كلما أُضيفت عناصر.

عندما تنفذ `cargo check` مجدّدًا ينبغي أن تنجح.

20.2.8 هيكل عامل Worker Struct مسؤول عن إرسال شيفرة من مجمع الخيط

تركنا تعليقًا في حلقة `for` متعلق بإنشاء الخيوط في الشيفرة 14. سننظر هنا إلى كيفية إنشاء الخيوط حقيقةً، إذ تؤمن المكتبة القياسية `thread::spawn` بمثابة طريقة لإنشاء الخيوط ويتوقع `thread::spawn` الحصول على بعض الشيفرة لكي ينفذها الخيط بعد إنشائه فورًا، ولكن نريد في حالتنا إنشاء خيوط وجعلهم ينتظرون شيفرةً سنرسلها لاحقًا. لا يقدم تنفيذ المكتبة القياسية للخيوط أي طريقة لعمل ذلك، إذ يجب تنفيذها يدويًا.

سننفذ هذا السلوك عن طريق إضافة هيكلية بيانات جديدة بين `ThreadPool` والخيوط التي ستدير هذه السلوك الجديد، وسندعو هيكل البيانات هذا "العامل Worker" وهو مصطلح عام في تنفيذات مجمّع الخيوط. يأخذ العامل الشيفرة التي بحاجة لتنفيذ وينفّذها في خيط العامل. فكر كيف يعمل الناس في مطبخ المطعم، إذ ينتظر العاملون طلبات الزبائن ويكونوا مسؤولين عن أخذ هذه الطلبات وتنفيذها.

بدلًا من تخزين شعاع نسخة `JoinHandle<()>` في مجمع الخيط، نخزن نسخًا من هيكل `Worker`. يخزن كل `Worker` نسخة `JoinHandle<()>` واحدة، ثم ننفذ تابع على `Worker` الذي يأخذ مغلف شيفرة لينفذه ويرسله إلى خيط يعمل حاليًا لينفذه. سنعطي كل عامل رقمًا معرّفًا `id` للتمييز بين العمال المختلفين في المجمع عند التسجيل أو تنقيح الأخطاء.

هكذا ستكون العملية الجديدة عند إنشاء `ThreadPool`. سننفذ الشيفرة التي ترسل المغلف إلى الخيط بعد إعداد `Worker` بهذه الطريقة:

1. عرّف هيكل `Worker` الذي يحتوي `id` و `JoinHandle<()>`.
2. عدّل `ThreadPool` لتحتوي شعاع من نسخ `Worker`.
3. عرّف دالة `Worker::new` التي تأخذ رقم `id` وتعيد نسخة `Worker` التي تحتوي `id` وخيط مُنشأ بمغلف فارغ.
4. استخدم عداد حلقة `for` لإنشاء `id` وإنشاء `Worker` جديد مع ذلك الرقم `id` وخزن العامل في الشعاع.

إذا كنت جاهزًا للتحدي، جرّب تنفيذ هذه التغييرات بنفسك قبل النظر إلى الشيفرة في الشيفرة 15.

جاهز؟ يوجد في الشيفرة 15 إحدى طرق عمل التعديلات السابقة.

اسم الملف:src/lib.rs

```
use std::thread;

pub struct ThreadPool {
    workers: Vec<Worker>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool { workers }
    }
    // --snip--
}
```

```
struct Worker {
    id: usize,
    thread: thread::JoinHandle<()>,
}

impl Worker {
    fn new(id: usize) -> Worker {
        let thread = thread::spawn(|| {});

        Worker { id, thread }
    }
}
```

[الشيفرة 15: تعديل `ThreadPool` بحيث تحتوي نسخة `Worker` بدلًا من احتواء الخيط مباشرةً]

عدّلنا اسم حقل `ThreadPool` من `threads` إلى `workers` لأنه يحتوي نسخ `Worker` بدلًا من نسخ `JoinHandle<()>`. استخدمنا العداد في حلقة `for` مثل وسيط لدالة `Worker::new` وخزّنا كل `Worker` جديد في شعاع اسمه `workers`.

لا تحتاج الشيفرة الخارجية (كما في الخادم في src/main.rs) أن تعرف تفاصيل التنفيذ بما يتعلق باستخدام هيكل `Worker` داخل `ThreadPool`، لذا نجعل كل من هيكل `Worker` ودالة `new` خاصين private. تستخدم الدالة `Worker::new` المعرّف `id` المُعطى وتخزن نسخة `JoinHandle<()>` المُنشأة عن طريق إنشاء خيط جديد باستخدام مغلف فارغ.

> سيهلع `thread::spawn` إذا كان نظام التشغيل لا يستطيع إنشاء خيط بسبب عدم توفر موارد كافية، وسيؤدي هذا إلى هلع كامل الخادم حتى لو كان إنشاء بعض الخيوط ممكنًا. للتبسيط يمكن قبول هذا السلوك ولكن في تنفيذ مجمع خيط مُنتج ينبغي استخدام `std::thread::Builder` ودالة `spawn` الخاصة به التي تعيد `Result`.

تُصرّف هذه الشيفرة وتخزن عددًا من نسخ `Worker` الذي حددناه مثل وسيط إلى `ThreadPool::new`. لكننا لم نعالج المغلف الذي نحصل عليه في `execute`. لنتعرف على كيفية عمل ذلك تاليًا.

20.2.9 إرسال طلبات إلى الخيوط عن طريق القنوات

المشكلة التالية التي سنتعامل معها هي أن المغلفات المُعطاة إلى `thread::spawn` لا تفعل شيئًا إطلاقًا، وسنحصل حاليًا على المغلف الذي نريد تنفيذه في تابع `execute`، لكن نحن بحاجة لإعطاء `thread::spawn` مغلفًا لينفذه عندما ننشئ كل `Worker` خلال إنشاء `ThreadPool`.

نريد تشغيل هياكل `Worker` التي أنشأناها للبحث عن شيفرة من الرتل في `ThreadPool` وأن ترسل تلك الشيفرة إلى خيطها لينفّذها.

ستكون القنوات التي تعلمناها سابقًا في الفصل 16 -والتي تُعد طريقة بسيطة للتواصل بين خيطين- طريقةً ممتازةً لحالتنا، إذ سنستعمل قناةً لتعمل مثل رتل للوظائف، وترسل `execute` وظيفة من `ThreadPool` إلى نسخة `Worker` التي ترسل بدورها الوظيفة إلى خيطها. ستكون الخطة على النحو التالي:

1. يُنشئ `ThreadPool` قناة ويحتفظ بالمرسل.
2. يحتفظ كل `Worker` بالمستقبل.
3. ننشئ هيكل `Job` جديد يحتفظ بالمغلف الذي نريد إرساله عبر القناة.
4. يرسل تابع `execute` الوظيفة المراد تنفيذها عبر المرسل.
5. سيتكرر مرور `Worker` على المستقبل وينفذ المغلف لأي وظيفة يستقبلها في الخيط.

لنحاول إنشاء قناة في `ThreadPool::new` والاحتفاظ بالمرسل في نسخة `ThreadPool` كما في الشيفرة 16. لا يحتوي هيكل `Job` أي شيء الآن، لكنه سيكون نوع العنصر المُرسل عبر القناة.

اسم الملف: src/lib.rs

```
use std::{sync::mpsc, thread};

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

struct Job;

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);
```

```
            for id in 0..size {
                workers.push(Worker::new(id));
            }

            ThreadPool { workers, sender }
        }
        // --snip--
    }
```

[الشيفرة 16: تعديل `ThreadPool` لتخزين مرسل القناة التي ترسل نسخ `Job`]

أنشــأنا القنــاة الجديــدة في `ThreadPool:new` وجعلنــا المجمــع يحتفــظ بالمرســل. ستصــرّف هــذه الشيفرة بنجاح.

لنجرب تمرير مستقبل القناة إلى كل عامل عنـدما ينشـئ مجمـع الخيـط القنـاة. نعـرف أننـا نريـد اسـتخدام المستقبل في الخيط الذي أنشأه العامل، لذا سنشير إلى معامـل `receiver` في المغلـف بمرجـع reference. لن تُصرّف الشيفرة 17.

اسم الملف: src/lib.rs

```
impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, receiver));
        }

        ThreadPool { workers, sender }
    }
    // --snip--
}
```

```
// --snip--

impl Worker {
    fn new(id: usize, receiver: mpsc::Receiver<Job>) -> Worker {
        let thread = thread::spawn(|| {
            receiver;
        });

        Worker { id, thread }
    }
}
```

[الشيفرة 17: تمرير المستقبل إلى العمال workers]

أجرينا بعض التغييرات الصغيرة والمباشرة، إذ مررنا المستقبل إلى `Worker::new` واستخدمناه داخل المغلف.

عندما نتحقق من الشيفرة سنحصل على هذا الخطأ:

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0382]: use of moved value: `receiver`
  --> src/lib.rs:26:42
   |
|         let (sender, receiver) = mpsc::channel();
   |                      -------- move occurs because `receiver` has
type `std::sync::mpsc::Receiver<Job>`, which does not implement the
`Copy` trait
...
|             workers.push(Worker::new(id, receiver));
   |                                          ^^^^^^^^ value moved
here, in previous iteration of loop

For more information about this error, try `rustc --explain E0382`.
error: could not compile `hello` due to previous error
```

تحاول الشيفرة تمرير `receiver` لنسخ متعددة من `Worker` ولكن هذا لن يعمل كما تتذكر سابقًا من القسم "استخدام ميزة تمرير الرسائل Message Passing لنقل البيانات بين الخيوط Threads" في الفصل 16، إذ أن تنفيذ القناة المقدم من رست هو مُنتجين producer متعددين ومستهلك consumer واحد، وهذا

يعني أنه لا يمكن نسخ الطرف المستهلك من القناة لإصلاح هذا الخطأ ولا نريد أيضًا إرسال رسائل متعددة لمستهلكين متعددين، بل نحتاج قائمة رسائل واحدة مع عمال متعددين لكي تعالج كل رسالة مرةً واحدةً فقط. إضافةً إلى ذلك، يتطلب أخذ وظيفة من رتل القناة تغيير `receiver`، لذا تحتاج الخيوط طريقةً آمنةً لتشارك وتعدل `receiver`، وإلا نحصل على حالات سباق (كما تحدثنا في الفصل السابق المشار إليه).

تذكر المؤشرات الذكية الآمنة للخيوط التي تحدثنا عنها سابقًا في القسم "تزامن الحالة المشتركة Shared-State Concurrency" في الفصل 16؛ فنحن بحاجة لاستخدام `Arc<Mutex<T>>` لمشاركة الملكية لعدد من الخيوط والسماح للخيوط بتغيير القيمة. يسمح نوع `Arc` لعدد من العمال من مُلك المستقبل وتضمن `Mutex` حصول عامل واحد على الوظيفة من المستقبل. تظهر الشيفرة 18 التغييرات التي يجب عملها.

اسم الملف: src/lib.rs

