100 Exercises To Learn Rust

A hands-on course by Mainmatter

Luca Palmieri

Contents

Chapter 1

Welcome

Welcome to **"100 Exercises To Learn Rust"**!

This course will teach you Rust's core concepts, one exercise at a time.
You'll learn about Rust's syntax, its type system, its standard library, and its ecosystem.

We don't assume any prior knowledge of Rust, but we assume you know at least another programming language. We also don't assume any prior knowledge of systems programming or memory management. Those topics will be covered in the course.

In other words, we'll be starting from scratch!
You'll build up your Rust knowledge in small, manageable steps. By the end of the course, you will have solved ~100 exercises, enough to feel comfortable working on small to medium-sized Rust projects.

Methodology

This course is based on the "learn by doing" principle.
It has been designed to be interactive and hands-on.

Mainmatter developed this course to be delivered in a classroom setting, over 4 days: each attendee advances through the lessons at their own pace, with an experienced instructor providing guidance, answering questions and diving deeper into the topics as needed.
If you're interested in attending one of our training sessions, or if you'd like to bring this course to your company, please get in touch.

You can also follow the course on your own, but we recommend you find a friend or a mentor to help you along the way should you get stuck. You can also find solutions to all exercises in the `solutions` branch of the GitHub repository.

Formats

You can go through the course material in the browser.
You can also download it as a PDF file, for offline reading.

Structure

On the left side of the screen, you can see that the course is divided into sections. Each section introduces a new concept or feature of the Rust language.
To verify your understanding, each section is paired with an exercise that you need to solve.

You can find the exercises in the companion GitHub repository.
Before starting the course, make sure to clone the repository to your local machine:

```
# If you have an SSH key set up with GitHub
git clone git@github.com:mainmatter/100-exercises-to-learn-rust.git
# Otherwise, use the HTTPS URL:
#   https://github.com/mainmatter/100-exercises-to-learn-rust.git
```

We also recommend you work on a branch, so you can easily track your progress and pull in updates from the main repository, if needed:

```
cd 100-exercises-to-learn-rust
git checkout -b my-solutions
```

All exercises are located in the `exercises` folder. Each exercise is structured as a Rust package. The package contains the exercise itself, instructions on what to do (in `src/lib.rs`), and a test suite to automatically verify your solution.

`wr`, the workshop runner

To verify your solutions, we've provided a tool that will guide you through the course. It is the `wr` CLI (short for "workshop runner"). Install it with:

```
cargo install --locked workshop-runner
```

In a new terminal, navigate back to the top-level folder of the repository. Run the `wr` command to start the course:

```
wr
```

`wr` will verify the solution to the current exercise.
Don't move on to the next section until you've solved the exercise for the current one.

> We recommend committing your solutions to Git as you progress through the course, so you can easily track your progress and "restart" from a known point if needed.

Enjoy the course!

Author

This course was written by Luca Palmieri, Principal Engineering Consultant at Mainmatter.
Luca has been working with Rust since 2018, initially at TrueLayer and then at AWS.
Luca is the author of "Zero to Production in Rust", the go-to resource for learning how to build backend applications in Rust.
He is also the author and maintainer of a variety of open-source Rust projects, including `cargo-chef`, Pavex and `wiremock`.

Exercise

The exercise for this section is located in `01_intro/00_welcome`

1.1 Syntax

Don't jump ahead!
Complete the exercise for the previous section before you start this one.
It's located in `exercises/01_intro/00_welcome`, in the course GitHub's repository.
Use `wr` to start the course and verify your solutions.

The previous task doesn't even qualify as an exercise, but it already exposed you to quite a bit of Rust **syntax**. We won't cover every single detail of Rust's syntax used in the previous exercise. Instead, we'll cover *just enough* to keep going without getting stuck in the details.
One step at a time!

Comments

You can use `//` for single-line comments:

```
// This is a single-line comment
// Followed by another single-line comment
```

Functions

Functions in Rust are defined using the `fn` keyword, followed by the function's name, its input parameters, and its return type. The function's body is enclosed in curly braces `{}`.

In previous exercise, you saw the `greeting` function:

```
// `fn` <function_name> ( <input params> ) -> <return_type> { <body> }
fn greeting() -> &'static str {
    // TODO: fix me 👇
    "I'm ready to __!"
}
```

`greeting` has no input parameters and returns a reference to a string slice (`&'static str`).

Return type

The return type can be omitted from the signature if the function doesn't return anything (i.e. if it returns `()`, Rust's unit type). That's what happened with the `test_welcome` function:

```rust
fn test_welcome() {
    assert_eq!(greeting(), "I'm ready to learn Rust!");
}
```

The above is equivalent to:

```rust
// Spelling out the unit return type explicitly
//                   👇
fn test_welcome() -> () {
    assert_eq!(greeting(), "I'm ready to learn Rust!");
}
```

Returning values

The last expression in a function is implicitly returned:

```rust
fn greeting() -> &'static str {
    // This is the last expression in the function
    // Therefore its value is returned by `greeting`
    "I'm ready to learn Rust!"
}
```

You can also use the `return` keyword to return a value early:

```rust
fn greeting() -> &'static str {
    // Notice the semicolon at the end of the line!
    return "I'm ready to learn Rust!";
}
```

It is considered idiomatic to omit the `return` keyword when possible.

Input parameters

Input parameters are declared inside the parentheses `()` that follow the function's name.
Each parameter is declared with its name, followed by a colon `:`, followed by its type.

For example, the `greet` function below takes a `name` parameter of type `&str` (a "string slice"):

```rust
// An input parameter
//        👇
fn greet(name: &str) -> String {
    format!("Hello, {}!", name)
}
```

If there are multiple input parameters, they must be separated with commas.

Type annotations

Since we've been mentioned "types" a few times, let's state it clearly: Rust is a **statically typed language**.
Every single value in Rust has a type and that type must be known to the compiler at compile-time.

Types are a form of **static analysis**.
You can think of a type as a **tag** that the compiler attaches to every value in your program. Depending on the tag, the compiler can enforce different rules—e.g. you can't add a string to a number, but you can add two numbers together. If leveraged correctly, types can prevent whole classes of runtime bugs.

Exercise

The exercise for this section is located in `01_intro/01_syntax`

Chapter 2

A Basic Calculator

In this chapter we'll learn how to use Rust as a **calculator**.
It might not sound like much, but it'll give us a chance to cover a lot of Rust's basics, such as:

- How to define and call functions
- How to declare and use variables
- Primitive types (integers and booleans)
- Arithmetic operators (including overflow and underflow behavior)
- Comparison operators
- Control flow
- Panics

Nailing the basics with a few exercises will get the language flowing under your fingers. When we move on to more complex topics, such as traits and ownership, you'll be able to focus on the new concepts without getting bogged down by the syntax or other trivial details.

Exercise

The exercise for this section is located in `02_basic_calculator/00_intro`

2.1 Types, part 1

In the “Syntax” section `compute`’s input parameters were of type `u32`.
Let’s unpack what that *means*.

Primitive types

`u32` is one of Rust’s **primitive types**. Primitive types are the most basic building blocks of a language. They’re built into the language itself—i.e. they are not defined in terms of other types.

You can combine these primitive types to create more complex types. We’ll see how soon enough.

Integers

`u32`, in particular, is an **unsigned 32-bit integer**.

An integer is a number that can be written without a fractional component. E.g. `1` is an integer, while `1.2` is not.

Signed vs. unsigned

An integer can be **signed** or **unsigned**.
An unsigned integer can only represent non-negative numbers (i.e. `0` or greater). A signed integer can represent both positive and negative numbers (e.g. `-1`, `12`, etc.).

The `u` in `u32` stands for **unsigned**.
The equivalent type for signed integer is `i32`, where the `i` stands for integer (i.e. any integer, positive or negative).

Bit width

The `32` in `u32` refers to the **number of bits**[1] used to represent the number in memory. The more bits, the larger the range of numbers that can be represented.

Rust supports multiple bit widths for integers: `8`, `16`, `32`, `64`, `128`.

With 32 bits, `u32` can represent numbers from `0` to `2^32 - 1` (a.k.a. `u32::MAX`).
With the same number of bits, a signed integer (`i32`) can represent numbers from `-2^31` to `2^31 - 1` (i.e. from `i32::MIN` to `i32::MAX`).
The maximum value for `i32` is smaller than the maximum value for `u32` because one

[1] A bit is the smallest unit of data in a computer. It can only have two values: `0` or `1`.

bit is used to represent the sign of the number. Check out the two's complement representation for more details on how signed integers are represented in memory.

Summary

Combining the two variables (signed/unsigned and bit width), we get the following integer types:

Bit width	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`

Literals

A **literal** is a notation for representing a fixed value in source code.
For example, `42` is a Rust literal for the number forty-two.

Type annotations for literals

But all values in Rust have a type, so... what's the type of `42`?

The Rust compiler will try to infer the type of a literal based on how it's used.
If you don't provide any context, the compiler will default to `i32` for integer literals.
If you want to use a different type, you can add the desired integer type as a suffix—e.g. `2u64` is a 2 that's explicitly typed as a `u64`.

Underscores in literals

You can use underscores `_` to improve the readability of large numbers.
For example, `1_000_000` is the same as `1000000`.

Arithmetic operators

Rust supports the following arithmetic operators[2] for integers:

[2] Rust doesn't let you define custom operators, but it puts you in control of how the built-in operators behave. We'll talk about operator overloading later in the course, after we've covered traits.

- `+` for addition
- `-` for subtraction
- `*` for multiplication
- `/` for division
- `%` for remainder

Precedence and associativity rules for these operators are the same as in mathematics. You can use parentheses to override the default precedence. E.g. `2 * (3 + 4)`.

> ⚠️ **Warning**
>
> The division operator `/` performs integer division when used with integer types. I.e. the result is truncated towards zero. For example, `5 / 2` is `2`, not `2.5`.

No automatic type coercion

As we discussed in the previous exercise, Rust is a statically typed language.
In particular, Rust is quite strict about type coercion. It won't automatically convert a value from one type to another[3], even if the conversion is lossless. You have to do it explicitly.

For example, you can't assign a `u8` value to a variable with type `u32`, even though all `u8` values are valid `u32` values:

```rust
let b: u8 = 100;
let a: u32 = b;
```

It'll throw a compilation error:

```
error[E0308]: mismatched types
  |
3 |     let a: u32 = b;
  |            ---   ^ expected `u32`, found `u8`
  |            |
  |            expected due to this
  |
```

We'll see how to convert between types later in this course.

[3]There are some exceptions to this rule, mostly related to references, smart pointers and ergonomics. We'll cover those later on. A mental model of "all conversions are explicit" will serve you well in the meantime.

Further reading

- The integer types section in the official Rust book

Exercise

The exercise for this section is located in `02_basic_calculator/01_integers`

2.2 Variables

In Rust, you can use the `let` keyword to declare **variables**.
For example:

```
let x = 42;
```

Above we defined a variable `x` and assigned it the value `42`.

Type

Every variable in Rust must have a type. It can either be inferred by the compiler or explicitly specified by the developer.

Explicit type annotation

You can specify the variable type by adding a colon `:` followed by the type after the variable name. For example:

```
// let <variable_name>: <type> = <expression>;
let x: u32 = 42;
```

In the example above, we explicitly constrained the type of `x` to be `u32`.

Type inference

If we don't specify the type of a variable, the compiler will try to infer it based on the context in which the variable is used.

```
let x = 42;
let y: u32 = x;
```

In the example above, we didn't specify the type of `x`.
`x` is later assigned to `y`, which is explicitly typed as `u32`. Since Rust doesn't perform automatic type coercion, the compiler infers the type of `x` to be `u32`—the same as `y` and the only type that will allow the program to compile without errors.

Inference limitations

The compiler sometimes needs a little help to infer the correct variable type based on its usage.
In those cases you'll get a compilation error and the compiler will ask you to provide an explicit type hint to disambiguate the situation.

Function arguments are variables

Not all heroes wear capes, not all variables are declared with `let`.
Function arguments are variables too!

```rust
fn add_one(x: u32) -> u32 {
    x + 1
}
```

In the example above, `x` is a variable of type `u32`.
The only difference between `x` and a variable declared with `let` is that functions arguments **must** have their type explicitly declared. The compiler won't infer it for you. This constraint allows the Rust compiler (and us humans!) to understand the function's signature without having to look at its implementation. That's a big boost for compilation speed[4]!