```
use std::{
    sync::{mpsc, Arc, Mutex},
    thread,
};
// --snip--

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool { workers, sender }
    }

    // --snip--
```

```
}

// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) ->
Worker {
        // --snip--
    }
}
```

[الشيفرة 18: مشاركة المستقبل بين العمال باستخدام `Arc` و `Mutex`]

نضع المستقبل في`ThreadPool::new` في `Arc` و `Mutex`، وننسخ `Arc` لكل عامل لتزيد عدّ المرجع ليستطيع العمال مشاركة ملكية المستقبل.

تُصرّف الشيفرة بنجاح مع هذه التغييرات، لقد اقتربنا من تحقيق هدفنا.

20.2.10 تنفيذ تابع التنفيذ execute

لننفذ أخيرًا تابع `execute` على `ThreadPool`، إذ سغيّر أيضًا `Job` من هيكل إلى نوع اسم بديل لكائن السمة الذي يحتوي نوع المغلف الذي يستقبله `execute`. كما تحدثنا في قسم "إنشاء مرادفات للنوع بواسطة أسماء النوع البديلة" في الفصل 19، يسمح لنا نوع الاسم البديل بتقصير الأنواع الطويلة لسهولة الاستخدام كما في الشفرة 19.

اسم الملف: src/lib.rs

```
// --snip--

type Job = Box<dyn FnOnce() + Send + 'static>;

impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
```

```
        let job = Box::new(f);

        self.sender.send(job).unwrap();
    }
}

// --snip--
```

[الشيفرة 19: إنشاء نوع اسم بديل Job لـ Box يحتوي كل مغلف وإرسال العمل عبر القناة]

بعد إنشاء نسخة Job جديدة باستخدام المغلف نحصل على execute ونرسل الوظيفة عبر الطرف المرسل للقناة. نستدعي unwrap على send في حال فشل الإرسال؛ إذ يمكن حصول ذلك إذا أوقفنا كل الخيوط من التنفيذ، وهذا يعني توقُّف الطرف المستقبل عن استقبال أي رسائل جديدة. لا يمكننا الآن إيقاف الخيوط من التنفيذ، إذ تستمر خيوطنا بالتنفيذ طالما المجمع موجود. سبب استخدام unwrap هو أننا نعرف أن حالة الفشل هذه لن تحصل ولكن المصرّف لا يعرف ذلك.

لم ننتهي كليًا بعد، فالمغلف المُمرر إلى thread::spawn يسند الطرف المستقبل من القناة فقط، لكن نريد بدلًا من ذلك أن يتكرر المغلف للأبد ويسأل الطرف المستقبل من القناة عن وظيفة وينفذ الوظيفة عندما يحصل عليها. دعنا نجري التغييرات الموضحة في الشيفرة 20 للدالة Worker::new.

اسم الملف: src/lib.rs

```
// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) ->
Worker {
        let thread = thread::spawn(move || loop {
            let job = receiver.lock().unwrap().recv().unwrap();

            println!("Worker {id} got a job; executing.");

            job();
        });

        Worker { id, thread }
    }
```

```
}
```

[الشيفرة 20: استقبال وتنفيذ الوظائف في خيط العامل]

نستدعي أولًا `lock` على `receiver` للحصول على mutex، ونستدعي `unwrap` ليهلع على أي خطأ. قد يفشل الحصول على قفل إذا كان mutex في حالة مسمومة poisoned، والتي تحصل إذا هلع أحد الخيوط عند احتفاظه بالقفل بدلًا من ترك القفل، وسيكون استدعاء `unwrap` في هذه الحالة هو العمل الأفضل. غيّر `unwrap` إلى `expect` على راحتك لتظهر رسالة خطأ ذات معنى.

إذا حصلنا على القفل على mutex، نستدعي `recv` لاستقبال `Job` من القناة. يتخطى استدعاء `unwrap` الأخير أي أخطاء أيضًا والتي ربما قد تحصل إذا أُغلق، على نحوٍ مشابه لكيفية إعادة `Err` من قِبل تابع `send` إذا أُغلق المستقبل.

إذا لم توجد أي وظيفة في استدعاء كتل `recv`، سينتظر الخيط حتى تتوفر وظيفة. يضمن `Mutex<T>` أن يكون هناك خيط `Worker` واحد يطلب وظيفة.

يعمل مجمع الخيط الآن، جرب `cargo run` وأرسل بعض الطلبات.

```
$ cargo run
   Compiling hello v0.1.0 (file:///projects/hello)
warning: field is never read: `workers`
 --> src/lib.rs:7:5
  |
|     workers: Vec<Worker>,
  |     ^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(dead_code)]` on by default

warning: field is never read: `id`
  --> src/lib.rs:48:5
   |
|     id: usize,
   |     ^^^^^^^^

warning: field is never read: `thread`
  --> src/lib.rs:49:5
   |
|     thread: thread::JoinHandle<()>,
```

```
    |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

warning: `hello` (lib) generated 3 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 1.40s
     Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
```

لقد نجحنا، ولدينا الآن مجمع خيط ينفذ الاتصالات على نحوٍ غير متزامن. لا يُنشئ أكثر من أربعة خيوط حتى لا يُحمَل النظام بصورةٍ زائدة إذا استقبل الخادم طلبات كثيرة. إذا أرسلنا طلبًا إلى sleep/ سيكون الخادم قادرًا على خدمة طلبات أُخرى بجعل خيط آخر ينفذهم.

> إذا فتحنا sleep/ في نوافذ متعددة في المتصفح بنفس الوقت، ستُحمل واحدةٌ تلو الأُخرى بفواصل زمنية مدتها 5 ثواني لأن بعض المتصفحات تنفذ النسخ المتعددة لنفس الطلب بالترتيب لأسباب التخزين المؤقت. ليس الخادم هو سبب هذا التقصير.

بعد أن تعلمنا عن حلقة `while let` في الفصل 18، ربما تتساءل لماذا لم نكتب شيفرة الخيط العامل كما في الشيفرة 21.

اسم الملف: src/lib.rs

```rust
// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) ->
Worker {
        let thread = thread::spawn(move || {
            while let Ok(job) = receiver.lock().unwrap().recv() {
                println!("Worker {id} got a job; executing.");
```

```
                job();
            }
        });

        Worker { id, thread }
    }
}
```

[الشيفرة 21: طريقة تنفيذ مختلفة لدالة `Worker::new` باستخدام `while let`]

تُصرّف الشيفرة وتُنفذ ولكن لا تعطي نتيجة عمل الخيوط المرغوبة، إذ يسبب الطلب البطيء انتظار باقي الطلبات لتُعالج، والسبب بسيطٌ إلى حد ما؛ فليس لدى هيكل `Mutex` دالة `unlock` عامة لأن ملكية القفل مبينةٌ على دورة حياة `MutexGuard<T>` داخل `LockResult<MutexGuard<T>>` التي يعيدها التابع `lock`. يطبق متحقق الاستعارة قاعدة أن المورد المحمي بهيكل `Mutex` لا يمكن الوصول له إلا إذا احتفظنا بالقفل وقت التصريف، ولكن بهذا التنفيذ يمكن أن يبقى القفل مُحتفظًا به أكثر من اللازم إذا لم نكن منتبهين إلى دورة حياة `MutexGuard<T>`.

تعمل الشيفرة في الشيفرة 20 التي تستخدم ما يلي:

```
let job = receiver.lock().unwrap().recv().unwrap();
```

إذ تُسقط أي قيمة مؤقتة مُستخدمة في التعبير على الطرف اليمين من إشارة المساواة "=" مع `let` عندما تنتهي تعليمة `let`، ولكن لا تُسقط `while let` (وأيضًا `if let` و `match`) القيم المؤقتة حتى نهاية الكتلة المرتبطة بها. يبقى القفل مُحتفظًا به حتى نهاية فترة استدعاء `job()` يعني أن العمال الباقين لا يمكن أن يستقبلوا وظائف.

20.3 الإغلاق الرشيق وتحرير الذاكرة

تستجيب الشيفرة 20 للطلبات بصورةٍ غير متزامنة عبر استخدام مجمع خيط كما نريد، إذ نحصل على بعض التحذيرات من حقول `workers` و `id` و `thread` التي لن نستخدمها مباشرةً وتذكرنا أننا لم نحرر أي شيء من الذاكرة. عندما نستخدم الحل البدائي الذي هو استخدام مفتاحي "ctrl-c" لإيقاف الخيط الرئيسي، تتوقف الخيوط مباشرةً حتى لو كانوا يخدّمون طلبًا.

سننفّذ سمة `Drop` لاستدعاء `join` على كل خيط في المجمع لكي ننهي الطلبات التي تعمل قبل الإغلاق، ثم سننفّذ طريقةً لإخبار الخيوط ألا تقبل طلبات جديدة قبل الإغلاق. لرؤية عمل هذا الكود سنعدّل الخادم ليقبل طلبين فقط قبل أن يغلق مجمع الخيط thread pool.

20.3.1 تنفيذ سمة Drop على مجمع خيط

لنبدأ بتنفيذ `Drop` على مجمع الخيط الخاص بنا. عندما يُسقط المجمع يجب أن تجتمع كـل الخيـوط للتأكـد من أن عملهم قد انتهى. تظهر الشيفرة 22 المحاولة الأولى لتطبيق `Drop`، إذ لن تعمل الشيفرة حاليًا.

اسم الملف: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            worker.thread.join().unwrap();
        }
    }
}
```

[الشيفرة 22: ضم كل خيط عندما يخرج المجمع خارج النطاق]

أولًا، نمرّ على كل `workers` في مجمع الخيط، واستُخدمت `&mut` هنا لأن `self` هـو مرجـع متغيّـر، ونريـد أيضًا تغيير `worker`. نطبع لكل عامل رسالةً تقول أن هذا العامل سيُغلق، ثم نستدعي `join` على خيط العمـال. إذا فشل استدعاء `join` نستخدم `unwrap` لجعل رست تهلع وتذهب إلى إغلاق غير رشيق.