Initialization

You don't have to initialize a variable when you declare it.
For example

```rust
let x: u32;
```

is a valid variable declaration.
However, you must initialize the variable before using it. The compiler will throw an error if you don't:

```rust
let x: u32;
let y = x + 1;
```

will throw a compilation error:

```
error[E0381]: used binding `x` isn't initialized
 --> src/main.rs:3:9
  |
2 | let x: u32;
  |     - binding declared here but left uninitialized
3 | let y = x + 1;
  |         ^ `x` used here but it isn't initialized
  |
help: consider assigning a value
  |
2 | let x: u32 = 0;
  |            +++
```

[4]The Rust compiler needs all the help it can get when it comes to compilation speed.

Exercise

The exercise for this section is located in `02_basic_calculator/02_variables`

2.3 Control flow, part 1

All our programs so far have been pretty straightforward.
A sequence of instructions is executed from top to bottom, and that's it.

It's time to introduce some **branching**.

`if` clauses

The `if` keyword is used to execute a block of code only if a condition is true.

Here's a simple example:

```rust
let number = 3;
if number < 5 {
    println!("`number` is smaller than 5");
}
```

This program will print `number is smaller than 5` because the condition `number < 5` is true.

`else` clauses

Like most programming languages, Rust supports an optional `else` branch to execute a block of code when the condition in an `if` expression is false.
For example:

```rust
let number = 3;

if number < 5 {
    println!("`number` is smaller than 5");
} else {
    println!("`number` is greater than or equal to 5");
}
```

Booleans

The condition in an `if` expression must be of type `bool`, a **boolean**.
Booleans, just like integers, are a primitive type in Rust.

A boolean can have one of two values: `true` or `false`.

No truthy or falsy values

If the condition in an `if` expression is not a boolean, you'll get a compilation error.

For example, the following code will not compile:

```rust
let number = 3;
if number {
    println!("`number` is not zero");
}
```

You'll get the following compilation error:

```
error[E0308]: mismatched types
 --> src/main.rs:3:8
  |
3 |     if number {
  |        ^^^^^^ expected `bool`, found integer
```

This follows from Rust's philosophy around type coercion: there's no automatic conversion from non-boolean types to booleans. Rust doesn't have the concept of **truthy** or **falsy** values, like JavaScript or Python.
You have to be explicit about the condition you want to check.

Comparison operators

It's quite common to use comparison operators to build conditions for `if` expressions.
Here are the comparison operators available in Rust when working with integers:

- `==`: equal to
- `!=`: not equal to
- `<`: less than
- `>`: greater than
- `<=`: less than or equal to
- `>=`: greater than or equal to

`if/else` is an expression

In Rust, `if` expressions are **expressions**, not statements: they return a value.
That value can be assigned to a variable or used in other expressions. For example:

```rust
let number = 3;
let message = if number < 5 {
    "smaller than 5"
} else {
    "greater than or equal to 5"
};
```

In the example above, each branch of the `if` evaluates to a string literal, which is then assigned to the `message` variable.
The only requirement is that both `if` branches return the same type.

Exercise

The exercise for this section is located in `02_basic_calculator/03_if_else`

2.4 Panics

Let's go back to the speed function you wrote for the "Variables" section. It probably looked something like this:

```
fn speed(start: u32, end: u32, time_elapsed: u32) -> u32 {
    let distance = end - start;
    distance / time_elapsed
}
```

If you have a keen eye, you might have spotted one issue[5]: what happens if `time_elapsed` is zero?

You can try it out on the Rust playground!
The program will exit with the following error message:

```
thread 'main' panicked at src/main.rs:3:5:
attempt to divide by zero
```

This is known as a **panic**.
A panic is Rust's way to signal that something went so wrong that the program can't continue executing, it's an **unrecoverable error**[6]. Division by zero classifies as such an error.

The panic! macro

You can intentionally trigger a panic by calling the `panic!` macro[7]:

```
fn main() {
    panic!("This is a panic!");
    // The line below will never be executed
    let x = 1 + 2;
}
```

There are other mechanisms to work with recoverable errors in Rust, which we'll cover later. For the time being we'll stick with panics as a brutal but simple stopgap solution.

---

[5] There's another issue with speed that we'll address soon enough. Can you spot it?

[6] You can try to catch a panic, but it should be a last resort attempt reserved for very specific circumstances.

[7] If it's followed by a `!`, it's a macro invocation. Think of macros as spicy functions for now. We'll cover them in more detail later in the course.

Further reading

- The panic! macro documentation

Exercise

The exercise for this section is located in `02_basic_calculator/04_panics`

2.5 Factorial

So far you've learned:

- How to define a function
- How to call a function
- Which integer types are available in Rust
- Which arithmetic operators are available for integers
- How to execute conditional logic via comparisons and `if`/`else` expressions

It looks like you're ready to tackle factorials!

Exercise

The exercise for this section is located in `02_basic_calculator/05_factorial`

2.6 Loops, part 1: while

Your implementation of `factorial` has been forced to use recursion.
This may feel natural to you, especially if you're coming from a functional programming background. Or it may feel strange, if you're used to more imperative languages like C or Python.

Let's see how you can implement the same functionality using a **loop** instead.

The while loop

A `while` loop is a way to execute a block of code as long as a **condition** is true.
Here's the general syntax:

```
while <condition> {
    // code to execute
}
```

For example, we might want to sum the numbers from 1 to 5:

```
let sum = 0;
let i = 1;
// "while i is less than or equal to 5"
while i <= 5 {
    // `+=` is a shorthand for `sum = sum + i`
    sum += i;
    i += 1;
}
```

This will keep adding 1 to `i` and `i` to `sum` until `i` is no longer less than or equal to 5.

The mut keyword

The example above won't compile as is. You'll get an error like:

```
error[E0384]: cannot assign twice to immutable variable `sum`
 --> src/main.rs:7:9
  |
2 |     let sum = 0;
  |         ---
  |         |
  |         first assignment to `sum`
  |         help: consider making this binding mutable: `mut sum`
...
```

```
7 |         sum += i;
  |         ^^^^^^^^ cannot assign twice to immutable variable

error[E0384]: cannot assign twice to immutable variable `i`
 --> src/main.rs:8:9
  |
3 |     let i = 1;
  |         -
  |         |
  |         first assignment to `i`
  |         help: consider making this binding mutable: `mut i`
...
8 |         i += 1;
  |         ^^^^^^ cannot assign twice to immutable variable
```

This is because variables in Rust are **immutable** by default.
You can't change their value once it has been assigned.

If you want to allow modifications, you have to declare the variable as **mutable** using the `mut` keyword:

```rust
// `sum` and `i` are mutable now!
let mut sum = 0;
let mut i = 1;

while i <= 5 {
    sum += i;
    i += 1;
}
```

This will compile and run without errors.

Further reading

- `while` loop documentation

Exercise

The exercise for this section is located in `02_basic_calculator/06_while`

2.7 Loops, part 2: `for`

Having to manually increment a counter variable is somewhat tedious. The pattern is also extremely common!
To make this easier, Rust provides a more concise way to iterate over a range of values: the `for` loop.

The `for` loop

A `for` loop is a way to execute a block of code for each element in an iterator[8].

Here's the general syntax:

```
for <element> in <iterator> {
    // code to execute
}
```

Ranges

Rust's standard library provides **range** type that can be used to iterate over a sequence of numbers[9].

For example, if we want to sum the numbers from 1 to 5:

```
let mut sum = 0;
for i in 1..=5 {
    sum += i;
}
```

Every time the loop runs, `i` will be assigned the next value in the range before executing the block of code.

There are five kinds of ranges in Rust:

- `1..5`: A (half-open) range. It includes all numbers from 1 to 4. It doesn't include the last value, 5.
- `1..=5`: An inclusive range. It includes all numbers from 1 to 5. It includes the last value, 5.
- `1..`: An open-ended range. It includes all numbers from 1 to infinity (well, until the maximum value of the integer type).

[8] Later in the course we'll give a precise definition of what counts as an "iterator". For now, think of it as a sequence of values that you can loop over.

[9] You can use ranges with other types too (e.g. characters and IP addresses), but integers are definitely the most common case in day-to-day Rust programming.

- `..5`: A range that starts at the minimum value for the integer type and ends at 4. It doesn't include the last value, 5.
- `..=5`: A range that starts at the minimum value for the integer type and ends at 5. It includes the last value, 5.

You can use a `for` loop with the first three kinds of ranges, where the starting point is explicitly specified. The last two range types are used in other contexts, that we'll cover later.

The extreme values of a range don't have to be integer literals—they can be variables or expressions too!

For example:

```
let end = 5;
let mut sum = 0;

for i in 1..(end + 1) {
    sum += i;
}
```

Further reading

- `for` loop documentation

Exercise

The exercise for this section is located in `02_basic_calculator/07_for`

2.8 Overflow

The factorial of a number grows quite fast.
For example, the factorial of 20 is 2,432,902,008,176,640,000. That's already bigger than the maximum value for a 32-bit integer, 2,147,483,647.

When the result of an arithmetic operation is bigger than the maximum value for a given integer type, we are talking about **an integer overflow**.

Integer overflows are an issue because they violate the contract for arithmetic operations.
The result of an arithmetic operation between two integers of a given type should be another integer of the same type. But the *mathematically correct result* doesn't fit into that integer type!

> If the result is smaller than the minimum value for a given integer type, we refer to the event as **an integer underflow**.
> For brevity, we'll only talk about integer overflows for the rest of this section, but keep in mind that everything we say applies to integer underflows as well.
>
> The `speed` function you wrote in the "Variables" section underflowed for some input combinations. E.g. if end is smaller than `start`, `end - start` will underflow the `u32` type since the result is supposed to be negative but `u32` can't represent negative numbers.

No automatic promotion

One possible approach would be automatically promote the result to a bigger integer type. E.g. if you're summing two `u8` integers and the result is 256 (`u8::MAX + 1`), Rust could choose to interpret the result as `u16`, the next integer type that's big enough to hold 256.

But, as we've discussed before, Rust is quite picky about type conversions. Automatic integer promotion is not Rust's solution to the integer overflow problem.

Alternatives

Since we ruled out automatic promotion, what can we do when an integer overflow occurs?
It boils down to two different approaches:

- Reject the operation
- Come up with a "sensible" result that fits into the expected integer type

Reject the operation

This is the most conservative approach: we stop the program when an integer overflow occurs.
That's done via a panic, the mechanism we've already seen in the "Panics" section.

Come up with a "sensible" result

When the result of an arithmetic operation is bigger than the maximum value for a given integer type, you can choose to **wrap around**.
If you think of all the possible values for a given integer type as a circle, wrapping around means that when you reach the maximum value, you start again from the minimum value.

For example, if you do a **wrapping addition** between 1 and 255 (=`u8::MAX`), the result is 0 (=`u8::MIN`). If you're working with signed integers, the same principle applies. E.g. adding 1 to 127 (=`i8::MAX`) with wrapping will give you -128 (=`i8::MIN`).

`overflow-checks`

Rust lets you, the developer, choose which approach to use when an integer overflow occurs. The behaviour is controlled by the `overflow-checks` profile setting.

If `overflow-checks` is set to `true`, Rust will **panic at runtime** when an integer operation overflows. If `overflow-checks` is set to `false`, Rust will **wrap around** when an integer operation overflows.

You may be wondering—what is a profile setting? Let's get into that!

Profiles

A **profile** is a set of configuration options that can be used to customize the way Rust code is compiled.

Cargo provides two built-in profiles: `dev` and `release`.
The `dev` profile is used every time you run `cargo build`, `cargo run` or `cargo test`. It's aimed at local development, therefore it sacrifices runtime performance in favor of faster compilation times and a better debugging experience.
The `release` profile, instead, is optimized for runtime performance but incurs longer compilation times. You need to explicitly request via the `--release` flag—e.g. `cargo build --release` or `cargo run --release`.

> "Have you built your project in release mode?" is almost a meme in the Rust community.

> It refers to developers who are not familiar with Rust and complain about its performance on social media (e.g. Reddit, Twitter, etc.) before realizing they haven't built their project in release mode.

You can also define custom profiles or customize the built-in ones.

`overflow-check`

By default, `overflow-checks` is set to:

- `true` for the `dev` profile
- `false` for the `release` profile

This is in line with the goals of the two profiles.
`dev` is aimed at local development, so it panics in order to highlight potential issues as early as possible.
`release`, instead, is tuned for runtime performance: checking for overflows would slow down the program, so it prefers to wrap around.

At the same time, having different behaviours for the two profiles can lead to subtle bugs.
Our recommendation is to enable `overflow-checks` for both profiles: it's better to crash than to silently produce incorrect results. The runtime performance hit is negligible in most cases; if you're working on a performance-critical application, you can run benchmarks to decide if it's something you can afford.

Further reading

- Check out "Myths and legends about integer overflow in Rust" for an in-depth discussion about integer overflow in Rust.

Exercise

The exercise for this section is located in `02_basic_calculator/08_overflow`

2.9 Case-by-case behavior

`overflow-checks` is a blunt tool: it's a global setting that affects the whole program. It often happens that you want to handle integer overflows differently depending on the context: sometimes wrapping is the right choice, other times panicking is preferable.

`wrapping_` methods

You can opt into wrapping arithmetic on a per-operation basis by using the `wrapping_` methods[10].
For example, you can use `wrapping_add` to add two integers with wrapping:

```
let x = 255u8;
let y = 1u8;
let sum = x.wrapping_add(y);
assert_eq!(sum, 0);
```

`saturating_` methods

Alternatively, you can opt into **saturating arithmetic** by using the `saturating_` methods.
Instead of wrapping around, saturating arithmetic will return the maximum or minimum value for the integer type. For example:

```
let x = 255u8;
let y = 1u8;
let sum = x.saturating_add(y);
assert_eq!(sum, 255);
```

Since `255 + 1` is `256`, which is bigger than `u8::MAX`, the result is `u8::MAX` (255). The opposite happens for underflows: `0 - 1` is `-1`, which is smaller than `u8::MIN`, so the result is `u8::MIN` (0).

You can't get saturating arithmetic via the `overflow-checks` profile setting—you have to explicitly opt into it when performing the arithmetic operation.

Exercise

The exercise for this section is located in `02_basic_calculator/09_saturating`

[10] You can think of methods as functions that are "attached" to a specific type. We'll cover methods (and how to define them) in the next chapter.

2.10 Conversions, pt. 1

We've repeated over and over again that Rust won't perform implicit type conversions for integers.
How do you perform *explicit* conversions then?

`as`

You can use the `as` operator to convert between integer types.
`as` conversions are **infallible**.

For example:

```rust
let a: u32 = 10;

// Cast `a` into the `u64` type
let b = a as u64;

// You can use `_` as the target type
// if it can be correctly inferred
// by the compiler. For example:
let c: u64 = a as _;
```

The semantics of this conversion are what you expect: all `u32` values are valid `u64` values.

Truncation

Things get more interesting if we go in the opposite direction:

```rust
// A number that's too big
// to fit into a `u8`
let a: u16 = 255 + 1;
let b = a as u8;
```

This program will run without issues, because `as` conversions are infallible. But what is the value of `b`? When going from a larger integer type to a smaller, the Rust compiler will perform a **truncation**.

To understand what happens, let's start by looking at how `256u16` is represented in memory, as a sequence of bits:

```
 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
|               |               |
+---------------+---------------+
  First 8 bits    Last 8 bits
```

When converting to a u8, the Rust compiler will keep the last 8 bits of a u16 memory representation:

```
 0 0 0 0 0 0 0 0
|               |
+---------------+
  Last 8 bits
```

Hence `256 as u8` is equal to `0`. That's… not ideal, in most scenarios.
In fact, the Rust compiler will actively try to stop you if it sees you trying to cast a literal value which will result in a truncation:

```
error: literal out of range for `i8`
  |
4 |     let a = 255 as i8;
  |             ^^^
  |
  = note: the literal `255` does not fit into the type `i8`
          whose range is `-128..=127`
  = help: consider using the type `u8` instead
  = note: `#[deny(overflowing_literals)]` on by default
```

Recommendation

As a rule of thumb, be quite careful with `as` casting.
Use it *exclusively* for going from a smaller type to a larger type. To convert from a larger to smaller integer type, rely on the *fallible* conversion machinery that we'll explore later in the course.

Limitations

Surprising behaviour is not the only downside of `as` casting. It is also fairly limited: you can only rely on `as` casting for primitive types and a few other special cases.
When working with composite types, you'll have to rely on different conversion mechanisms (fallible and infallible), which we'll explore later on.

Further reading

- Check out Rust's official reference to learn the precise behaviour of `as` casting for each source/target combination, as well as the exhaustive list of allowed conversions.

Exercise

The exercise for this section is located in `02_basic_calculator/10_as_casting`

Chapter 3

Modelling A Ticket

The first chapter should have given you a good grasp over some of Rust's primitive types, operators and basic control flow constructs.
In this chapter we'll go one step further and cover what makes Rust truly unique: **ownership**.
Ownership is what enables Rust to be both memory-safe and performant, with no garbage collector.

As our running example, we'll use a (JIRA-like) ticket, the kind you'd use to track bugs, features, or tasks in a software project.
We'll take a stab at modeling it in Rust. It'll be the first iteration—it won't be perfect nor very idiomatic by the end of the chapter. It'll be enough of a challenge though!
To move forward you'll have to pick up several new Rust concepts, such as:

- `struct`s, one of Rust's ways to define custom types
- Ownership, references and borrowing
- Memory management: stack, heap, pointers, data layout, destructors
- Modules and visibility
- Strings

Exercise

The exercise for this section is located in `03_ticket_v1/00_intro`

3.1 Structs

We need to keep track of three pieces of information for each ticket:

- A title
- A description
- A status

We can start by using a `String` to represent them. `String` is the type defined in Rust's standard library to represent UTF-8 encoded text.

But how do we **combine** these three pieces of information into a single entity?

Defining a `struct`

A `struct` defines a **new Rust type**.

```
struct Ticket {
    title: String,
    description: String,
    status: String
}
```

A struct is quite similar to what you would call a class or an object in other programming languages.

Defining fields

The new type is built by combining other types as **fields**.
Each field must have a name and a type, separated by a colon, `:`. If there are multiple fields, they are separated by a comma, `,`.

Fields don't have to be of the same type, as you can see in the `Configuration` struct below:

```
struct Configuration {
   version: u32,
   active: bool
}
```

Instantiation

You can create an instance of a struct by specifying the values for each field:

```rust
// Syntax: <StructName> { <field_name>: <value>, ... }
let ticket = Ticket {
    title: "Build a ticket system".into(),
    description: "A Kanban board".into(),
    status: "Open".into()
};
```

Accessing fields

You can access the fields of a struct using the `.` operator:

```rust
// Field access
let x = ticket.description;
```

Methods

We can attach behaviour to our structs by defining **methods**.
Using the `Ticket` struct as an example:

```rust
impl Ticket {
    fn is_open(self) -> bool {
        self.status == "Open"
    }
}

// Syntax:
// impl <StructName> {
//    fn <method_name>(<parameters>) -> <return_type> {
//        // Method body
//    }
// }
```

Methods are pretty similar to functions, with two key differences:

1. methods must be defined inside an **`impl` block**
2. methods may use `self` as their first parameter. `self` is a keyword and represents the instance of the struct the method is being called on.

`self`

If a method takes `self` as its first parameter, it can be called using the **method call syntax**:

```
// Method call syntax: <instance>.<method_name>(<parameters>)
let is_open = ticket.is_open();
```

This is the same calling syntax you used to perform saturating arithmetic operations on u32 values in the previous chapter.

Static methods

If a method doesn't take `self` as its first parameter, it's a **static method**.

```
struct Configuration {
    version: u32,
    active: bool
}

impl Configuration {
    // `default` is a static method on `Configuration`
    fn default() -> Configuration {
        Configuration { version: 0, active: false }
    }
}
```

The only way to call a static method is by using the **function call syntax**:

```
// Function call syntax: <StructName>::<method_name>(<parameters>)
let default_config = Configuration::default();
```

Equivalence

You can use the function call syntax even for methods that take `self` as their first parameter:

```
// Function call syntax:
//   <StructName>::<method_name>(<instance>, <parameters>)
let is_open = Ticket::is_open(ticket);
```

The function call syntax makes it quite clear that `ticket` is being used as `self`, the first parameter of the method, but it's definitely more verbose. Prefer the method call syntax when possible.

Exercise

The exercise for this section is located in `03_ticket_v1/01_struct`

3.2 Validation

Let's go back to our ticket definition:

```
struct Ticket {
    title: String,
    description: String,
    status: String,
}
```

We are using "raw" types for the fields of our `Ticket` struct. This means that users can create a ticket with an empty title, a suuuuuuuper long description or a nonsensical status (e.g. "Funny").
We can do better than that!

Further reading

- Check out `String`'s documentation for a thorough overview of the methods it provides. You'll need it for the exercise!

Exercise

The exercise for this section is located in `03_ticket_v1/02_validation`

3.3 Modules

The `new` method you've just defined is trying to enforce some **constraints** on the field values for `Ticket`. But are those invariants really enforced? What prevents a developer from creating a `Ticket` without going through `Ticket::new`?

To get proper **encapsulation** you need to become familiar with two new concepts: **visibility** and **modules**. Let's start with modules.

What is a module?

In Rust a **module** is a way to group related code together, under a common namespace (i.e. the module's name).
You've already seen modules in action: the unit tests that verify the correctness of your code are defined in a different module, named `tests`.

```
mod tests {
    // [...]
}
```

Inline modules

The `tests` module above is an example of an **inline module**: the module declaration (`mod tests`) and the module contents (the stuff inside `{ ... }`) are next to each other.

Module tree

Modules can be nested, forming a **tree** structure.
The root of the tree is the **crate** itself, which is the top-level module that contains all the other modules. For a library crate, the root module is usually `src/lib.rs` (unless its location has been customized). The root module is also known as the **crate root**.

The crate root can have submodules, which in turn can have their own submodules, and so on.

External modules and the filesystem

Inline modules are useful for small pieces of code, but as your project grows you'll want to split your code into multiple files. In the parent module, you declare the existence of a submodule using the `mod` keyword.

```rust
mod dog;
```

`cargo`, Rust's build tool, is then in charge of finding the file that contains the module implementation.
If your module is declared in the root of your crate (e.g. `src/lib.rs` or `src/main.rs`), `cargo` expects the file to be named either:

- `src/<module_name>.rs`
- `src/<module_name>/mod.rs`

If your module is a submodule of another module, the file should be named:

- `[..]/<parent_module>/<module_name>.rs`
- `[..]/<parent_module>/<module_name>/mod.rs`

E.g. `src/animals/dog.rs` or `src/animals/dog/mod.rs` if `dog` is a submodule of `animals`.

Your IDE might help you create these files automatically when you declare a new module using the `mod` keyword.

Item paths and use statements

You can access items defined in the same module without any special syntax. You just use their name.

```rust
struct Ticket {
    // [...]
}

// No need to qualify `Ticket` in any way here
// because we're in the same module
fn mark_ticket_as_done(ticket: Ticket) {
    // [...]
}
```

That's not the case if you want to access an entity from a different module.
You have to use a **path** pointing to the entity you want to access.

You can compose the path in various ways:

- starting from the root of the current crate, e.g. `crate::module_1::MyStruct`
- starting from the parent module, e.g. `super::my_function`

- starting from the current module, e.g. `sub_module_1::MyStruct`

Having to write the full path every time you want to refer to a type can be cumbersome. To make your life easier, you can introduce a `use` statement to bring the entity into scope.

```rust
// Bring `MyStruct` into scope
use crate::module_1::module_2::MyStruct;

// Now you can refer to `MyStruct` directly
fn a_function(s: MyStruct) {
     // [...]
}
```

Star imports

You can also import all the items from a module with a single `use` statement.

```rust
use crate::module_1::module_2::*;
```

This is known as a **star import**.
It is generally discouraged because it can pollute the current namespace, making it hard to understand where each name comes from and potentially introducing name conflicts.
Nonetheless, it can be useful in some cases, like when writing unit tests. You might have noticed that most of our test modules start with a `use super::*;` statement to bring all the items from the parent module (the one being tested) into scope.

Visualizing the module tree

If you're struggling to picture the module tree of your project, you can try using `cargo-modules` to visualize it!

Refer to their documentation for installation instructions and usage examples.

Exercise

The exercise for this section is located in `03_ticket_v1/03_modules`

3.4 Visibility

When you start breaking down your code into multiple modules, you need to start thinking about **visibility**. Visibility determines which regions of your code (or other people's code) can access a given entity, be it a struct, a function, a field, etc.