سنحصل على هذا الخطأ عند تصريف هذه الشيفرة:

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0507]: cannot move out of `worker.thread` which is behind a
mutable reference
  --> src/lib.rs:52:13
   |
|             worker.thread.join().unwrap();
   |             ^^^^^^^^^^^^^ ------ `worker.thread` moved due to
this method call
   |             |
   |             move occurs because `worker.thread` has type
`JoinHandle<()>`, which does not implement the `Copy` trait
   |
note: this function takes ownership of the receiver `self`, which
moves `worker.thread`
```

```
  -->
/rustc/d5a82bbd26e1ad8b7401f6a718a9c57c96905483/library/std/src/thread
/mod.rs:1581:17


For more information about this error, try `rustc --explain E0507`.
error: could not compile `hello` due to previous error
```

يوضّح الخطأ أننا لا يمكن أن نستدعي `join` لأنه لدينا استعارة متغيرة على كل `worker` وتأخذ `join` ملكية وسطائها، ولمعالجة هذه المشكلة نحن بحاجة لنقل الخيط خارج نسخة `Worker` التي تملك `thread` حتى تستطيع `join` استهلاك الخيط، وقد فعلنا ذلك في الشيفرة 15 من الفصل 17. إذا احتفظ `Worker` بـ `Option<thread::JoinHandle<()>>`، يمكننا استدعاء تابع `take` على `Option` لنقل القيمة خارج المتغاير `Some` وإبقاء المتغاير `None` في مكانه، بمعنى آخر سيحتوي `Worker` عامل على متغاير `Some` في `Worker` الخاص به وعندما نريد تحرير ذاكرة `Worker` نستبدل `Some` بالقيمة `None` حتى لا يوجد لدى أي خيط لينفذه.

لذا نحن نعرف أننا نريد تحديث تعريف `Worker` على النحو التالي.

اسم الملف: src/lib.rs

```
struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}
```

الآن لنتابع المصرّف لنجد أية أماكن أُخرى تحتاج تغيير، وبالتحقق من الشيفرة نجد خطأين:

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no method named `join` found for enum `Option` in the
current scope
  --> src/lib.rs:52:27
   |
|             worker.thread.join().unwrap();
   |                           ^^^^ method not found in
`Option<JoinHandle<()>>`
   |
note: the method `join` exists on the type `JoinHandle<()>`
```

```
  -->
/rustc/d5a82bbd26e1ad8b7401f6a718a9c57c96905483/library/std/src/thread
/mod.rs:1581:5
help: consider using `Option::expect` to unwrap the `JoinHandle<()>`
value, panicking if the value is an `Option::None`
   |
|             worker.thread.expect("REASON").join().unwrap();
   |                           +++++++++++++++++

error[E0308]: mismatched types
  --> src/lib.rs:72:22
   |
|         Worker { id, thread }
   |                      ^^^^^^ expected enum `Option`, found struct
`JoinHandle`
   |
   = note: expected enum `Option<JoinHandle<()>>`
            found struct `JoinHandle<_>`
help: try wrapping the expression in `Some`
   |
|         Worker { id, thread: Some(thread) }
   |                      ++++++++++++      +

Some errors have detailed explanations: E0308, E0599.
For more information about an error, try `rustc --explain E0308`.
error: could not compile `hello` due to 2 previous errors
```

لنعالج الخطأ الثاني الذي يشير إلى الشيفرة في نهاية `Worker::new`، إذ نريـد تغليـف قيمـة `thread` في `Some` عندما ننشئ `Worker` جديد. أجرِ الخطوات التالية لتصحيح هذا الخطأ:

اسم الملف: src/lib.rs

```
impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) ->
Worker {
        // --snip--

        Worker {
            id,
```

```
                thread: Some(thread),
            }
        }
    }
```

الخطأ الأول هو في تنفيذ `Drop`، وذكرنا سابقًا أننا أردنا استدعاء `take` على قيمة `Option` لنقل `thread` خارج `worker`. أجرِ التغييرات التالية لتصحيح هذا الخطأ:

اسم الملف: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}
```

كما تحدثنا سابقًا في الفصل 17، يأخذ التابع `take` على `Option` المتغاير `Some` خارجًا ويبقي `None` بدلًا عنه. استخدمنا `if let` لتفكيك `Some` والحصول على الخيط، ثم استدعينا `join` على الخيط. إذا كان خيط العامل هو أساسًا `None` نعرف أن العامل قد حرَّر ذاكرته ولا يحصل شيء في هذه الحالة.

20.3.2 الإشارة للخيط ليتوقف عن الاستماع إلى الوظائف

تُصرّف الشيفرة بدون تحذيرات بعد كل التغييرات التي أجريناها، ولكن الخبر السيء أنها لا تعمل كما أردنا. النقطة المهمة هي في منطق المغلفات المنفذة بواسطة خيوط نسخ `Worker`، إذ نستدعي حتى اللحظة `join` لكن لا تُغلق الخيوط لأنها تعمل في `loop` للأبد بحثًا عن وظائف. إذا أسقطنا `Threadpool` بتنفيذنا الحالي للسمة `drop`، سيُمنع الخيط الأساسي للأبد بانتظار الخيط الأول حتى ينتهي، ولحل هذه المشكلة نحتاج لتغيير تنفيذ `drop` في `ThreadPool`، ثم إجراء تغيير في حلقة `Worker`.

أولًا، سنغير تنفيذ `drop` في `ThreadPool` ليسقِط صراحةً `sender` قبل انتظار الخيوط لتنتهي. تظهر الشيفرة 23 التغييرات في `ThreadPool` لتسقط صراحةً `sender`. استخدمنا نفس تقنيات`Option` و `take` كما فعلنا مع الخيط لكي يستطيع نقل `sender` خارج `ThreadPool`.

اسم الملف: src/lib.rs

```rust
pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: Option<mpsc::Sender<Job>>,
}
// --snip--
impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        // --snip--

        ThreadPool {
            workers,
            sender: Some(sender),
        }
    }

    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let job = Box::new(f);

        self.sender.as_ref().unwrap().send(job).unwrap();
    }
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
        drop(self.sender.take());

        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
```

```
        }
    }
}
```

[الشيفرة 23: إسقاط `sender` صراحةً قبل جمع الخيوط الفعالة]

يغلق إسقاط `sender` القناة، وهذا يشير بدوره إلى عدم إرسال أي رسائل إضافية، وعندما نفعل ذلك تعيد كل الاستدعاءات إلى `recv` التي تجريها الخيوط الفعالة في الحلقة اللانهائية خطأً. نغير حلقة `Worker` في الشيفرة 24 لتخرج من الحلقة برشاقة في تلك الحالة، يعني أن الخيوط ستنتهي عندما يستدعي `join` عليهم في تنفيذ `drop` في `ThreadPool`.

اسم الملف: src/lib.rs

```
impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) ->
Worker {
        let thread = thread::spawn(move || loop {
            let message = receiver.lock().unwrap().recv();

            match message {
                Ok(job) => {
                    println!("Worker {id} got a job; executing.");

                    job();
                }
                Err(_) => {
                    println!("Worker {id} disconnected; shutting
down.");
                    break;
                }
            }
        });

        Worker {
            id,
            thread: Some(thread),
        }
    }
```

```
}
```

[الشيفرة 24: الخروج صراحةً من الحلقة عندما تعيد `recv` خطأ]

لرؤية عمل هذه الشيفرة: سنعدل `main` لتقبل فقط طلبين قبل أن تُغلق الخادم برشاقة كما تظهر الشيفرة 25.

اسم الملف: src/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}
```

[الشيفرة 25: إغلاق الخادم بعد خدمة طلبين عن طريق الخروج من الحلقة]

لا نريد أن يتوقف خادم حقيقي بعد خدمة طلبين فقط، وتبين هذه الشيفرة أن الإغلاق الرشيق وتحرير الذاكرة يعملان بصورةٍ نظامية.

تُعرّف دالة `take` في سمة `Iterator` وتحدد التكرار إلى أول عنصرين بالحد الأقصى. سيخرج `ThreadPool` خارج النطاق في نهاية `main` وستُطبَّق سمة `drop`.

شغّل الخادم باستخدام `cargo run` وأرسل ثلاثة طلبات. سيعطي الطلب الثالث خطأ وسترى الخرج في الطرفية على النحو التالي:

```
$ cargo run
   Compiling hello v0.1.0 (file:///projects/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 1.0s
     Running `target/debug/hello`
```

```
Worker 0 got a job; executing.
Shutting down.
Shutting down worker 0
Worker 3 got a job; executing.
Worker 1 disconnected; shutting down.
Worker 2 disconnected; shutting down.
Worker 3 disconnected; shutting down.
Worker 0 disconnected; shutting down.
Shutting down worker 1
Shutting down worker 2
Shutting down worker 3
```

يمكن أن ترى ترتيبًا مختلفًا للخيوط الفعالة والرسائل المطبوعة. تعمل الشيفرة وفقًا لهذه الرسائل كما يلي: أخذ العاملان 0 و3 الطلبين الأولين وتوقف الخادم عن قبول الاتصالات بعد ثاني اتصال، وبدأ تنفيذ `Drop` في العمل على `ThreadPool` قبل أخذ العامل 3 وظيفته. يفصل إسقاط `sender` كل العمال ويخبرهم أن يُغلقوا، ويطبع كل عامل رسالةً عندما يُغلقوا ويستدعي مجمع الخيط `join` لانتظار كل خيط عامل لينتهي.

لاحظ ميّزة مهمة في هذا التنفيذ، إذ أسقط `ThreadPool` `sender` وجرّبنا ضم العامل 0 قبل أن يستقبل أي عامل خطأ. لم يتلق العامل 0 أي خطأ من `recv` بعد، لذا تنتظر كتلة الخيط الأساسية أن ينتهي العامل 0. في تلك الأثناء استقبل العامل 3 وظيفة ثم استقبلت كل الخيوط خطأ. ينتظر الخيط الأساسي باقي العمال لينتهوا عندما ينتهي العامل 0. وبحلول هذه النقطة يخرج كل عامل من حلقته ويتوقف.

تهانينا، فقد أنهينا المشروع ولدينا الآن خادم ويب بسيط يستخدم مجمع خيط للاستجابة بصورةٍ غير متزامنة، ونستطيع إجراء إغلاق رشيق للخادم الذي يحرر من الذاكرة كل الخيوط في المجمع.

هذه هي الشيفرة الكاملة بمثابة مرجع.

اسم الملف: src/main.rs

```
use hello::ThreadPool;
use std::fs;
use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;
use std::thread;
use std::time::Duration;

fn main() {
```

```
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK", "hello.html")
    } else if buffer.starts_with(sleep) {
        thread::sleep(Duration::from_secs(5));
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    let contents = fs::read_to_string(filename).unwrap();

    let response = format!(
        "{}\r\nContent-Length: {}\r\n\r\n{}",
        status_line,
        contents.len(),
```

```
            contents
        );

        stream.write_all(response.as_bytes()).unwrap();
        stream.flush().unwrap();
    }
```

اسم الملف: src/lib.rs

```
use std::{
    sync::{mpsc, Arc, Mutex},
    thread,
};

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: Option<mpsc::Sender<Job>>,
}

type Job = Box<dyn FnOnce() + Send + 'static>;

impl ThreadPool {
    /// Create a new ThreadPool.
    ///
    /// The size is the number of threads in the pool.
    ///
    /// # Panics
    ///
    /// The `new` function will panic if the size is zero.
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);
```

```
        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool {
            workers,
            sender: Some(sender),
        }
    }

    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let job = Box::new(f);

        self.sender.as_ref().unwrap().send(job).unwrap();
    }
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
        drop(self.sender.take());

        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}

struct Worker {
```

```
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) ->
Worker {
        let thread = thread::spawn(move || loop {
            let message = receiver.lock().unwrap().recv();

            match message {
                Ok(job) => {
                    println!("Worker {id} got a job; executing.");

                    job();
                }
                Err(_) => {
                    println!("Worker {id} disconnected; shutting
down.");
                    break;
                }
            }
        });

        Worker {
            id,
            thread: Some(thread),
        }
    }
}
```

يمكننا إجراء المزيد إذا أردنا تحسين المشروع، وإليك بعض الأفكار:

- أضِف المزيد من التوثيق إلى `ThreadPool` وتوابعه العامة.
- أضِف بعض الاختبارات لوظيفة المكتبة.
- غيّر الاستدعاءات من `unwrap` إلى معالجة خطأ أكثر متانة.

- استخدم `ThreadPool` لتنفيذ أعمال غير خدمة طلبات الويب.
- ابحث عن وحدة مجمع خيط مصرفة على creats.io ونفـذ خـادم ويب باسـتخدام الوحـدة المصـرفة، ثم قارن واجهة برمجة التطبيقات API والمتانة بينها وبين مجمع الخيط الذي نفذناه.

20.4 خاتمة

عظيم جدًا، فقد وصلنا إلى نهاية السلسلة. نريد أن نشكرك لانضمامك إلينا في هذه الجولة في رســت. أنت الآن جاهز لتنفيذ مشاريع رست ومساعدة الآخرين في مشاريعهم. تذكر أنه هناك مجتمع مرحب من مسـتخدمي رست الذين يحبون المساعدة في أي صعوبة يمكن أن تواجهها في استعمالك رست.

الملحق

يحتوي هذا القسم على مواد مرجعية ستجدها مفيدةً في استخدامك لرست.

الملحق أ: الكلمات المفتاحية

تحتوي القائمة التالية على الكلمات المفتاحية keywords المحجوزة المستخدمة حاليًا أو مستقبلًا من لغة رست البرمجية، وبالتالي لا يمكن استخدامها مثل معرّفات identifiers (إلا في حالة المعرفات الخام raw identifiers التي ناقشناها سابقًا في قسم "المعرفات الخام"). المعرفات هي أسماء دوال function، أو متغيرات variable، أو معاملات parameter، أو حقول هيكلية struct fields، أو وحدات مصرّفة crate، أو ثوابت constant، أو ماكرو Macro، أو قيم ساكنة static values، أو خاصيات attributes، أو أنواع type، أو سمات trait، أو دورات حياة lifetime.

الكلمات المستخدمة حاليا

إليك قائمة من الكلمات المفتاحية المستخدمة حاليًا مع تعريف لوظيفتها.

- `as` - إجراء تحويل أنواع أولي primitive casting أو توضيح سمة معينة تحتوي على عنصر أو إعادة تسمية عنصر ضمن تعليمات `use`.
- `async` - تعيد قيمة `Future` بدلًا من منع الخيط thread الحالي.
- `await` - تعليق التنفيذ حتى تصبح قيمة `Future` جاهزة.
- `break` - الخروج من الحلقة مباشرة.
- `const` - تعريف ثوابت أو مؤشرات خام ثابتة جديدة.

- `continue` - الانتقال إلى تكرار الحلقة التالية.
- `crate` - يشير إلى جذر الوحدة المصرّفة في مسار الوحدة.
- `dyn` - إرسال ديناميكي لكائن سمة trait object.
- `else` - شيفرة برمجية بديلة عن هياكل التحكم في البنية `if` و `if let`.
- `enum` - تعريف التعداد.
- `extern` - ربط دالة أو متغير خارجي.
- `false` - قيمة بوليانية مجرّدة خاطئة.
- `fn` - تعريف دالة أو نوع مؤشر دالة function pointer type.
- `for` - المرور على عناصر من مكرر iterator أو تطبيق سمة أو تحديد دورة حياة ذات مستوى مرتفع.
- `if` - فرع للتحكم بسير البرنامج مبني على قيمة التعبير الشرطي.
- `impl` - تنفيذ وراثة أو وظيفة سمة.
- `in` - جزء من صيغة حلقة `for`.
- `let` - ربط متغير.
- `loop` - التكرار دون شروط.
- `match` - مطابقة قيمة بالأنماط.
- `mod` - تعريف وحدة module.
- `move` - جعل مغلف closure يأخذ ملكية كل ما يلتقطه.
- `mut` - تحديد قابلية التغيير mutability في المراجع أو المؤشرات الخام أو ارتباطات النمط.
- `pub` - تحديد أن العنصر عام public في حقول الهيكل أو كتل `impl` أو الوحدات.
- `ref` - الربط بالمرجع.
- `return` - العودة من الدالة.
- `Self` - نوع اسم بديل للنوع الذي نعرّفه أو ننفّذه.
- `self` - موضوع تابع method subject أو وحدة حالية.
- `static` - متغير عام global أو دورة حياة تبقى طوال مدة تنفيذ البرنامج.

- `struct` - تعريف هيكل.
- `super` - الوحدة الأب للوحدة الحالية.
- `trait` - تعريف سمة.
- `true` - قيمة بوليانية مجردة صحيحة.
- `type` - تعريف نوع الاسم البديل أو النوع المرتبط associated.
- `union` - تحدد الوحدة هي الكلمة المفتاحية الوحيدة التي تستخدم في تصريح الوحدة
- `unsafe` - نحدد الشيفرة أو الدوال أو السمات أو التنفيذات غير الآمنة
- `use` - يجلب الرموز إلى النطاق
- `where` - تحدد المغلف الذي يحتوي النوع
- `while` - التكرار بشرط اعتمادًا على نتيجة التعبير.

الكلمات المفتاحية المحجوزة للاستخدام مستقبلا

هي كلمات مفتاحية ليس لها وظيفة حاليًا ولكنها محجوزة من رست لاحتمال الاستخدام مستقبلًا.

- `abstract`
- `become`
- `box`
- `do`
- `final`
- `macro`
- `override`
- `priv`
- `try`
- `typeof`
- `unsized`
- `virtual`
- `yield`

المعرفات الخام

تسمح لنا صيغة المعرفات الخام باستخدام الكلمات المفتاحية في الأماكن التي لا يمكن استخدامها فيها، ويمكن استخدام المعرفات الخام عن طريق سبق الكلمات المفتاحية بـ `r#`.

مثلًا، ستحصل على خطأ إذا جربت تصريف الدالة التالية التي تستخدم الكلمة المفتاحية `match` في اسمها.

اسم الملف: src/main.rs

```
fn match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}
```

وسيكون الخطأ كما يلي:

```
error: expected identifier, found keyword `match`
 --> src/main.rs:4:4
  |
| fn match(needle: &str, haystack: &str) -> bool {
  |    ^^^^^ expected identifier, found keyword
```

يظهر الخطأ أنه لا يمكن استخدام `match` مثل معرّف دالة، إذ تحتاج لصيغة المعرف الخام لاستخدام `match` اسمًا لدالة كما يلي.

اسم الملف: src/main.rs

```
fn r#match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}

fn main() {
    assert!(r#match("foo", "foobar"));
}
```

ستُصرّف الشيفرة بدون أي أخطاء، لاحظ وجود سابقة `r#` على اسم الدالة في تعريفها أيضًا وأيضًا أينما تُستدعى الدالة في `main`.

تسمح المعرفات الخام باستخدام أي كلمة تريدها مثل معرف حتى لو كانت محجوزة لكلمة مفتاحية، ويمنحك ذلك حريةً أكبر في اختيار اسم المعرف والعمل مع برامج مكتوبة بلغة برمجية لا تستخدم تلك الكلمات

مثل كلمات مفتاحية، إضافةً إلى السماح باستخدام مكتبات مكتوبة باستخدام إصدارات رست مختلفة عن التي تستخدمها الوحدات المصرّفة، مثلًا كلمة `try` ليست كلمة مفتاحية في إصدار 2015 ولكنها كذلك في إصدار 2018؛ فإذا كنت تعتمد على مكتبة مكتوبة بإصدار 2015 ولديها تابع `try` فستحتاج لاستخدام صيغة معرف خام في حالتنا `r#try` لاستدعاء الدالة شيفرة إصدار 2018. ألقِ نظرةً على الملحق "ج" للمزيد من المعلومات عن الإصدارات.

الملحق ب : المعاملات والرموز

يحتوي هذا الملحق مجموعةً من صياغات رست، تتضمن العوامل operators ورموزًا أُخرى قد تظهر لوحدها أو في سياق المسارات والأنواع المعممّة generics وقيود السمة والماكرو والتعليقات والصفوف والأقواس.

العوامل

يحتوي الجدول ب-1 العوامل في رست وكيفية ظهور العامل في السياق وشرح بسيط عنه وإذا كان العامل قابل لزيادة التحميل؛ فإذا كان كذلك يوجد في الجدول السمة المستخدمة لزيادة تحميل العامل.

[الجدول ب-1:]

العامل	المثال	الشرح	قابلية زيادة التحميل
`!`	`ident!(...)` أو `ident!{...}` أو `ident![...]`	تمديد الماكرو	
`!`	`!expr`	عملية ثنائية أو متمم منطقي	`Not`
`!=`	`expr != expr`	مقارنة غير متساوية	`PartialEq`
`%`	`expr % expr`	باقي حسابي	`Rem`
`%=`	`var %= expr`	باقي حسابي وإسناد	`RemAssign`
`&`	`&expr, &mut expr`	استعارة	
`&`	`type, &mut type, &'a type, &'a mut type`	نوع مؤشر استعارة	
`&`	`expr & expr`	عملية ثنائية "و"	`BitAnd`
`&=`	`var &= expr`	عملية ثنائية "و" وإسناد	`BitAndAssign`
`&&`	`expr && expr`	قصر دارة منطقي "و"	
`*`	`expr * expr`	ضرب حسابي	`Mul`
`*=`	`var *= expr`	ضرب حسابي وإسناد	`MulAssign`
`*`	`*expr`	تحصيل	`Deref`

*	*const type, *mut type	مؤشر خام	
+	trait + trait, 'a + trait	نوع قيد مركب	
+	expr + expr	جمع حسابي	Add
+=	var += expr	جمع حسابي وإسناد	AddAssign
,	expr, expr	فاصل بين الوسيط والعنصر	
-	- expr	سالب حسابي	Neg
-	expr - expr	طرح حسابي	Sub
-=	var -= expr	طرح حسابي وإسناد	SubAssign
->	fn(...) -> type, |...| -> type	نوع إرجاع الدالة والمغلف	
.	expr.ident	عضو وصول	
..	..,expr.., ..expr, expr..expr	مجال مجرد خاص لليمين	PartialOrd
..=	..=expr, expr..=expr	مجال مجرد يضم اليمين	PartialOrd
..	..expr	تحديث صياغة هيكل مجرد	
..	variant(x, ..), struct_type { x, .. }	نمط ارتباط "والباقي"	
...	expr...expr	(مُهمل، استخدم ..= بدلًا عنه) مجال نمط ضمني في الأنماط	
/	expr / expr	قسمة حسابية	Div
/=	var /= expr	قسمة حسابية وإسناد	DivAssign
:	pat: type, ident: type	قيود	
:	ident: expr	مُهيئ حقل الهيكل	
:	'a: loop{...}	عنوان حلقة	
;	expr;	تعبير وإنهاء عنصر	
;	[...; len]	جزء من صياغة مصفوفة بحجم ثابت	
<<	expr << expr	انتقال لليسار	Shl
<<=	var <<= expr	انتقال لليسار وإسناد	ShlAssign
<	expr < expr	مقارنة أقل من	PartialOrd