Private by default

By default, everything in Rust is **private**.
A private entity can only be accessed:

1. within the same module where it's defined, or
2. by one of its submodules

We've used this extensively in the previous exercises:

- `create_todo_ticket` worked (once you added a `use` statement) because `helpers` is a submodule of the crate root, where `Ticket` is defined. Therefore, `create_todo_ticket` can access `Ticket` without any issues even though `Ticket` is private.
- All our unit tests are defined in a submodule of the code they're testing, so they can access everything without restrictions.

Visibility modifiers

You can modify the default visibility of an entity using a **visibility modifier**.
Some common visibility modifiers are:

- pub: makes the entity **public**, i.e. accessible from outside the module where it's defined, potentially from other crates.
- `pub(crate)`: makes the entity public within the same **crate**, but not outside of it.
- `pub(super)`: makes the entity public within the parent module.
- `pub(in path::to::module)`: makes the entity public within the specified module.

You can use these modifiers on modules, structs, functions, fields, etc. For example:

```rust
pub struct Configuration {
    pub(crate) version: u32,
    active: bool,
}
```

`Configuration` is public, but you can only access the `version` field from within the same crate. The `active` field, instead, is private and can only be accessed from within the same module or one of its submodules.

Exercise

The exercise for this section is located in `03_ticket_v1/04_visibility`

3.5 Encapsulation

Now that we have a basic understanding of modules and visibility, let's circle back to **encapsulation**.
Encapsulation is the practice of hiding the internal representation of an object. It is most commonly used to enforce some **invariants** on the object's state.

Going back to our `Ticket` struct:

```
struct Ticket {
    title: String,
    description: String,
    status: String,
}
```

If all fields are made public, there is no encapsulation.
You must assume that the fields can be modified at any time, set to any value that's allowed by their type. You can't rule out that a ticket might have an empty title or a status that doesn't make sense.

To enforce stricter rules, we must keep the fields private[1]. We can then provide public methods to interact with a `Ticket` instance. Those public methods will have the responsibility of upholding our invariants (e.g. a title must not be empty).

If at least one field is private it is no longer possible to create a `Ticket` instance directly using the struct instantiation syntax:

```
// This won't work!
let ticket = Ticket {
    title: "Build a ticket system".into(),
    description: "A Kanban board".into(),
    status: "Open".into()
};
```

You've seen this in action in the previous exercise on visibility.
We now need to provide one or more public **constructors**—i.e. static methods or functions that can be used from outside the module to create a new instance of the struct.
Luckily enough we already have one: `Ticket::new`, as implemented in a previous exercise.

Accessor methods

In summary:

[1] Or refine their type, a technique we'll explore later on.

- All `Ticket` fields are private
- We provide a public constructor, `Ticket::new`, that enforces our validation rules on creation

That's a good start, but it's not enough: apart from creating a `Ticket`, we also need to interact with it. But how can we access the fields if they're private?

We need to provide **accessor methods**.
Accessor methods are public methods that allow you to read the value of a private field (or fields) of a struct.

Rust doesn't have a built-in way to generate accessor methods for you, like some other languages do. You have to write them yourself—they're just regular methods.

Exercise

The exercise for this section is located in `03_ticket_v1/05_encapsulation`

3.6 Ownership

If you solved the previous exercise using what this course has taught you so far, your accessor methods probably look like this:

```rust
impl Ticket {
    pub fn title(self) -> String {
        self.title
    }

    pub fn description(self) -> String {
        self.description
    }

    pub fn status(self) -> String {
        self.status
    }
}
```

Those methods compile and are enough to get tests to pass, but in a real-world scenario they won't get you very far. Consider this snippet:

```rust
if ticket.status() == "To-Do" {
    // We haven't covered the `println!` macro yet,
    // but for now it's enough to know that it prints
    // a (templated) message to the console
    println!("Your next task is: {}", ticket.title());
}
```

If you try to compile it, you'll get an error:

```
error[E0382]: use of moved value: `ticket`
  --> src/main.rs:30:43
   |
25 |     let ticket = Ticket::new(/* */);
   |         ------ move occurs because `ticket` has type `Ticket`,
   |                which does not implement the `Copy` trait
26 |     if ticket.status() == "To-Do" {
   |               -------- `ticket` moved due to this method call
...
30 |         println!("Your next task is: {}", ticket.title());
   |                                           ^^^^^^
   |                                value used here after move
   |
note: `Ticket::status` takes ownership of the receiver `self`,
      which moves `ticket`
  --> src/main.rs:12:23
```

```
   |
12 |         pub fn status(self) -> String {
   |                       ^^^^
```

Congrats, this is your first borrow-checker error!

The perks of Rust's ownership system

Rust's ownership system is designed to ensure that:

- Data is never mutated while it's being read
- Data is never read while it's being mutated
- Data is never accessed after it has been destroyed

These constraints are enforced by the **borrow checker**, a subsystem of the Rust compiler, often the subject of jokes and memes in the Rust community.

Ownership is a key concept in Rust, and it's what makes the language unique. Ownership enables Rust to provide **memory safety without compromising performance**. All these things are true at the same time for Rust:

1. There is no runtime garbage collector
2. As a developer, you rarely have to manage memory directly
3. You can't cause dangling pointers, double frees, and other memory-related bugs

Languages like Python, JavaScript, and Java give you 2. and 3., but not 1.
Language like C or C++ give you 1., but neither 2. nor 3.

Depending on your background, 3. might sound a bit arcane: what is a "dangling pointer"? What is a "double free"? Why are they dangerous?
Don't worry: we'll cover these concepts in more details during the rest of the course.

For now, though, let's focus on learning how to work within Rust's ownership system.

The owner

In Rust, each value has an **owner**, statically determined at compile-time. There is only one owner for each value at any given time.

Move semantics

Ownership can be transferred.

If you own a value, for example, you can transfer ownership to another variable:

```rust
let a = "hello, world".to_string(); // <- `a` is the owner of the String
let b = a;  // <- `b` is now the owner of the String
```

Rust's ownership system is baked into the type system: each function has to declare in its signature *how* it wants to interact with its arguments.

So far, all our methods and functions have **consumed** their arguments: they've taken ownership of them. For example:

```rust
impl Ticket {
    pub fn description(self) -> String {
        self.description
    }
}
```

`Ticket::description` takes ownership of the `Ticket` instance it's called on. This is known as **move semantics**: ownership of the value (`self`) is **moved** from the caller to the callee, and the caller can't use it anymore.

That's exactly the language used by the compiler in the error message we saw earlier:

```
error[E0382]: use of moved value: `ticket`
  --> src/main.rs:30:43
   |
25 |     let ticket = Ticket::new(/* */);
   |         ------ move occurs because `ticket` has type `Ticket`,
   |                which does not implement the `Copy` trait
26 |     if ticket.status() == "To-Do" {
   |               -------- `ticket` moved due to this method call
...
30 |         println!("Your next task is: {}", ticket.title());
   |                                           ^^^^^^
   |                               value used here after move
   |
note: `Ticket::status` takes ownership of the receiver `self`,
      which moves `ticket`
  --> src/main.rs:12:23
   |
12 |         pub fn status(self) -> String {
   |                       ^^^^
```

In particular, this is the sequence of events that unfold when we call `ticket.status()`:

- `Ticket::status` takes ownership of the `Ticket` instance
- `Ticket::status` extracts `status` from `self` and transfers ownership of `status` back to the caller
- The rest of the `Ticket` instance is discarded (`title` and `description`)

When we try to use `ticket` again via `ticket.title()`, the compiler complains: the `ticket` value is gone now, we no longer own it, therefore we can't use it anymore.

To build *useful* accessor methods we need to start working with **references**.

Borrowing

It is desirable to have methods that can read the value of a variable without taking ownership of it.
Programming would be quite limited otherwise. In Rust, that's done via **borrowing**.

Whenever you borrow a value, you get a **reference** to it.
References are tagged with their privileges[2]:

- Immutable references (&) allow you to read the value, but not to mutate it
- Mutable references (`&mut`) allow you to read and mutate the value

Going back to the goals of Rust's ownership system:

- Data is never mutated while it's being read
- Data is never read while it's being mutated

To ensure these two properties, Rust has to introduce some restrictions on references:

- You can't have a mutable reference and an immutable reference to the same value at the same time
- You can't have more than one mutable reference to the same value at the same time
- The owner can't mutate the value while it's being borrowed
- You can have as many immutable references as you want, as long as there are no mutable references

In a way, you can think of an immutable reference as a "read-only" lock on the value, while a mutable reference is like a "read-write" lock.

All these restrictions are enforced at compile-time by the borrow checker.

[2]This is a great mental model to start out, but it doesn't capture the *full* picture. We'll refine our understanding of references later in the course.

Syntax

How do you borrow a value, in practice?
By adding & or &mut **in front a variable**, you're borrowing its value. Careful though! The same symbols (& and &mut) in **front of a type** have a different meaning: they denote a different type, a reference to the original type.

For example:

```
struct Configuration {
    version: u32,
    active: bool,
}

fn main() {
    let config = Configuration {
        version: 1,
        active: true,
    };
    // `b` is a reference to the `version` field of `config`.
    // The type of `b` is `&u32`, since it contains a reference to
    // a `u32` value.
    // We create a reference by borrowing `config.version`, using
    // the `&` operator.
    // Same symbol (`&`), different meaning depending on the context!
    let b: &u32 = &config.version;
    //     ^ The type annotation is not necessary,
    //       it's just there to clarify what's going on
}
```

The same concept applies to function arguments and return types:

```
// `f` takes a mutable reference to a `u32` as an argument,
// bound to the name `number`
fn f(number: &mut u32) -> &u32 {
    // [...]
}
```

Breathe in, breathe out

Rust's ownership system can be a bit overwhelming at first.
But don't worry: it'll become second nature with practice.
And you're going to get a lot of practice over the rest of this chapter, as well as the rest of the course! We'll revisit each concept multiple times to make sure you get familiar with them and truly understand how they work.

Towards the end of this chapter we'll explain *why* Rust's ownership system is designed the way it is. For the time being, focus on understanding the *how*. Take each compiler error as a learning opportunity!

Exercise

The exercise for this section is located in `03_ticket_v1/06_ownership`

3.7 Mutable references

Your accessor methods should look like this now:

```
impl Ticket {
    pub fn title(&self) -> &String {
        &self.title
    }

    pub fn description(&self) -> &String {
        &self.description
    }

    pub fn status(&self) -> &String {
        &self.status
    }
}
```

A sprinkle of & here and there did the trick!
We now have a way to access the fields of a `Ticket` instance without consuming it in the process. Let's see how we can enhance our `Ticket` struct with **setter methods** next.

Setters

Setter methods allow users to change the values of `Ticket`'s private fields while making sure that its invariants are respected (i.e. you can't set a `Ticket`'s title to an empty string).

There are two common ways to implement setters in Rust:

- Taking `self` as input.
- Taking `&mut self` as input.

Taking `self` as input

The first approach looks like this:

```
impl Ticket {
    pub fn set_title(mut self, new_title: String) -> Self {
        // Validate the new title [...]
        self.title = new_title;
        self
    }
}
```

It takes ownership of `self`, changes the title, and returns the modified `Ticket` instance.
This is how you'd use it:

```
let ticket = Ticket::new(
    "Title".into(),
    "Description".into(),
    "To-Do".into()
);
let ticket = ticket.set_title("New title".into());
```

Since `set_title` takes ownership of `self` (i.e. it **consumes it**), we need to reassign the result to a variable. In the example above we take advantage of **variable shadowing** to reuse the same variable name: when you declare a new variable with the same name as an existing one, the new variable **shadows** the old one. This is a common pattern in Rust code.

`self`-setters work quite nicely when you need to change multiple fields at once: you can chain multiple calls together!

```
let ticket = ticket
    .set_title("New title".into())
    .set_description("New description".into())
    .set_status("In Progress".into());
```

Taking `&mut self` as input

The second approach to setters, using `&mut self`, looks like this instead:

```
impl Ticket {
    pub fn set_title(&mut self, new_title: String) {
        // Validate the new title [...]

        self.title = new_title;
    }
}
```

This time the method takes a mutable reference to `self` as input, changes the title, and that's it. Nothing is returned.

You'd use it like this:

```
let mut ticket = Ticket::new(
    "Title".into(),
    "Description".into(),
    "To-Do".into()
```

```
);
ticket.set_title("New title".into());

// Use the modified ticket
```

Ownership stays with the caller, so the original `ticket` variable is still valid. We don't need to reassign the result. We need to mark `ticket` as mutable though, because we're taking a mutable reference to it.