`<=`	`expr <= expr`	مقارنة أقل من أو يساوي	`PartialOrd`
`=`	`var = expr, ident = type`	إسناد/تساوي	
`==`	`expr == expr`	مقارنة مساواة	`PartialEq`
`=>`	`pat => expr`	جزء من صياغة ذراع مطابقة	
`>`	`expr > expr`	مقارنة أكبر من	`PartialOrd`
`>=`	`expr >= expr`	مقارنة أكبر من أو يساوي	`PartialOrd`
`>>`	`expr >> expr`	انتقال لليمين	`Shr`
`>>=`	`var >>= expr`	انتقال لليمين وإسناد	`ShrAssign`
`@`	`ident @ pat`	ارتباط الأنماط	
`^`	`expr ^ expr`	عملية ثنائية "أو" خاصة	`BitXor`
`^=`	`var ^= expr`	عملية ثنائية "أو" خاصة وإسناد	`BitXorAssign`
`|\`	`pat | pat`	بدائل النمط	
`|`	`expr | expr`	عملية ثنائية "أو"	`BitOr`
`|=`	`var |= expr`	عملية ثنائية "أو" خاصة وإسناد	`BitOrAssign`
`||`	`expr || expr`	قصر دارة منطقي "أو"	
`?`	`?expr`	نشر الخطأ	

الجدول 3: العوامل

20.4.1 الرموز التي ليست بعوامل

تحتوي القائمة التالية كل الرموز التي لا تعمل مثل عوامل أي أنها لا تتصرف مثل دوال أو استدعاء لتابع.

يظهر الجدول ب-2 الرموز التي تظهر لوحدها وهي صالحة في عدد من الأماكن.

الرمز	الشرح
`'ident`	اسم دورة حياة أو تسمية حلقة
`...u8` أو `...i32` أو `...f64` أو `...unsize` ...إلخ.	قيمة رقمية مجرّدة أو نوع محدد
`"…"`	سلسلة نصية مجردة
`r"..."` أو `r#"..."#` أو `r##"..."##` إلخ	سلسلة نصية مجردة خام أو رموز هاربة غير معالجة
`b"…"`	سلسلة بايت نصية مجردة تنشئ مصفوفة من البايتات بدلًا من سلسلة نصية
`br"…"` أو `br#"…"#` أو `br##"…"##` ...إلخ.	سلسلة بايت نصية مجردة خام، مجموعة سلسلة نصية خام وبايتات
`'…'`	محرف مجرد
`b'…'`	بايت ASCII مجرد
`|…| expr`	مغلف
`!`	نوع أسفل فارغ دائمًا لتباعد الدوال
`_`	ارتباط نمط "مُتجاهل" يُستخدم أيضًا لجعل الأعداد الصحيحة المجردة مقروءة

الجدول 4: الصياغة الوحيدة

يظهر الجدول ب-3 الرموز التي تظهر في سياق المسار خلال الترتيب الهرمي لوحدة العنصر.

الرمز	الشرح
`ident::ident`	مسار مكان الاسم
`::path`	مسار متعلق بجذر الوحدة المصرفة (مثال: مسار مطلق حصرًا)
`self::path`	مسار متعلق بالوحدة الحالية
`super::path`	مسار متعلق بأب الوحدة الحالية
`type::ident` أو `<type as trait>::ident`	ثوابت ودوال وأنواع مرتبطة
`<type>::...`	عناصر مرتبطة لنوع لا يمكن تسميته مباشرة (مثال: `<T&>::...` أو `<[T]>::...` إلخ)
`trait::method(…)`	توضيح استدعاء تابع عن طريق تسمية السمة التي تعرفه
`type::method(…)`	توضيح استدعاء تابع بتسمية النوع الذي يُعرّف لأجله
`<type as trait>::method(…)`	توضيح استدعاء تابع بتسمية السمة والنوع

الجدول 5: الصياغة المتعلقة بالمسار

يظهر الجدول ب-4 الرموز التي تظهر في سياق استخدام معاملات النوع المعمم.

الرمز	الشرح
`path<…>`	يخصص معاملًا لتوع معمم في النوع (مثال: `Vec<u8>`)
`path::<…>` أو `method::<…>`	يخصص معاملًا لنوع أو دالة أو تابع معممين في التعبير، يقال له غالبًا turbofish (مثال: `"42".parse::<i32>()`)
`fn ident<…> …`	يعرف تابع معمم
`struct ident<…> …`	يعرف هيكل معمم
`enum ident<…> …`	يعرف تنفيذ معمم
`impl<…> …`	قيد مرتبة عالية لدورة الحياة
`type<ident=type>`	نوع معمم حيث لواحد أو أكثر من الأنواع المرتبطة إسناد مخصص (مثال: `Iterator<Item=T>`)

الجدول 6: الأنواع المعممة

يظهر الجدول ب-5 الرموز التي تظهر في سياق تقييد معاملات النوع المعمم مع قيد سمة.

الرمز	الشرح
`T: U`	معامل معمم `T` مقيد للنوع الذي يُنفذ `U`
`T: 'a`	يجب على المعامل المعمم `T` أن يبقى بعد دورة حياة `'a` (يعني أنه لا يجب للنوع أن يحوي أي مرجع بدورة حياة أقصر من `'a`)
`T: 'static`	لا يحتوي المعامل معمم `T` أي مرجع عدا `'static`
`'d: 'a`	يجب على المعامل المعمم `'b` أن يبقى بعد دورة حياة `'a`
`T: ?Sized`	السماح للنوع المعمم أن يصبح نوع يتغير حجمه ديناميكيًا
`'a + trait` أو `trait + trait`	قيد المعامل المركب

الجدول 7: قيود قيد السمة

يظهر الجدول ب-6 الرمز التي تظهر في سياق استدعاء أو تعريف الماكرو وتحديد خاصيّة عنصر.

الرمز	الشرح
`#[meta]`	خاصيّة خارجية
`#![meta]`	خاصيّة داخلية
`$ident`	تبديل الماكرو
`$ident:kind`	التقاط الماكرو
`$(…)…`	تكرار الماكرو
`ident!(…)` أو `ident!{…}` أو `ident![…]`	استدعاء الماكرو

الجدول 8: الماكرو والخاصيّات

يظهر الجدول ب-7 الرموز التي تُنشئ تعليقات:

الرمز	الشرح
`//`	تعليق سطر
`//!`	تعليق doc لسطر داخلي
`///`	تعليق doc لسطر خارجي
`/*…*/`	تعليق كتلة
`/*!...*/`	تعليق doc لكتلة داخلية
`/**…*/`	تعليق doc لكتلة خارجية

الجدول 9: التعليقات

يظهر الجدول ب-8: الرموز التي تظهر في سياق استخدام الصفوف:

الرمز	الشرح
`()`	صف فارغ (وحدة) مجردة ونوع
`(expr)`	تعبير بين قوسين
`(expr,)`	تعبير صف بعصر واحد
`(type)`	نوع صف بعنصر واحد
`(expr, …)`	تعبير صف
`(type, …)`	نوع صف
`(expr(expr, …)`	تعبير استدعاء دالة، يُستخدم أيضًا لتهيئة صف `struct` ومتغايرات `enum`
`expr.0` أو `expr.1` ...إلخ.	دليل الصف

الجدول 10: الصفوف

يظهر الجدول ب-9: محتوى الذي يُستخدم فيه الأقواس المعقوصة.

السياق	الشرح
`{…}`	تعبير كتلة
`Type {…}`	`struct` مجردة

الجدول 11: الأقواس المعقوصة

يظهر الجدول ب-10 السياق الذي يُستخدم به الأقواس المعقوفة:

السياق	الشرح
`[…]`	مصفوفة مجردة
`[expr; len]`	مصفوفة مجردة تحتوي نسخ `len` من `expr`
`[type; len]`	مصفوفة نوع تحتوي نسخ `len` من `type`
`expr[expr]`	فهرسة مجموعة. قابلة لزيادة التحميل (`Index` و `IndexMut`)
`expr[..]` أو `expr[a..]` أو `expr[..b]` أو `expr[a..b]`	فهرسة مجموعة تدّعي بكونها مجموعة تقطيع، باستخدام `Range`و `RangeFrom` و `RangeTo` و `RangeFull` مثل "فهرس index"

الجدول 12: الأقواس المعقوفة

20.5 الملحق ج: السمات القابلة للاشتقاق

تحدثنا في أماكن مختلفة ضمن الكتاب عن خاصيّة `derive` التي يمكن تطبيقها على تعاريف الهياكل والمعدّدات، إذ تُنشئ خاصيّة `derive` شيفرةً تنفّذ سمةً trait مع تنفيذها القياسي الخاص على النوع الذي وصَّفته بصيغة `derive`.

سنقدم في هذا الملحق مرجعًا لكل السمات في المكتبة القياسية التي يمكن استخدام `derive` معها، سيغطي كل قسم:

- ماذا سيُمكّن اشتقاق المعاملات والتوابع على هذه السمة.
- ماذا تفعل تطبيقات السمة المقدمة من `derive`.
- ماذا يعني تطبيق السمة على النوع.
- الظروف التي تسمح ولا تسمح بتنفيذ السمة.
- أمثلة عن عمليات تحتاج السمة.

راجع توثيق المكتبة القياسية لكل سمة إذا أردت سلوكًا مختلفًا من السلوك المُقدم في خاصيّة `derive` لتعرف كيفية تطبيقها يدويًا.

السمات المذكورة هنا مقتصرة على السمات المعرّفة في المكتبة القياسية التي يمكن تنفيذها على الأنواع باستخدام `derive`، إذ ليس للسمات الأُخرى المعرّفة في المكتبة القياسية سلوك افتراضي معقول لذلك لك حرية تنفيذها بطريقة تجعلها منطقية لما تريد فعله.

مثال على السمات التي يمكن اشتقاقها هي `Display` التي تتولى تنسيق الخرج للمستخدم النهائي، إذ يجب عليك دائما التفكير بالطريقة المناسبة لإظهار النوع للمستخدم النهائي، وما هي الأجزاء التي يجب أن يراها المستخدم النهائي من النوع؟ ما هي الأجزاء التي يجدها ملائمة؟ ما هو تنسيق البيانات التي تكون أكثر ملائمة لهم؟ ليس لمصرّف رست القدرة على تحديد ذلك، لذا يجب عليك تحديد السلوك الافتراضي المناسب لك بنفسك.

ليست قائمة السمات المشتقة المقدمة هنا شاملة، إذ يمكن للمكتبات تنفيذ `derive` على سماتها الخاصة مما يجعل قائمة السمات التي يمكن استخدام `derive` معها لا نهائية. يتضمن تنفيذ `derive` استخدام ماكرو إجرائي تحدثنا عنه سابقًا في القسم "ماكرو" في الفصل 19.

20.5.1 استخدام السمة Debug لخرج المستخدم

تسمح سمة `Debug` بالعثور على أخطاء التنسيق في تنسيق السلاسل النصية الذي يُشار إليه بإضافة `:?` داخل الموضع المؤقت `{}`؛ كما تسمح سمة `Debug` بطبع نسخ من النوع لأغراض العثور على الأخطاء، بحيث يسمح لمبرمجين آخرين يستخدمون النوع بالتدقيق في نسخة في وقت معين من تنفيذ البرنامج.

سمة `Debug` مطلوبة عند استخدام ماكرو `assert_eq!` مثلًا، والذي يطبع قيم نسخ مُعطاة مثل وسطاء، إذا فشل تأكيد المساواة لكي يرى المبرمج سبب عدم تساوي النسختين.

20.5.2 سمتا PartialEq و Eq للمقارنة بين المساواة

تسمح سمة `PartialEq` بمقارنة نسخ النوع للتحقق من المساواة، وتسمح باستخدام المعاملين `==` و `!=`.

ينفّذ اشتقاق `PartialEq` التابع `eq` وعندما تُشتق `PartialEq` على الهياكل تكون النسختان متساويتين فقط إذا تساوت كل الحقول، وتكونان غير متساويتين إذا كانت واحدة من الحقول غير متساوية. عندما يُنفّذ الاشتقاق على المعدّدات يكون كل متغاير variant مساويًا لنفسه وليس مساويًا لباقي المتغايرات.

سمة `PartialEq` مطلوبة مثلًا في استخدام ماكرو `assert_eq!` الذي يحتاج أن يكون قادرًا على التحقق من تساوي نسختين من النوع.

ليس لسمة `Eq` توابعًا هدفها أن تشير إلى أن كل القيمة في النوع الموصف تساوي نفسها، وتُنفّذ سمة `Eq` فقط على الأنواع التي تنفّذ أيضًا `PartialEq` على الرغم من أنه ليس ضروريًا أن يستطيع كل نوع ينفّذ

`PartialEq` تنفيذ `Eq`. مثالٌ على ذلك هو نوع الأعداد العشرية، إذ يشير تنفيذ الأعداد العشرية إلى عدم تساوي نسختين من قيم ليست رقمًا not-a-number أي (`NaN`).

تكون `Eq` مطلوبة مثلًا من أجل المفاتيح في `HashMap<K, V>`، لذا تستطيع `HashMap<K, V>` معرفة إذا كان المفتاحان متشابهين.

20.5.3 سمتا PartialOrd و Ord لمقارنة الترتيب

تسمح سمة `PartialOrd` بالمقارنة بين نسخ النوع لأغراض الترتيب، إذ يمكن لنوع ينفّذ `PartialOrd` أن يُستخدم مع العوامل `<` و `>` و `=<` و `=>`. يمكن تنفيذ سمة `PartialOrd` على الأنواع التي تنفّذ `PartialEq` أيضًا.

ينفّذ اشتقاق `PartialOrd` تابع `partial_cmp` الذي يعيد `Option<Ordering>`، والتي ستكون مساويةً إلى `None` عندما لا تكون القيمة المعطاة مرتبة، ويُعد العدد العشري مثالًا على قيمة لا تُنتج ترتيبًا حتى لو كانت معظم قيم النوع قابلة للمقارنة في قيمة ليس رقم NaN. يعيد استدعاء `partial_cmp` مع أي عدد عشري وقيمة العدد العشري `NaN` القيمة `None`.

تقارن `PartialOrd` عندما تُشتق من الهياكل نسختين عن طريق مقارنة القيم في كل حقل بنفس الترتيب الذي تظهر فيه الحقول في تعريف الهيكل. تعدّ متغايرات المعدّد المصرح عنها أولًا أقل أهمية من المتغايرات المذكورة لاحقًا عند اشتقاقها على المعدّدات.

تكون سمة `PartialOrd` مطلوبة مثلًا في تابع `gen_range` من الوحدة المصرّفة `rand` التي تُنشئ قيمةً عشوائيةً في المجال المحدد بتعبير المجال.

تسمح سمة `Ord` أن تعرِف بأنه يوجد ترتيب صالح لأي قيمتين موصّفتين. تنفّذ سمة `Ord` التابع `cmp` الذي يعيد `Ordering` بدلًا من `Option<Ordering>` لأنه سيكون هناك طريقة ترتيب صالحة. يمكن تنفيذ سمة `Ord` على الأنواع التي تنفذ `PartialOrd` و `Eq` (تحتاج `Eq` إلى `PartialEq`). تتصرف `cmp` عندما تُشتق على الهياكل والمعدّدات بنفس الطريقة التي يفعلها التنفيذ المُشتق `partial_cmp` مع `PartiaOrd`.

مثال عندما تكون `Ord` مطلوبة هو عند تخزين القيم في `BTreeSet<T>` الذي هو هيكلٌ يخزن بيانات اعتمادًا على طريقة ترتيب القيم.

20.5.4 استخدام Clone و Copy لنسخ القيم

تسمح سمة `Clone` بالنسخ العميق للقيم صراحةً، وقد تتضمن طريقة النسخ تنفيذ شيفرة برمجية عشوائية أو نسخ بيانات من الكومة heap. راجع قسم " طرق التفاعل بين المتغيرات والبيانات: الاستنساخ" في الفصل 4 لمعلومات أكثر عن `Clone`.

ينفّذ اشتقاق Clone التابع clone الذي يستدعي clone على كل جزء من النوع عندما يُنفّذ على كل النوع، وهذا يعني أن كل الحقول أو القيم في النوع يجب أن تنفّذ Clone لاشتقاق Clone.

مثال عندما تكون Clone مطلوبة هي عند استدعاء تابع to_vec على شريحة، إذ لا تملك الشريحة نسخة النوع الذي يحتويها ولكن يحتاج الشعاع المُعاد من to_vec النسخ الخاصة به، لذلك تستدعي to_vec التابع clone على كل عنصر، وبالتالي يجب على كل نوع مخزن في الشريحة تنفيذ Clone.

تسمح سمة Copy بنسخ الأجزاء المخزنة على المكدّس وليست بحاجة لشيفرة عشوائية. راجع قسم "بيانات الخاصة بالمكدّس :نسخ" في الفصل 4 لمعلومات أكثر عن Copy.

لا تعرِّف سمة Copy أي توابع لمنع المبرمجين من التحميل الزائد لهذه التوابع وانتهاك الافتراض الذي ينص على عدم تنفيذ أية شيفرة عشوائية، وبذلك يمكن أن يفترض المبرمج أن نسخ القيمة سيكون سريعًا.

يمكن اشتقاق Copy على أي نوع تنفّذ كل أجزائه Copy، إذ يجب على النوع الذي ينفّذ Copy تنفيذ Clone أيضًا لأن النوع الذي ينفّذ Copy لديه تنفيذ عديم الأهمية من Clone التي تنفّذ مهام Copy ذاتها.

نادرًا ما تكون سمة Copy مطلوبة؛ إذ تتوافر لدى الأنواع التي تنفّذ Copy تحسينات، أي لسنا بحاجة استدعاء clone الذي يجعل الشيفرة أكثر اختصارًا.

يمكن تنفيذ كل شيء ممكن مع Copy مع Clone ولكن ستكون الشيفرة أبطأ أو يجب عليك استخدام clone.

20.5.5 سمة Hash لربط القيمة مع قيمة بحجم ثابت

تسمح سمة Hash أخذ نسخة من نوع بحجم عشوائي وربطها بقيمة ذات حجم ثابت باستخدام دالة تعمية. ينفّذ اشتقاق Hash التابع hash، ويجمع التنفيذ المشتق لتابع hash القيم الناتجة عن استدعاء Hash على كل جزء من النوع، وهذا يعني أن كل الحقول أو القيم يجب أن تنفّذ Hash لاشتقاق Hash.

مثال عن كون Hash مطلوبة هي عند تخزين المفاتيح في HashMap<K, V> لتخزين البيانات بكفاءة.

20.5.6 سمة Default للقيم الافتراضية

تسمح سمة Default بإنشاء قيم افتراضية للنوع، وينفّذ اشتقاق Default دالة default، بحيث يستدعي التنفيذ المُشتق لدالة default الدالة default على كل جزء من النوع، ما يعني أنه يجب على كل الحقول والقيم أن تنفّذ Default لتشتق Default.

تُستخدم دالة Default::default عادةً مع صيغة تحديث الهيكل التي تحدثنا عنها في القسم "إنشاء نسخ من نسخ أُخرى باستخدام صيغة تحديث الهيكل" في الفصل 5. يمكنك تعديل بعض حقول الهيكل، ثم ضبط واستخدام قيمة افتراضية مع باقي الحقول باستخدام ..Default::default().

تكون السمة `Default` مطلوبة عندما تستخدم التابع `unwrap_or_default` على نسخ `Option<T>` مثلًا؛ فإذا كانت `Option<T>` هي `None` يعيد التابع `unwrap_or_default` القيمة `Default::default` للنوع `T` المخزن في `Option<T>`.

20.6 الملحق د: أدوات تطوير مفيدة

سنتحدث في هذا الملحق عن أدوات التطوير المفيدة التي يقدمها مشروع رست، إذ سنتحدث عن التنسيق التلقائي وطرق سريعة لتطبيق حلول لبعض التحذيرات ومنقّح الصياغة linter والدمج مع بيئات التطوير المتكاملة IDEs.

20.6.1 التنسيق التلقائي باستخدام rustfmt

تُعيد أداة `rustfmt` تنسيق الشيفرة حسب أسلوب تنسيق الشيفرة الشائع في مجتمع رست، إذ تستخدم العديد من المشاريع المشتركة `rustfmt` للتخلص من الجدال المتعلق بطريقة التنسيق المُستخدمة أثناء كتابة الشيفرة في رست، ويستخدم الكل هذه الأداة لتنسيق الشيفرة الخاصة بهم.

اكتب الأمر التالي لتثبيت `rustfmt`:

```
$ rustup component add rustfmt
```

يعطيك هذا الأمر `rustfmt` و `cargo-fmt` بطريقة مماثلة عندما تعطيك رست `rustc` و `cargo`، اكتب الأمر التالي لتنسيق أي مشروع كارجو:

```
$ cargo fmt
```

يعيد تنفيذ هذا الأمر تنسيق كل شيفرة رست في الوحدة المصرفة الحالية، ويغير فقط تنسيق الشيفرة وليس دلالات الشيفرة. للمزيد من المعلومات عن `rustfmt` راجع توثيقاتها.