`&mut`-setters have a downside: you can't chain multiple calls together. Since they don't return the modified `Ticket` instance, you can't call another setter on the result of the first one. You have to call each setter separately:

```
ticket.set_title("New title".into());
ticket.set_description("New description".into());
ticket.set_status("In Progress".into());
```

Exercise

The exercise for this section is located in `03_ticket_v1/07_setters`

3.8 Memory layout

We've looked at ownership and references from an operational point of view—what you can and can't do with them. Now it's a good time to take a look under the hood: let's talk about **memory**.

Stack and heap

When discussing memory, you'll often hear people talk about the **stack** and the **heap**. These are two different memory regions used by programs to store data.

Let's start with the stack.

Stack

The **stack** is a **LIFO** (Last In, First Out) data structure.
When you call a function, a new **stack frame** is added on top of the stack. That stack frame stores the function's arguments, local variables and a few "bookkeeping" values.
When the function returns, the stack frame is popped off the stack[3].

```
+-----------------+
| frame for func1 |
+-----------------+
        |
        | func2 is
        | called
        v
+-----------------+
| frame for func2 |
+-----------------+
| frame for func1 |
+-----------------+
        |
        | func2
        | returns
        v
+-----------------+
| frame for func1 |
+-----------------+
```

[3]If you have nested function calls, each function pushes its data onto the stack when it's called but it doesn't pop it off until the innermost function returns. If you have too many nested function calls, you can run out of stack space—the stack is not infinite! That's called a **stack overflow**.

From an operational point of view, stack allocation/de-allocation is **very fast**.
We are always pushing and popping data from the top of the stack, so we don't need to search for free memory. We also don't have to worry about fragmentation: the stack is a single contiguous block of memory.

Rust

Rust will often allocate data on the stack.
You have a `u32` input argument in a function? Those 32 bits will be on the stack.
You define a local variable of type `i64`? Those 64 bits will be on the stack.
It all works quite nicely because the size of those integers is known at compile time, therefore the compiled program knows how much space it needs to reserve on the stack for them.

`std::mem::size_of`

You can verify how much space a type would take on the stack using the `std::mem::size_of` function.

For a u8, for example:

```
// We'll explain this funny-looking syntax (`::<u8>`) later on.
// Ignore it for now.
assert_eq!(std::mem::size_of::<u8>(), 1);
```

1 makes sense, because a u8 is 8 bits long, or 1 byte.

Exercise

The exercise for this section is located in `03_ticket_v1/08_stack`

3.9 Heap

The stack is great, but it can't solve all our problems. What about data whose size is not known at compile time? Collections, strings, and other dynamically-sized data cannot be (entirely) stack-allocated. That's where the **heap** comes in.

Heap allocations

You can visualize the heap as a big chunk of memory—a huge array, if you will.
Whenever you need to store data on the heap, you ask a special program, the **allocator**, to reserve for you a subset of the heap. We call this interaction (and the memory you reserved) a **heap allocation**. If the allocation succeeds, the allocator will give you a **pointer** to the start of the reserved block.

No automatic de-allocation

The heap is structured quite differently from the stack.
Heap allocations are not contiguous, they can be located anywhere inside the heap.

```
+---+---+---+---+---+---+-...-+-...-+---+---+---+---+---+---+
|  Allocation 1 | Free  | ... | ... |  Allocation N |    Free   |
+---+---+---+---+---+---+ ... + ... +---+---+---+---+---+---+
```

It's the allocator's job to keep track of which parts of the heap are in use and which are free. The allocator won't automatically free the memory you allocated, though: you need to be deliberate about it, calling the allocator again to **free** the memory you no longer need.

Performance

The heap's flexibility comes at a cost: heap allocations are **slower** than stack allocations. There's a lot more bookkeeping involved!
If you read articles about performance optimization you'll often be advised to minimize heap allocations and prefer stack-allocated data whenever possible.

`String`'s memory layout

When you create a local variable of type `String`, Rust is forced to allocate on the heap[4]: it doesn't know in advance how much text you're going to put in it, so it can't

[4] `std` doesn't allocate if you create an **empty** `String` (i.e. `String::new()`). Heap memory will be reserved when you push data into it for the first time.

reserve the right amount of space on the stack.
But a `String` is not *entirely* heap-allocated, it also keeps some data on the stack. In particular:

- The **pointer** to the heap region you reserved.
- The **length** of the string, i.e. how many bytes are in the string.
- The **capacity** of the string, i.e. how many bytes have been reserved on the heap.

Let's look at an example to understand this better:

```
let mut s = String::with_capacity(5);
```

If you run this code, memory will be laid out like this:

```
      +---------+--------+----------+
Stack | pointer | length | capacity |
      |  |      |   0    |    5     |
      +--|------+--------+----------+
         |
         |
         v
       +---+---+---+---+---+
Heap:  | ? | ? | ? | ? | ? |
       +---+---+---+---+---+
```

We asked for a `String` that can hold up to 5 bytes of text.
`String::with_capacity` goes to the allocator and asks for 5 bytes of heap memory. The allocator returns a pointer to the start of that memory block.
The `String` is empty, though. On the stack, we keep track of this information by distinguishing between the length and the capacity: this `String` can hold up to 5 bytes, but it currently holds 0 bytes of actual text.

If you push some text into the `String`, the situation will change:

```
s.push_str("Hey");
```

```
      +---------+--------+----------+
Stack | pointer | length | capacity |
      |  |      |   3    |    5     |
      +--|  ----+--------+----------+
         |
         |
         v
       +---+---+---+---+---+
Heap:  | H | e | y | ? | ? |
       +---+---+---+---+---+
```

`s` now holds 3 bytes of text. Its length is updated to 3, but capacity remains 5. Three of the five bytes on the heap are used to store the characters `H`, `e`, and `y`.

`usize`

How much space do we need to store pointer, length and capacity on the stack?
It depends on the **architecture** of the machine you're running on.

Every memory location on your machine has an **address**, commonly represented as an unsigned integer. Depending on the maximum size of the address space (i.e. how much memory your machine can address), this integer can have a different size. Most modern machines use either a 32-bit or a 64-bit address space.

Rust abstracts away these architecture-specific details by providing the `usize` type: an unsigned integer that's as big as the number of bytes needed to address memory on your machine. On a 32-bit machine, `usize` is equivalent to `u32`. On a 64-bit machine, it matches `u64`.

Capacity, length and pointers are all represented as `usize`s in Rust[5].

No `std::mem::size_of` for the heap

`std::mem::size_of` returns the amount of space a type would take on the stack, which is also known as the **size of the type**.

> What about the memory buffer that `String` is managing on the heap? Isn't that part of the size of `String`?

No!
That heap allocation is a **resource** that `String` is managing. It's not considered to be part of the `String` type by the compiler.

`std::mem::size_of` doesn't know (or care) about additional heap-allocated data that a type might manage or refer to via pointers, as is the case with `String`, therefore it doesn't track its size.

Unfortunately there is no equivalent of `std::mem::size_of` to measure the amount of heap memory that a certain value is allocating at runtime. Some types might provide methods to inspect their heap usage (e.g. `String`'s `capacity` method), but there is no general-purpose "API" to retrieve runtime heap usage in Rust.
You can, however, use a memory profiler tool (e.g. DHAT or a custom allocator) to inspect the heap usage of your program.

---

[5] The size of a pointer depends on the operating system too. In certain environments, a pointer is **larger** than a memory address (e.g. CHERI). Rust makes the simplifying assumption that pointers are the same size as memory addresses, which is true for most modern systems you're likely to encounter.

Exercise

The exercise for this section is located in `03_ticket_v1/09_heap`

3.10 References

What about references, like `&String` or `&mut String`? How are they represented in memory?

Most references[6] in Rust are represented, in memory, as a pointer to a memory location.
It follows that their size is the same as the size of a pointer, a `usize`.

You can verify this using `std::mem::size_of`:

```rust
assert_eq!(std::mem::size_of::<&String>(), 8);
assert_eq!(std::mem::size_of::<&mut String>(), 8);
```

A `&String`, in particular, is a pointer to the memory location where the `String`'s metadata is stored.
If you run this snippet:

```rust
let s = String::from("Hey");
let r = &s;
```

you'll get something like this in memory:

```
           --------------------------------------
           |                                    |
      +----v----+--------+----------+      +----|----+
Stack | pointer | length | capacity |      | pointer |
      |  |      |   3    |    5     |      |         |
      +--|  ----+--------+----------+      +---------+
         |          s                           r
         |
         v
       +---+---+---+---+---+
Heap   | H | e | y | ? | ? |
       +---+---+---+---+---+
```

It's a pointer to a pointer to the heap-allocated data, if you will. The same goes for `&mut String`.

Not all pointers point to the heap

The example above should clarify one thing: not all pointers point to the heap.
They just point to a memory location, which *may* be on the heap, but doesn't have to be.

[6] Later in the course we'll talk about **fat pointers**, i.e. pointers with additional metadata. As the name implies, they are larger than the pointers we discussed in this chapter, also known as **thin pointers**.

Exercise

The exercise for this section is located in `03_ticket_v1/10_references_in_memory`

3.11 Destructors

When introducing the heap, we mentioned that you're responsible for freeing the memory you allocate.
When introducing the borrow-checker, we also stated that you rarely have to manage memory directly in Rust.

These two statements might seem contradictory at first. Let's see how they fit together by introducing **scopes** and **destructors**.

Scopes

The **scope** of a variable is the region of Rust code where that variable is valid, or **alive**.

The scope of a variable starts with its declaration. It ends when one of the following happens:

1. the block (i.e. the code between `{}`) where the variable was declared ends

```
fn main() {
   // `x` is not yet in scope here
   let y = "Hello".to_string();
   let x = "World".to_string(); // <-- x's scope starts here...
   let h = "!".to_string(); //   |
} //  <-------------- ...and ends here
```

2. ownership of the variable is transferred to someone else (e.g. a function or another variable)

```
fn compute(t: String) {
   // Do something [...]
}

fn main() {
    let s = "Hello".to_string(); // <-- s's scope starts here...
                //                    |
    compute(s); // <------------------- ..and ends here
                //   because `s` is moved into `compute`
}
```

Destructors

When the owner of a value goes out of scope, Rust invokes its **destructor**.
The destructor tries to clean up the resources used by that value—in particular, whatever memory it allocated.

You can manually invoke the destructor of a value by passing it to `std::mem::drop`. That's why you'll often hear Rust developers saying "that value has been **dropped**" as a way to state that a value has gone out of scope and its destructor has been invoked.

Visualizing drop points

We can insert explicit calls to `drop` to "spell out" what the compiler does for us. Going back to the previous example:

```rust
fn main() {
   let y = "Hello".to_string();
   let x = "World".to_string();
   let h = "!".to_string();
}
```

It's equivalent to:

```rust
fn main() {
   let y = "Hello".to_string();
   let x = "World".to_string();
   let h = "!".to_string();
   // Variables are dropped in reverse order of declaration
   drop(h);
   drop(x);
   drop(y);
}
```

Let's look at the second example instead, where `s`'s ownership is transferred to `compute`:

```rust
fn compute(s: String) {
   // Do something [...]
}

fn main() {
   let s = "Hello".to_string();
   compute(s);
}
```

It's equivalent to this:

```rust
fn compute(t: String) {
    // Do something [...]
    drop(t); // <-- Assuming `t` wasn't dropped or moved
             //     before this point, the compiler will call
```

```
            //     `drop` here, when it goes out of scope
}

fn main() {
    let s = "Hello".to_string();
    compute(s);
}
```

Notice the difference: even though `s` is no longer valid after `compute` is called in `main`, there is no `drop(s)` in `main`. When you transfer ownership of a value to a function, you're also **transferring the responsibility of cleaning it up**.

This ensures that the destructor for a value is called **at most**[7] **once**, preventing double free bugs by design.

Use after drop

What happens if you try to use a value after it's been dropped?

```
let x = "Hello".to_string();
drop(x);
println!("{}", x);
```

If you try to compile this code, you'll get an error:

```
error[E0382]: use of moved value: `x`
 --> src/main.rs:4:20
  |
3 |     drop(x);
  |          - value moved here
4 |     println!("{}", x);
  |                    ^ value used here after move
```

Drop **consumes** the value it's called on, meaning that the value is no longer valid after the call.
The compiler will therefore prevent you from using it, avoiding use-after-free bugs.