20.6.2 أصلح شيفرتك البرمجية باستخدام rustfix

أداة rustfix مُضمّنة مع تثبيت رست ويمكنها إصلاح تحذيرات المصرّف التي يمكن إصلاحها بوضوح تلقائيًا وهذا شيء مرغوب. لربما لاحظت تحذيرات المصرّف سابقًا، لاحظ مثلًا هذه الشيفرة:

اسم الملف: src/main.rs

```
fn do_something() {}

fn main() {
    for i in 0..100 {
```

```
        do_something();
    }
}
```

استدعينا هنا الدالة `do_somthing` مائة مرة لكننـا لم نسـتخدم المتغـير `i` في متن الحلقـة `for`، وتحـذرنا رست بخصوص ذلك:

```
$ cargo build
   Compiling myprogram v0.1.0 (file:///projects/myprogram)
warning: unused variable: `i`
 --> src/main.rs:4:9
  |
|     for i in 0..100 {
  |         ^ help: consider using `_i` instead
  |
  = note: #[warn(unused_variables)] on by default

    Finished dev [unoptimized + debuginfo] target(s) in 0.50s
```

يقترح التحذير أن نستخدم `_i` اسمًا، إذ تشـير الشـرطة السـفلية underscore أننـا أردنـا ألا يُسـتخدم هـذا المتغير، ويمكننا تطبيق ذلك الاقتراح تلقائيًا باستخدام أداة `rustfix` عن طريق تنفيذ الأمر `cargo fix`:

```
$ cargo fix
    Checking myprogram v0.1.0 (file:///projects/myprogram)
      Fixing src/main.rs (1 fix)
    Finished dev [unoptimized + debuginfo] target(s) in 0.59s
```

عندما نلقي نظرةً على الملف src/main.rs مجددًا، سنلاحظ أن `cargo fix` قد غيّر الشيفرة.

اسم الملف: src/main.rs

```
fn do_something() {}

fn main() {
    for _i in 0..100 {
        do_something();
    }
}
```

أصبح اسم المتغير `_i` في الحلقة `for` ولا يظهر أي تحذير.

يمكننا أيضًا استخدام أمر `cargo fix` لتحويل الشيفرة بين إصدارات رست المختلفة، وسنتحدث عن الإصدارات في الملحق "هـ" التالي.

20.6.3 تنقيح أكثر للصياغة مع Clippy

أداة Clippy هي مجموعة من أدوات التنقيح لتحليل الشيفرة البرمجية لملاحظة الأخطاء الشائعة وتحسين شيفرة رست الخاصة بك.

اكتب الأمر التالي لتثبيت Clippy:

```
$ rustup component add clippy
```

اكتب التالي لتنفيذ تنقيح Clippy على أي مشروع Cargo:

```
$ cargo clippy
```

لنقل أنك تريد كتابة برنامج يستخدم تقريبًا لثابت رياضي مثل باي π كما يفعل هذا البرنامج.

اسم الملف: src/main.rs

```
fn main() {
    let x = 3.1415;
    let r = 8.0;
    println!("the area of the circle is {}", x * r * r);
}
```

يعطي تنفيذ `cargo clippy` على هذا المشروع هذا الخطأ.

```
error: approximate value of `f{32, 64}::consts::PI` found
 --> src/main.rs:2:13
  |
|     let x = 3.1415;
  |             ^^^^^^
  |
  = note: `#[deny(clippy::approx_constant)]` on by default
  = help: consider using the constant directly
  = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#approx_const
ant
```

يجعلنا هذا الخطأ نعرف أن رست لديها ثابت `PI` أكثر دقة وأن البرنامج سيكون أكثر صحة إذا استخدمنا الثابت. يمكنك بعد ذلك بتغيير الشيفرة باستخدام الثابت `PI`، ولن تعطي الشيفرة التالية أي أخطاء أو تحذيرات من Clippy.

اسم الملف: src/main.rs

```
fn main() {
    let x = std::f64::consts::PI;
    let r = 8.0;
    println!("the area of the circle is {}", x * r * r);
}
```

للمزيد عن Clippy راجع التوثيق الخاص به.

20.6.4 الدمج مع بيئة التطوير المتكاملة IDEs باستخدام rust-analyzer

ينصح مجتمع رست باستخدام rust-analyzer للمساعدة بالدمج مع IDE، إذ تحتوي هذه الأداة على مجموعة من الأدوات المتعلقة بالمصرّف وتستخدم بروتوكول لغة الخادم Language Server Protocol وهي خاصية لبيئة التطوير المتكاملة واللغات البرمجية لكي تتواصل مع بعضها. يمكن أن يستخدم العديد من العملاء `rust-analyzer` مثل إضافة محلّل رست الموجودة على محرّر الشيفرة البرمجية Visual Studio Code.

اذهب إلى الصفحة الرئيسية لمشروع `rust-analyzer` لخطوات التثبيت ثم ثبّت داعم لغة الخادم في بيئة التطوير المتكاملة الخاصة بك وبذلك تكسب البيئة البرمجية المتكاملة الخاصة بك بعض القدرات مثل الملء التلقائي والانتقال إلى التعريف والأخطاء الضمنية.

20.7 الملحق هـ: الإصدارات

رأينا في الفصل الأول أن `cargo new` تضيف بعض البيانات الوصفية metadata إلى ملف cargo.toml الخاص بك، وسنتحدث في هذا الملحق عن معناها.

لدى لغة رست والمصرّف دورة إصدار كل ستة أسابيع، ويعني هذا أن المستخدم يحصل دائمًا على ميزات جديدة، مقارنةً باللغات البرمجية الأُخرى التي تصدر ميزات جديدة بتواتر أقل؛ إذ تصدر لغة رست تحديثات أصغر بتواتر أكبر، وتتراكم هذه التغيرات بعد فترة ولكن من إصدار لإصدار، فلا يمكننا القول أن "رست بإصدار 1.31 تغيرت كثيرًا عن رست بإصدار 1.10".

يصدر فريق رست كل سنتين أو ثلاث إصدار رست جديد، بحيث يجمع كل إصدار الميزات التي وصلت في مجموعة واضحة مع تحديث التوثيقات والأدوات. تُرسل الإصدارات الجديدة مثل جزء من عملية الإصدار كل ستة أسابيع.

تفيد الإصدارات الناس بطرق مختلفة:

- بالنسبة لمستخدمي رست الفاعلين؛ يجمع الإصدار الجديد كل التغييرات الجزئية إلى مجموعة كاملة سهلة الفهم.
- بالنسبة لغير المستخدمين؛ يشير الإصدار الجديد أن تقدمًا كبيرًا قد حصل مما قد يجعلهم يعيدوا النظر باستخدام رست.
- بالنسبة لمطوّري رست؛ يقدم الإصدار الجديد نقطة مرجع للمشروع ككل.

يوجد في وقتنا الحالي ثلاث إصدارات من رست متوفرة: رست 2015 ورست 2018 ورست 2021، ويستخدم هذا الكتاب مصطلحات إصدار رست 2021.

يشير المفتاح `edition` في cargo.toml إلى أي إصدار يجب أن يستخدم المصرّف للشيفرة الخاصة بك. إذا لم يتواجد المفتاح فإن رست تستخدم `2015` بمثابة قيمة للنسخة لأسباب تتعلق بالتوافق الرجعي.

يمكن أن يختار كل مشروع أي إصدار غير الإصدار الافتراضي 2015، إذ يمكن أن تحتوي الإصدارات بعض التغيرات غير المتوافقة مثل كلمات مفتاحية جديدة قد تتعارض مع بعض المعرّفات في الشيفرة ولكن إذا رفضت هذه التغييرات ستبقى الشيفرة تصرّف حتى لو حدثت نسخة مصرّف رست المستخدمة.

تتوافق كل مصرّفات رست مع كل الإصدارات التي سبقتها ويمكنها ربط الوحدات المصرّفة لأي إصدار مع بعضها، وتؤثر تغييرات الإصدار فقط على كيفية تحليل المصرّف الأولي للشيفرة، فمثلًا إذا كنت تستخدم رست 2015 وواحد من الاعتماديات الخاصة بك يستخدم رست 2018، سيّصرّف مشروعك بنجاح ويمكنه استخدام الاعتمادية تلك والعكس صحيح.

للمزيد من التوضيح، ستتوفر معظم الميزات على جميع الإصدارات وسيستمر المطورون الذين يستخدمون رست بملاحظة التحسينات كلما تُنشر إصدارات جديدة، ولكن في بعض الحالات ستتوفر ميزات جديدة في إصدار معيّن فقط إذا أضيفت إليه كلمات مفتاحية جديدة معينة، وبالتالي سيتوجب عليك تغيير الإصدار للاستفادة من تلك الميزات.

لتفاصيل أكثر يوضح كتاب "دليل الإصدار" الفرق بين الإصدارات ويشرح كيفية تحديث شيفرتك تلقائيًا عبر `cargo fix`.

20.8 الملحق ز: كيف صنعت رست ورست الليلية Nightly Rust

يتحدث هذا الملحق عن كيفية صنع رست وكيف يؤثر ذلك عليك بصفتك مطوّرًا.

20.8.1 استقرار دون ركود

تركّز رست كثيرًا على استقرار الشيفرة البرمجية، إذ نريد أن يكون لرست أساس قوي لتبنى التطبيقات عليها، وإذا كانت الأمور تتغير باستمرار فسيكون ذلك مستحيلًا؛ وفي الوقت ذاته، إذا لم يكن من الممكن أن نجرب ميزات جديدة فقد لا نستطيع العثور على الأخطاء المهمة إلا بعد نشر المشروع وعندها لا يمكننا تغيير أي شيء بخصوص ذلك.

نحل هذه المشكلة عبر ما نسميه "استقرار بدون ركود Stability Without Stagnation"، الذي ينص على أنه لا يجب الخوف من الانتقال إلى نسخة جديدة مستقرة من رست، فكل تحديث يجب أن يكون سهلًا ولكنه يضيف ميزات جديدة وأخطاء أقل ووقت تصريف أسرع في الوقت ذاته.

20.8.2 قنوات النشر واللحاق بالقطار

يعمل تطوير رست على جدول يدعى بجدول القطار train schedule، أي أن كل التطويرات تحدث على فرع `master` من مستودع رست. تتبع الإصدارات نموذج نشر قطار train model release الذي كان يُستعمل من قبل برنامج نظام تشغيل سيسكو Cisco IOS ومشاريع برمجية أُخرى. هناك ثلاث قنوات نشر لرست:

- ليلي Nightly
- تجريبي Beta
- مستقر Stable

يستخدم مطورو رست القناة المستقرة بصورةٍ أساسية ولكن يمكن للذين يريدون تجريب خاصيّات جديدة استخدام رست الليلية أو التجريبية.

إليك مثالًا عن كيفية عمل نظام التطوير والنشر: لنفترض أن فريق رست يعمل على الإصدار 1.5، وعلى الرغم من أن هذه النسخة أصدرت في ديسمبر 2015 إلا أنها ستمنحنا أرقام لنسخ حقيقية في سبيل توضيح مثالنا هذا، فعندما تضاف ميّزة جديدة لرست يضاف إيداع commit جديد على الفرع `master`، وتُضاف نسخة ليلية من رست كل ليلة. كل يوم هو يوم نشر وتُنشأ كل هذه الإصدارات من البُنية التحتية تلقائيًا أي تكون إصداراتنا بعد مرور الوقت على النحو التالي، مرةً كل ليلة:

```
nightly: * - - * - - *
```

يحين الـوقت كـل سـتة أسـابيع لإصـدار جديـد، ويتفـرع `beta` من الفـرع `master` في مسـتودع رسـت المستخدم من الفرع الليلي، الآن هناك إصداران.

```
nightly: * - - * - - *
                   |
beta:              *
```

لا يستخدم معظم المبرمجين الإصدار التجريبي، ولكن يجربونه مع أنظمـة CI لمسـاعدة رسـت بـالعثور على تراجع regression محتمل وفي تلك الأثناء لا يزال هناك إصدار ليلي كل ليلة:

```
nightly: * - - * - - * - - * - - *
                   |
beta:              *
```

لنفترض أنه وجد بعض التراجع، هذا أمرٌ جيد لأنه لدينا بعض الوقت لاختبار إصـدار بيتـا قبـل أن ينتقـل إلى إصدار مستقر. يُطبق الإصلاح على `master` لـذا يُصـلح الإصـدار الليلي وبعـدها يُحمـل عكسـيًا إلى فـرع `beta` ويضاف إصدار تجريبي جديد.

```
nightly: * - - * - - * - - * - - * - - *
                   |
beta:              * - - - - - - - - *
```

بعد ستة أسابيع من إصدار النسخة التجريبية يحين وقت نشر إصدار مستقر. ينتج الفرع `stable` من فـرع `beta`.

```
nightly: * - - * - - * - - * - - * - - * - * - *
                   |
beta:              * - - - - - - - - *
                                     |
stable:                              *
```

عظيم، فقد أصبحت رست 1.5 أصبح جاهزة، إلا أننا نسـينا شـيئًا واحـدًا لأن الأسـابيع السـتة الـتي مضـت تحتاج إلى نسخة تجريبية جديدة لنسـخة رسـت الجديـدة ألا وهي رسـت 1.6، لـذا بعـد أن يتفـرع `stable` من `beta` النسخة الجديدة من `beta` تتفرع من `nightly` مجددًا.

```
nightly: * - - * - - * - - * - - * - - * - * - *
                   |                         |
beta:              * - - - - - - - - *       *
```

```
                                    |
stable:                             *
```

يُسمى هذا "نموذج القطار" لأن إصدار جديد "يغادر المحطة" كل ستة أسابيع ولكنه بحاجة إلى أن يمر بقنـاة النسخة التجريبية قبل الوصول إلى الإصدار المستقر.

يُصدر رست كل ستة أسابيع تمامًا، فإذا علمت تاريخ أحد الإصدارات، يمكنك معرفة تـاريخ الإصـدار القـادم الذي هو بعد ستة أسابيع. الميزة الجيدة أن يكون هناك جدول زمني للإصدار كل ستة أشهر هو أن الإصدار التـالي قادم قريبًا، وإذا لم توجد ميزة في أحد الإصدارات لا داعٍ للقلق لأن إصدار جديد قادم في وقت قصير. هـذا يقلـل من حدوث ضغط دخول بعض الميّزات غير الكاملة في أوقات الإصدار القريبة.

يمكنك بفضل هذه العملية معاينة النسخة التالية من رست والتأكد أنه من السهل التحديث إليها. إذا كانت النسـخة التجريبيـة لا تعمـل فيمكن تقـديم تقريـر للفريـق لإصـلاحها قبـل موعـد الإصـدار المسـتقر. التـأخر في الإصدارات التجريبية أمر نادر الحدوث ولكن لا يزال `rustc` برنامجًا وهو عرضة للأخطاء.

20.8.3 الميزات غير المستقرة

هناك نقطة أخرى في هذا النموذج من الإصدار وهي الميزات غير المستقرة، إذ تستخدم رست تقنية اسمها "راية الميزة feature flag" لتحديد أي ميزات متوفرة في إصدار ما؛ فإذا كانت ميزة ما قيـد التطـوير تصـل إلى `master` -وبالتالي إلى رست الليلية- ولكن تحت راية ميزة، فهذا يعني أن المستخدم يستطيع تجربة هذه المـيزة قيد التطوير إن أراد ذلك، ولكن يجب أن يستخدم إصدار رست الليلي وأن يشـير إلى الرايـة المناسـبة لاسـتخدام الميزة في الشيفرة المصدرية.

لا يمكن استخدام رايات الميزة إذا كنت تستخدم إصدار رست التجريبي أو المستقر، فهذا هـو الشـيء الـذي يسمح لنا باستخدام الميزات الجديدة عمليًا قبل التصريح عنها بأنها مستقرة للأبد. يمكن للذين يريدون اسـتخدام الميزات الحديثة فعل ذلك بينما يمكن للذين يريدون تجربة مستقرة البقـاء على الإصـدار المسـتقر ويطمئنـوا أن شيفرتهم لن تتضرر بذلك. هذا هو معنى الاستقرار بدون ركود.

يحتوي هذا الكتاب فقط على الميزات المستقرة، لأن الميزات قيد التطوير تتغير دائمًا وحتمًا ستتغير ما بين وقت كتابة هذا الكتاب ووقت تفعيلها في الإصدارات المستقرة. يمكنك إيجاد توثيق عن مـيزات رسـت الليليـة على الإنترنت.

20.8.4 Rustup ودور رست الليلية

تقدم لك Rustup سـهولة التغيـير بين قنـوات إصـدارات رسـت المختلفـة بصـورةٍ شـاملة global أو على أساس كل مشروع. غالبًا ما تكون نسخة رست المستقرة مثبتة ولتنصيب رست الليلية نكتب الأمر:

```
$ rustup toolchain install nightly
```

بإمكانك رؤية كل سلاسل الأدوات toolchains (إصدارات رست والمحتوى المرتبط بها) التي ثبتتها مع `rustup` كذلك، إليك مثال لواحد من حواسيب ويندوز الخاصة بمؤلف الكتاب.

```
> rustup toolchain list
stable-x86_64-pc-windows-msvc (default)
beta-x86_64-pc-windows-msvc
nightly-x86_64-pc-windows-msvc
```

كما ترى، فإن سلسلة الأدوات المستقرة هي الافتراضية، ويعمل أغلب المستخدمين على نسخة رست المستقرة أغلب الأوقات وربما تريد أنت فعل ذلك أيضًا، ولكن تريد استخدام رست الليلي لمشروع معين لأنك مهتم بميزة حديثة، ولفعل ذلك يمكنك استخدام `rustup override` في مجلد ذلك المشروع لتضبط سلسلة الأدوات الليلية التي يجب أن يستخدمها `rustup` عندما تكون في ذلك المجلد:

```
$ cd ~/projects/needs-nightly
$ rustup override set nightly
```

سيضمن `rustup` الآن في كل مرة تستدعي `rustc` أو `cargo` داخل "~/projects/needs-nightly" أنك تستخدم رست الليلي بدلًا من نسخة رست المستقرة الافتراضية، وسيكون ذلك مفيدًا عندما يكون لديك العديد من مشاريع رست.

20.8.5 عملية RFC وفرق العمل المتعلقة بها

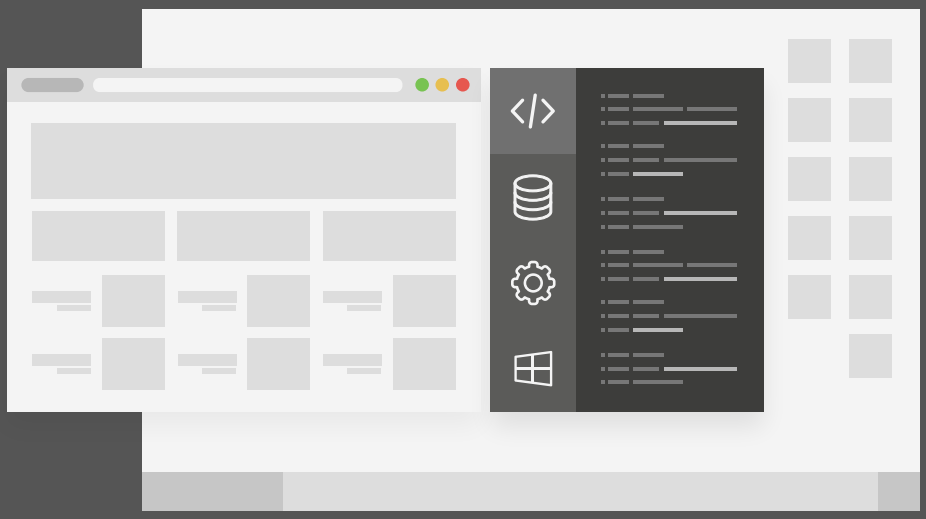
كيف تسمع عن هذه الميزات الجديدة؟ يتبع نموذج تطوير رست طريقة طلب تعليقات Request For Comments -أو اختصارًا RFC، إذ يمكنك كتابة طلب يُسمى RFC إذا أردت تحسينًا في رست.

يستطيع كل شخص كتابة RFC لتحسين رست، بحيث يراجع فريق رست هذه الطلبات ويناقشها وهذه العملية مكونة من العديد من الفرق الفرعية بحسب الموضوع. توجد قائمة كاملة للفرق على موقع رست تضم فرقًا لكل ناحية من نواحي المشروع كتصميم اللغة البرمجية وتنفيذ المصرّف والبنية التحتية والتوثيق والمزيد، إذ يقرأ الفريق المناسب التعليقات ويردّ عليها وفي النهاية يحدث قرار برفض أو قبول الميزة بالإجماع.

تُفتح قضية issue على مستودع رست إذا قُبلت ميزة ما بحيث يمكن لأي شخص أن ينفّذها. يمكن ألا يكون الشخص الذي يطبق الميزة هو الشخص الذي اقترح الميزة. يصل التنفيذ على الفرع `master` خلف بوابة ميزة عندما تجهز الميزة كما تحدثنا في قسم "ميزات غير مستقرة".

يناقش أعضاء الفريق بعد فترة هذه الميزة وكيفية عملها على رست الليلية عندما يجرب مطوري رست الذين يعملون على إصدار رست الليلي الميزة الجديدة، ويقررون إذا ما كان يجب وضعها في رست المستقر أو لا. إذا كان القرار للمضي قُدُمًا تُزال بوابة الميزة وتُعدّ الميزة مستقرة، وتركب القطار إلى إصدار جديد من رست.

دورة علوم الحاسوب



دورة تدريبية متكاملة تضعك على بوابة الاحتراف
في تعلم أساسيات البرمجة وعلوم الحاسوب

التحق بالدورة الآن

أحدث إصدارات أكاديمية حسوب