Dropping references

What if a variable contains a reference?
For example:

[7] Rust doesn't guarantee that destructors will run. They won't, for example, if you explicitly choose to leak memory.

```rust
let x = 42i32;
let y = &x;
drop(y);
```

When you call `drop(y)`... nothing happens.
If you actually try to compile this code, you'll get a warning:

```
warning: calls to `std::mem::drop` with a reference
         instead of an owned value does nothing
 --> src/main.rs:4:5
  |
4 |     drop(y);
  |     ^^^^^-^
  |          |
  |          argument has type `&i32`
  |
```

It goes back to what we said earlier: we only want to call the destructor once.
You can have multiple references to the same value—if we called the destructor for the value they point at when one of them goes out of scope, what would happen to the others? They would refer to a memory location that's no longer valid: a so-called **dangling pointer**, a close relative of **use-after-free bugs**. Rust's ownership system rules out these kinds of bugs by design.

Exercise

The exercise for this section is located in `03_ticket_v1/11_destructor`

3.12 Wrapping up

We've covered a lot of foundational Rust concepts in this chapter.
Before moving on, let's go through one last exercise to consolidate what we've learned. You'll have minimal guidance this time—just the exercise description and the tests to guide you.

Exercise

The exercise for this section is located in `03_ticket_v1/12_outro`

Chapter 4

Traits

In the previous chapter we covered the basics of Rust's type and ownership system. It's time to dig deeper: we'll explore **traits**, Rust's take on interfaces.

Once you learn about traits, you'll start seeing their fingerprints all over the place. In fact, you've already seen traits in action throughout the previous chapter, e.g. `.into()` invocations as well as operators like `==` and `+`.

On top of traits as a concept, we'll also cover some of the key traits that are defined in Rust's standard library:

- Operator traits (e.g. `Add`, `Sub`, `PartialEq`, etc.)
- `From` and `Into`, for infallible conversions
- `Clone` and `Copy`, for copying values
- `Deref` and deref coercion
- `Sized`, to mark types with a known size
- `Drop`, for custom cleanup logic

Since we'll be talking about conversions, we'll seize the opportunity to plug some of the "knowledge gaps" from the previous chapter—e.g. what is `"A title"`, exactly? Time to learn more about slices too!

Exercise

The exercise for this section is located in `04_traits/00_intro`

4.1 Traits

Let's look again at our `Ticket` type:

```
pub struct Ticket {
    title: String,
    description: String,
    status: String,
}
```

All our tests, so far, have been making assertions using `Ticket`'s fields.

```
assert_eq!(ticket.title(), "A new title");
```

What if we wanted to compare two `Ticket` instances directly?

```
let ticket1 = Ticket::new(/* ... */);
let ticket2 = Ticket::new(/* ... */);
ticket1 == ticket2
```

The compiler will stop us:

```
error[E0369]: binary operation `==` cannot be applied to type `Ticket`
  --> src/main.rs:18:13
   |
18 |     ticket1 == ticket2
   |     ------- ^^ ------- Ticket
   |     |
   |     Ticket
   |
note: an implementation of `PartialEq` might be missing for `Ticket`
```

`Ticket` is a new type. Out of the box, there is **no behavior attached to it**.
Rust doesn't magically infer how to compare two `Ticket` instances just because they contain `String`s.

The Rust compiler is nudging us in the right direction though: it's suggesting that we might be missing an implementation of `PartialEq`. `PartialEq` is a **trait**!

What are traits?

Traits are Rust's way of defining **interfaces**.
A trait defines a set of methods that a type must implement to satisfy the trait's contract.

Defining a trait

The syntax for a trait definition goes like this:

```
trait <TraitName> {
    fn <method_name>(<parameters>) -> <return_type>;
}
```

We might, for example, define a trait named `MaybeZero` that requires its implementors to define an `is_zero` method:

```
trait MaybeZero {
    fn is_zero(self) -> bool;
}
```

Implementing a trait

To implement a trait for a type we use the `impl` keyword, just like we do for regular[1] methods, but the syntax is a bit different:

```
impl <TraitName> for <TypeName> {
    fn <method_name>(<parameters>) -> <return_type> {
        // Method body
    }
}
```

For example, to implement the `MaybeZero` trait for a custom number type, `WrappingU32`:

```
pub struct WrappingU32 {
    inner: u32,
}

impl MaybeZero for WrappingU32 {
    fn is_zero(self) -> bool {
        self.inner == 0
    }
}
```

Invoking a trait method

To invoke a trait method, we use the `.` operator, just like we do with regular methods:

[1] A method defined directly on a type, without using a trait, is also known as an **inherent method**.

```rust
let x = WrappingU32 { inner: 5 };
assert!(!x.is_zero());
```

To invoke a trait method, two things must be true:

- The type must implement the trait.
- The trait must be in scope.

To satisfy the latter, you may have to add a `use` statement for the trait:

```rust
use crate::MaybeZero;
```

This is not necessary if:

- The trait is defined in the same module where the invocation occurs.
- The trait is defined in the standard library's **prelude**. The prelude is a set of traits and types that are automatically imported into every Rust program. It's as if `use std::prelude::*;` was added at the beginning of every Rust module.

You can find the list of traits and types in the prelude in the Rust documentation.

Exercise

The exercise for this section is located in `04_traits/01_trait`

4.2 Implementing traits

When a type is defined in another crate (e.g. u32, from Rust's standard library), you can't directly define new methods for it. If you try:

```
impl u32 {
    fn is_even(&self) -> bool {
        self % 2 == 0
    }
}
```

the compiler will complain:

```
error[E0390]: cannot define inherent `impl` for primitive types
  |
1 | impl u32 {
  | ^^^^^^^^
  |
  = help: consider using an extension trait instead
```

Extension trait

An **extension trait** is a trait whose primary purpose is to attach new methods to foreign types, such as u32. That's exactly the pattern you deployed in the previous exercise, by defining the IsEven trait and then implementing it for i32 and u32. You are then free to call is_even on those types as long as IsEven is in scope.

```
// Bring the trait in scope
use my_library::IsEven;

fn main() {
    // Invoke its method on a type that implements it
    if 4.is_even() {
        // [...]
    }
}
```

One implementation

There are limitations to the trait implementations you can write.
The simplest and most straight-forward one: you can't implement the same trait twice, in a crate, for the same type.

For example:

```rust
trait IsEven {
    fn is_even(&self) -> bool;
}

impl IsEven for u32 {
    fn is_even(&self) -> bool {
        true
    }
}

impl IsEven for u32 {
    fn is_even(&self) -> bool {
        false
    }
}
```

The compiler will reject it:

```
error[E0119]: conflicting implementations of trait `IsEven` for type `u32`
   |
5  | impl IsEven for u32 {
   | ------------------- first implementation here
...
11 | impl IsEven for u32 {
   | ^^^^^^^^^^^^^^^^^^^ conflicting implementation for `u32`
```

There can be no ambiguity as to what trait implementation should be used when `IsEven::is_even` is invoked on a `u32` value, therefore there can only be one.

Orphan rule

Things get more nuanced when multiple crates are involved. In particular, at least one of the following must be true:

- The trait is defined in the current crate
- The implementor type is defined in the current crate

This is known as Rust's **orphan rule**. Its goal is to make the method resolution process unambiguous.

Imagine the following situation:

- Crate `A` defines the `IsEven` trait
- Crate `B` implements `IsEven` for `u32`
- Crate `C` provides a (different) implementation of the `IsEven` trait for `u32`

- Crate `D` depends on both `B` and `C` and calls `1.is_even()`

Which implementation should be used? The one defined in `B`? Or the one defined in `C`?
There's no good answer, therefore the orphan rule was defined to prevent this scenario. Thanks to the orphan rule, neither crate `B` nor crate `C` would compile.

Further reading

- There are some caveats and exceptions to the orphan rule as stated above. Check out the reference if you want to get familiar with its nuances.

Exercise

The exercise for this section is located in `04_traits/02_orphan_rule`

4.3 Operator overloading

Now that we have a basic understanding of what traits are, let's circle back to **operator overloading**. Operator overloading is the ability to define custom behavior for operators like +, -, *, /, ==, !=, etc.

Operators are traits

In Rust, operators are traits.
For each operator, there is a corresponding trait that defines the behavior of that operator. By implementing that trait for your type, you **unlock** the usage of the corresponding operators.

For example, the `PartialEq` trait defines the behavior of the == and != operators:

```
// The `PartialEq` trait definition, from Rust's standard library
// (It is *slightly* simplified, for now)
pub trait PartialEq {
    // Required method
    //
    // `Self` is a Rust keyword that stands for
    // "the type that is implementing the trait"
    fn eq(&self, other: &Self) -> bool;

    // Provided method
    fn ne(&self, other: &Self) -> bool { ... }
}
```

When you write `x == y` the compiler will look for an implementation of the `PartialEq` trait for the types of `x` and `y` and replace `x == y` with `x.eq(y)`. It's syntactic sugar!

This is the correspondence for the main operators:

Operator	Trait
+	`Add`
-	`Sub`
*	`Mul`
/	`Div`
%	`Rem`
== and !=	`PartialEq`
<, >, <=, and >=	`PartialOrd`

Arithmetic operators live in the `std::ops` module, while comparison ones live in the `std::cmp` module.

Default implementations

The comment on `PartialEq::ne` states that "ne is a provided method".
It means that `PartialEq` provides a **default implementation** for ne in the trait definition—the `{ ... }` elided block in the definition snippet.
If we expand the elided block, it looks like this:

```rust
pub trait PartialEq {
    fn eq(&self, other: &Self) -> bool;

    fn ne(&self, other: &Self) -> bool {
        !self.eq(other)
    }
}
```

It's what you expect: ne is the negation of eq.
Since a default implementation is provided, you can skip implementing ne when you implement `PartialEq` for your type. It's enough to implement eq:

```rust
struct WrappingU8 {
    inner: u8,
}

impl PartialEq for WrappingU8 {
    fn eq(&self, other: &WrappingU8) -> bool {
        self.inner == other.inner
    }

    // No `ne` implementation here
}
```

You are not forced to use the default implementation though. You can choose to override it when you implement the trait:

```rust
struct MyType;

impl PartialEq for MyType {
    fn eq(&self, other: &MyType) -> bool {
        // Custom implementation
    }

    fn ne(&self, other: &MyType) -> bool {
        // Custom implementation
    }
}
```

Exercise

The exercise for this section is located in `04_traits/03_operator_overloading`

4.4 Derive macros

Implementing `PartialEq` for `Ticket` was a bit tedious, wasn't it? You had to manually compare each field of the struct.

Destructuring syntax

Furthermore, the implementation is brittle: if the struct definition changes (e.g. a new field is added), you have to remember to update the `PartialEq` implementation.

You can mitigate the risk by **destructuring** the struct into its fields:

```
impl PartialEq for Ticket {
    fn eq(&self, other: &Self) -> bool {
        let Ticket {
            title,
            description,
            status,
        } = self;
        // [...]
    }
}
```

If the definition of `Ticket` changes, the compiler will error out, complaining that your destructuring is no longer exhaustive.
You can also rename struct fields, to avoid variable shadowing:

```
impl PartialEq for Ticket {
    fn eq(&self, other: &Self) -> bool {
        let Ticket {
            title,
            description,
            status,
        } = self;
        let Ticket {
            title: other_title,
            description: other_description,
            status: other_status,
        } = other;
        // [...]
    }
}
```

Destructuring is a useful pattern to have in your toolkit, but there's an even more convenient way to do this: **derive macros**.

Macros

You've already encountered a few macros in past exercises:

- `assert_eq!` and `assert!`, in the test cases
- `println!`, to print to the console

Rust macros are **code generators**.
They generate new Rust code based on the input you provide, and that generated code is then compiled alongside the rest of your program. Some macros are built into Rust's standard library, but you can also write your own. We won't be creating our own macro in this course, but you can find some useful pointers in the "Further reading" section.

Inspection

Some IDEs let you expand a macro to inspect the generated code. If that's not possible, you can use `cargo-expand`.

Derive macros

A **derive macro** is a particular flavour of Rust macro. It is specified as an **attribute** on top of a struct.

```
struct Ticket {
    title: String,
    description: String,
    status: String
}
```

Derive macros are used to automate the implementation of common (and "obvious") traits for custom types. In the example above, the `PartialEq` trait is automatically implemented for `Ticket`. If you expand the macro, you'll see that the generated code is functionally equivalent to the one you wrote manually, although a bit more cumbersome to read:

```
impl ::core::cmp::PartialEq for Ticket {
    fn eq(&self, other: &Ticket) -> bool {
        self.title == other.title
            && self.description == other.description
            && self.status == other.status
    }
}
```

The compiler will nudge you to derive traits when possible.

Further reading

- The little book of Rust macros
- Proc macro workshop

Exercise

The exercise for this section is located in `04_traits/04_derive`

4.5 Trait bounds

We've seen two use cases for traits so far:

- Unlocking "built-in" behaviour (e.g. operator overloading)
- Adding new behaviour to existing types (i.e. extension traits)

There's a third use case: **generic programming**.

The problem

All our functions and methods, so far, have been working with **concrete types**.
Code that operates on concrete types is usually straightforward to write and understand. But it's also limited in its reusability.
Let's imagine, for example, that we want to write a function that returns `true` if an integer is even. Working with concrete types, we'd have to write a separate function for each integer type we want to support:

```rust
fn is_even_i32(n: i32) -> bool {
    n % 2 == 0
}

fn is_even_i64(n: i64) -> bool {
    n % 2 == 0
}

// Etc.
```

Alternatively, we could write a single extension trait and then different implementations for each integer type:

```rust
trait IsEven {
    fn is_even(&self) -> bool;
}

impl IsEven for i32 {
    fn is_even(&self) -> bool {
        self % 2 == 0
    }
}

impl IsEven for i64 {
    fn is_even(&self) -> bool {
        self % 2 == 0
    }
```

```
}

// Etc.
```

The duplication remains.

Generic programming

We can do better using **generics**.
Generics allow us to write code that works with a **type parameter** instead of a concrete type:

```
fn print_if_even<T>(n: T)
where
    T: IsEven + Debug
{
    if n.is_even() {
        println!("{n:?} is even");
    }
}
```

`print_if_even` is a **generic function**.
It isn't tied to a specific input type. Instead, it works with any type `T` that:

- Implements the `IsEven` trait.
- Implements the `Debug` trait.

This contract is expressed with a **trait bound**: `T: IsEven + Debug`.
The `+` symbol is used to require that `T` implements multiple traits. `T: IsEven + Debug` is equivalent to "where `T` implements `IsEven` **and** `Debug`".

Trait bounds

What purpose do trait bounds serve in `print_if_even`?
To find out, let's try to remove them:

```
fn print_if_even<T>(n: T) {
    if n.is_even() {
        println!("{n:?} is even");
    }
}
```

This code won't compile:

```
error[E0599]: no method named `is_even` found for type parameter `T`
              in the current scope
 --> src/lib.rs:2:10
  |
1 | fn print_if_even<T>(n: T) {
  |                 - method `is_even` not found
  |                   for this type parameter
2 |     if n.is_even() {
  |          ^^^^^^^ method not found in `T`

error[E0277]: `T` doesn't implement `Debug`
 --> src/lib.rs:3:19
  |
3 |         println!("{n:?} is even");
  |                   ^^^^^
  |   `T` cannot be formatted using `{:?}` because
  |         it doesn't implement `Debug`
  |
help: consider restricting type parameter `T`
  |
1 | fn print_if_even<T: std::fmt::Debug>(n: T) {
  |                   +++++++++++++++++
```

Without trait bounds, the compiler doesn't know what `T` **can do**.
It doesn't know that `T` has an `is_even` method, and it doesn't know how to format `T` for printing. From the compiler point of view, a bare `T` has no behaviour at all.
Trait bounds restrict the set of types that can be used by ensuring that the behaviour required by the function body is present.

Syntax: inlining trait bounds

All the examples above used a **`where` clause** to specify trait bounds:

```rust
fn print_if_even<T>(n: T)
where
    T: IsEven + Debug
//  ^^^^^^^^^^^^^^^^^
//  This is a `where` clause
{
    // [...]
}
```

If the trait bounds are simple, you can **inline** them directly next to the type parameter:

```
fn print_if_even<T: IsEven + Debug>(n: T) {
    //          ^^^^^^^^^^^^^^^^^
    //          This is an inline trait bound
    // [...]
}
```

Syntax: meaningful names

In the examples above, we used T as the type parameter name. This is a common convention when a function has only one type parameter.
Nothing stops you from using a more meaningful name, though:

```
fn print_if_even<Number: IsEven + Debug>(n: Number) {
    // [...]
}
```

It is actually **desirable** to use meaningful names when there are multiple type parameters at play or when the name T doesn't convey enough information about the type's role in the function. Maximize clarity and readability when naming type parameters, just as you would with variables or function parameters. Follow Rust's conventions, though: use upper camel case for type parameter names.

The function signature is king

You may wonder why we need trait bounds at all. Can't the compiler infer the required traits from the function's body?
It could, but it won't.
The rationale is the same as for explicit type annotations on function parameters: each function signature is a contract between the caller and the callee, and the terms must be explicitly stated. This allows for better error messages, better documentation, less unintentional breakages across versions, and faster compilation times.

Exercise

The exercise for this section is located in `04_traits/05_trait_bounds`

4.6 String slices

Throughout the previous chapters you've seen quite a few **string literals** being used in the code, like `"To-Do"` or `"A ticket description"`. They were always followed by a call to `.to_string()` or `.into()`. It's time to understand why!

String literals

You define a string literal by enclosing the raw text in double quotes:

```
let s = "Hello, world!";
```

The type of `s` is `&str`, a **reference to a string slice**.

Memory layout

`&str` and `String` are different types—they're not interchangeable.
Let's recall the memory layout of a `String` from our previous exploration. If we run:

```
let mut s = String::with_capacity(5);
s.push_str("Hello");
```

we'll get this scenario in memory:

```
      +---------+--------+----------+
Stack | pointer | length | capacity |
      |  |      |   5    |    5     |
      +--|------+--------+----------+
         |
         |
         v
       +---+---+---+---+---+
Heap:  | H | e | l | l | o |
       +---+---+---+---+---+
```

If you remember, we've also examined how a `&String` is laid out in memory:

```
     -------------------------------------
     |                                    |
+----v----+--------+----------+      +----|----+
| pointer | length | capacity |      | pointer |
|    |    |   5    |    5     |      |         |
+----|----+--------+----------+      +---------+
```

```
    |        s                          &s
    |
    v
  +---+---+---+---+---+
  | H | e | l | l | o |
  +---+---+---+---+---+
```

`&String` points to the memory location where the `String`'s metadata is stored.
If we follow the pointer, we get to the heap-allocated data. In particular, we get to the first byte of the string, `H`.

What if we wanted a type that represents a **substring** of `s`? E.g. `ello` in `Hello`?

String slices

A `&str` is a **view** into a string, a **reference** to a sequence of UTF-8 bytes stored elsewhere. You can, for example, create a `&str` from a `String` like this:

```
let mut s = String::with_capacity(5);
s.push_str("Hello");
// Create a string slice reference from the `String`,
// skipping the first byte.
let slice: &str = &s[1..];
```

In memory, it'd look like this:

```
                    s                              slice
      +---------+--------+----------+      +---------+--------+
Stack | pointer | length | capacity |      | pointer | length |
      |    |    |   5    |    5     |      |    |    |   4    |
      +----|----+--------+----------+      +----|----+--------+
           |        s                           |
           |                                    |
           v                                    |
         +---+---+---+---+---+                  |
Heap:    | H | e | l | l | o |                  |
         +---+---+---+---+---+                  |
               ^                                |
               |                                |
               +--------------------------------+
```

`slice` stores two pieces of information on the stack:

- A pointer to the first byte of the slice.
- The length of the slice.

`slice` doesn't own the data, it just points to it: it's a **reference** to the `String`'s heap-allocated data.
When `slice` is dropped, the heap-allocated data won't be deallocated, because it's still owned by `s`. That's why `slice` doesn't have a `capacity` field: it doesn't own the data, so it doesn't need to know how much space it was allocated for it; it only cares about the data it references.

`&str` vs `&String`

As a rule of thumb, use `&str` rather than `&String` whenever you need a reference to textual data.
`&str` is more flexible and generally considered more idiomatic in Rust code.

If a method returns a `&String`, you're promising that there is heap-allocated UTF-8 text somewhere that **matches exactly** the one you're returning a reference to.
If a method returns a `&str`, instead, you have a lot more freedom: you're just saying that *somewhere* there's a bunch of text data and that a subset of it matches what you need, therefore you're returning a reference to it.

Exercise

The exercise for this section is located in `04_traits/06_str_slice`

4.7 Deref trait

In the previous exercise you didn't have to do much, did you?

Changing

```rust
impl Ticket {
    pub fn title(&self) -> &String {
        &self.title
    }
}
```

to

```rust
impl Ticket {
    pub fn title(&self) -> &str {
        &self.title
    }
}
```

was all you needed to do to get the code to compile and the tests to pass. Some alarm bells should be ringing in your head though.

It shouldn't work, but it does

Let's review the facts:

- `self.title` is a `String`
- `&self.title` is, therefore, a `&String`
- The output of the (modified) `title` method is `&str`

You would expect a compiler error, wouldn't you? `Expected &String, found &str` or something similar. Instead, it just works. **Why**?

`Deref` to the rescue

The `Deref` trait is the mechanism behind the language feature known as **deref coercion**.
The trait is defined in the standard library, in the `std::ops` module:

```rust
// I've slightly simplified the definition for now.
// We'll see the full definition later on.
pub trait Deref {
    type Target;

    fn deref(&self) -> &Self::Target;
}
```

`type Target` is an **associated type**.
It's a placeholder for a concrete type that must be specified when the trait is implemented.

Deref coercion

By implementing `Deref<Target = U>` for a type `T` you're telling the compiler that `&T` and `&U` are somewhat interchangeable.
In particular, you get the following behavior:

- References to `T` are implicitly converted into references to `U` (i.e. `&T` becomes `&U`)
- You can call on `&T` all the methods defined on `U` that take `&self` as input.

There is one more thing around the dereference operator, `*`, but we don't need it yet (see `std`'s docs if you're curious).

`String` implements `Deref`

`String` implements `Deref` with `Target = str`:

```rust
impl Deref for String {
    type Target = str;

    fn deref(&self) -> &str {
        // [...]
    }
}
```

Thanks to this implementation and deref coercion, a `&String` is automatically converted into a `&str` when needed.

Don't abuse deref coercion

Deref coercion is a powerful feature, but it can lead to confusion.
Automatically converting types can make the code harder to read and understand. If a method with the same name is defined on both `T` and `U`, which one will be called?

We'll examine later in the course the "safest" use cases for deref coercion: smart pointers.

Exercise

The exercise for this section is located in `04_traits/07_deref`

4.8 Sized

There's more to `&str` than meets the eye, even after having investigated deref coercion.
From our previous discussion on memory layouts, it would have been reasonable to expect `&str` to be represented as a single `usize` on the stack, a pointer. That's not the case though. `&str` stores some **metadata** next to the pointer: the length of the slice it points to. Going back to the example from a previous section:

```rust
let mut s = String::with_capacity(5);
s.push_str("Hello");
// Create a string slice reference from the `String`,
// skipping the first byte.
let slice: &str = &s[1..];
```

In memory, we get:

```
                    s                              slice
      +---------+--------+----------+      +---------+--------+
Stack | pointer | length | capacity |      | pointer | length |
      |    |    |   5    |    5     |      |    |    |   4    |
      +----|----+--------+----------+      +----|----+--------+
           |        s                           |
           |                                    |
           v                                    |
         +---+---+---+---+---+                  |
Heap:    | H | e | l | l | o |                  |
         +---+---+---+---+---+                  |
               ^                                |
               |                                |
               +--------------------------------+
```

What's going on?

Dynamically sized types

`str` is a **dynamically sized type** (DST).
A DST is a type whose size is not known at compile time. Whenever you have a reference to a DST, like `&str`, it has to include additional information about the data it points to. It is a **fat pointer**.
In the case of `&str`, it stores the length of the slice it points to. We'll see more examples of DSTs in the rest of the course.

The `Sized` trait

Rust's `std` library defines a trait called `Sized`.

```rust
pub trait Sized {
    // This is an empty trait, no methods to implement.
}
```

A type is `Sized` if its size is known at compile time. In other words, it's not a DST.

Marker traits

`Sized` is your first example of a **marker trait**.
A marker trait is a trait that doesn't require any methods to be implemented. It doesn't define any behavior. It only serves to **mark** a type as having certain properties. The mark is then leveraged by the compiler to enable certain behaviors or optimizations.

Auto traits

In particular, `Sized` is also an **auto trait**.
You don't need to implement it explicitly; the compiler implements it automatically for you based on the type's definition.

Examples

All the types we've seen so far are `Sized`: `u32`, `String`, `bool`, etc.

`str`, as we just saw, is not `Sized`.
`&str` is `Sized` though! We know its size at compile time: two `usize`s, one for the pointer and one for the length.

Exercise

The exercise for this section is located in `04_traits/08_sized`

4.9 From and Into

Let's go back to where our string journey started:

```
let ticket = Ticket::new(
    "A title".into(),
    "A description".into(),
    "To-Do".into()
);
```

We now know enough to start unpacking what `.into()` is doing here.

The problem

This is the signature of the `new` method:

```
impl Ticket {
    pub fn new(
        title: String,
        description: String,
        status: String
    ) -> Self {
        // [...]
    }
}
```

We've also seen that string literals (such as `"A title"`) are of type `&str`.
We have a type mismatch here: a `String` is expected, but we have a `&str`. No magical coercion will come to save us this time; we need **to perform a conversion**.

From and Into

The Rust standard library defines two traits for **infallible conversions**: `From` and `Into`, in the `std::convert` module.

```
pub trait From<T>: Sized {
    fn from(value: T) -> Self;
}

pub trait Into<T>: Sized {
    fn into(self) -> T;
}
```

These trait definitions showcase a few concepts that we haven't seen before: **supertraits** and **implicit trait bounds**. Let's unpack those first.

Supertrait / Subtrait

The `From: Sized` syntax implies that `From` is a **subtrait** of `Sized`: any type that implements `From` must also implement `Sized`. Alternatively, you could say that `Sized` is a **supertrait** of `From`.

Implicit trait bounds

Every time you have a generic type parameter, the compiler implicitly assumes that it's `Sized`.

For example:

```
pub struct Foo<T> {
    inner: T,
}
```

is actually equivalent to:

```
pub struct Foo<T: Sized>
{
    inner: T,
}
```

In the case of `From<T>`, the trait definition is equivalent to:

```
pub trait From<T: Sized>: Sized {
    fn from(value: T) -> Self;
}
```

In other words, *both* `T` and the type implementing `From<T>` must be `Sized`, even though the former bound is implicit.

Negative trait bounds

You can opt out of the implicit `Sized` bound with a **negative trait bound**:

```
pub struct Foo<T: ?Sized> {
    //            ^^^^^^^
    //            This is a negative trait bound
    inner: T,
}
```

This syntax reads as "`T` may or may not be `Sized`", and it allows you to bind `T` to a DST (e.g. `Foo<str>`). It is a special case, though: negative trait bounds are exclusive to `Sized`, you can't use them with other traits.

`&str` to `String`

In `std`'s documentation you can see which `std` types implement the `From` trait. You'll find that `String` implements `From<&str> for String`. Thus, we can write:

```
let title = String::from("A title");
```

We've been primarily using `.into()`, though.
If you check out the implementors of `Into` you won't find `Into<String> for &str`. What's going on?

`From` and `Into` are **dual traits**.
In particular, `Into` is implemented for any type that implements `From` using a **blanket implementation**:

```
impl<T, U> Into<U> for T
where
    U: From<T>,
{
    fn into(self) -> U {
        U::from(self)
    }
}
```

If a type `U` implements `From<T>`, then `Into<U> for T` is automatically implemented.
That's why we can write `let title = "A title".into();`.

`.into()`

Every time you see `.into()`, you're witnessing a conversion between types.
What's the target type, though?

In most cases, the target type is either:

- Specified by the signature of a function/method (e.g. `Ticket::new` in our example above)
- Specified in the variable declaration with a type annotation (e.g. `let title: String = "A title".into();`)

`.into()` will work out of the box as long as the compiler can infer the target type from the context without ambiguity.

Exercise

The exercise for this section is located in `04_traits/09_from`

4.10 Generics and associated types

Let's re-examine the definition for two of the traits we studied so far, `From` and `Deref`:

```rust
pub trait From<T> {
    fn from(value: T) -> Self;
}

pub trait Deref {
    type Target;

    fn deref(&self) -> &Self::Target;
}
```

They both feature type parameters.
In the case of `From`, it's a generic parameter, `T`.
In the case of `Deref`, it's an associated type, `Target`.

What's the difference? Why use one over the other?

At most one implementation

Due to how deref coercion works, there can only be one "target" type for a given type. E.g. `String` can only deref to `str`. It's about avoiding ambiguity: if you could implement `Deref` multiple times for a type, which `Target` type should the compiler choose when you call a `&self` method?

That's why `Deref` uses an associated type, `Target`.
An associated type is uniquely determined **by the trait implementation**. Since you can't implement `Deref` more than once, you'll only be able to specify one `Target` for a given type and there won't be any ambiguity.

Generic traits

On the other hand, you can implement `From` multiple times for a type, **as long as the input type `T` is different**. For example, you can implement `From` for `WrappingU32` using both `u32` and `u16` as input types:

```rust
impl From<u32> for WrappingU32 {
    fn from(value: u32) -> Self {
        WrappingU32 { inner: value }
    }
}

impl From<u16> for WrappingU32 {
```

```
    fn from(value: u16) -> Self {
        WrappingU32 { inner: value.into() }
    }
}
```

This works because `From<u16>` and `From<u32>` are considered **different traits**. There is no ambiguity: the compiler can determine which implementation to use based on type of the value being converted.

Case study: Add

As a closing example, consider the Add trait from the standard library:

```
pub trait Add<RHS = Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
```

It uses both mechanisms:

- it has a generic parameter, RHS (right-hand side), which defaults to `Self`
- it has an associated type, `Output`, the type of the result of the addition

RHS

RHS is a generic parameter to allow for different types to be added together. For example, you'll find these two implementations in the standard library:

```
impl Add<u32> for u32 {
    type Output = u32;

    fn add(self, rhs: u32) -> u32 {
      //                      ^^^
      // This could be written as `Self::Output` instead.
      // The compiler doesn't care, as long as the type you
      // specify here matches the type you assigned to `Output`
      // right above.
      // [...]
    }
}

impl Add<&u32> for u32 {
    type Output = u32;
```

```
    fn add(self, rhs: &u32) -> u32 {
        // [...]
    }
}
```

This allows the following code to compile:

```
let x = 5u32 + &5u32 + 6u32;
```

because `u32` implements `Add<&u32>` *as well as* `Add<u32>`.

`Output`

`Output` represents the type of the result of the addition.

Why do we need `Output` in the first place? Can't we just use `Self` as output, the type implementing `Add`? We could, but it would limit the flexibility of the trait. In the standard library, for example, you'll find this implementation:

```
impl Add<&u32> for &u32 {
    type Output = u32;

    fn add(self, rhs: &u32) -> u32 {
        // [...]
    }
}
```

The type they're implementing the trait for is `&u32`, but the result of the addition is `u32`.
It would be impossible[2] to provide this implementation if `add` had to return `Self`, i.e. `&u32` in this case. `Output` lets `std` decouple the implementor from the return type, thus supporting this case.

On the other hand, `Output` can't be a generic parameter. The output type of the operation **must** be uniquely determined once the types of the operands are known. That's why it's an associated type: for a given combination of implementor and generic parameters, there is only one `Output` type.

Conclusion

To recap:

[2]Flexibility is rarely free: the trait definition is more complex due to `Output`, and implementors have to reason about what they want to return. The trade-off is only justified if that flexibility is actually needed. Keep that in mind when designing your own traits.

- Use an **associated type** when the type must be uniquely determined for a given trait implementation.
- Use a **generic parameter** when you want to allow multiple implementations of the trait for the same type, with different input types.

Exercise

The exercise for this section is located in `04_traits/10_assoc_vs_generic`

4.11 Copying values, pt. 1

In the previous chapter we introduced ownership and borrowing.
We stated, in particular, that:

- Every value in Rust has a single owner at any given time.
- When a function takes ownership of a value ("it consumes it"), the caller can't use that value anymore.

These restrictions can be somewhat limiting.
Sometimes we might have to call a function that takes ownership of a value, but we still need to use that value afterward.

```
fn consumer(s: String) { /* */ }

fn example() {
     let mut s = String::from("hello");
     consumer(s);
     s.push_str(", world!"); // error: value borrowed here after move
}
```

That's where `Clone` comes in.

`Clone`

`Clone` is a trait defined in Rust's standard library:

```
pub trait Clone {
    fn clone(&self) -> Self;
}
```

Its method, `clone`, takes a reference to `self` and returns a new **owned** instance of the same type.

In action

Going back to the example above, we can use `clone` to create a new `String` instance before calling `consumer`:

```rust
fn consumer(s: String) { /* */ }

fn example() {
     let mut s = String::from("hello");
     let t = s.clone();
     consumer(t);
     s.push_str(", world!"); // no error
}
```

Instead of giving ownership of `s` to `consumer`, we create a new `String` (by cloning `s`) and give that to `consumer` instead.
`s` remains valid and usable after the call to `consumer`.

In memory

Let's look at what happened in memory in the example above. When `let mut s: String::from("hello");` is executed, the memory looks like this:

```
                    s
      +---------+--------+----------+
Stack | pointer | length | capacity |
      |  |      |   5    |    5     |
      +--|------+--------+----------+
         |
         |
         v
       +---+---+---+---+---+
Heap:  | H | e | l | l | o |
       +---+---+---+---+---+
```

When `let t = s.clone()` is executed, a whole new region is allocated on the heap to store a copy of the data:

```
                    s                                     t
      +---------+--------+----------+      +---------+--------+----------+
Stack | pointer | length | capacity |      | pointer | length | capacity |
      |  |      |   5    |    5     |      |  |      |   5    |    5     |
      +--|------+--------+----------+      +--|------+--------+----------+
         |                                    |
         |                                    |
         v                                    v
       +---+---+---+---+---+                +---+---+---+---+---+
Heap:  | H | e | l | l | o |                | H | e | l | l | o |
       +---+---+---+---+---+                +---+---+---+---+---+
```

If you're coming from a language like Java, you can think of `clone` as a way to create a deep copy of an object.

Implementing `Clone`

To make a type `Clone`-able, we have to implement the `Clone` trait for it.
You almost always implement `Clone` by deriving it:

```
struct MyType {
    // fields
}
```

The compiler implements `Clone` for `MyType` as you would expect: it clones each field of `MyType` individually and then constructs a new `MyType` instance using the cloned fields.
Remember that you can use `cargo expand` (or your IDE) to explore the code generated by `derive` macros.

Exercise

The exercise for this section is located in `04_traits/11_clone`

4.12 Copying values, pt. 2

Let's consider the same example as before, but with a slight twist: using `u32` rather than `String` as a type.

```rust
fn consumer(s: u32) { /* */ }

fn example() {
     let s: u32 = 5;
     consumer(s);
     let t = s + 1;
}
```

It'll compile without errors! What's going on here? What's the difference between `String` and `u32` that makes the latter work without `.clone()`?

Copy

`Copy` is another trait defined in Rust's standard library:

```rust
pub trait Copy: Clone { }
```

It is a marker trait, just like `Sized`.

If a type implements `Copy`, there's no need to call `.clone()` to create a new instance of the type: Rust does it **implicitly** for you.
`u32` is an example of a type that implements `Copy`, which is why the example above compiles without errors: when `consumer(s)` is called, Rust creates a new `u32` instance by performing a **bitwise copy** of `s`, and then passes that new instance to `consumer`. It all happens behind the scenes, without you having to do anything.

What can be Copy?

`Copy` is not equivalent to "automatic cloning", although it implies it.
Types must meet a few requirements in order to be allowed to implement `Copy`.

First of all, it must implement `Clone`, since `Copy` is a subtrait of `Clone`. This makes sense: if Rust can create a new instance of a type *implicitly*, it should also be able to create a new instance *explicitly* by calling `.clone()`.

That's not all, though. A few more conditions must be met:

1. The type doesn't manage any *additional* resources (e.g. heap memory, file handles, etc.) beyond the `std::mem::size_of` bytes that it occupies in memory.

2. The type is not a mutable reference (`&mut T`).

If both conditions are met, then Rust can safely create a new instance of the type by performing a **bitwise copy** of the original instance—this is often referred to as a `memcpy` operation, after the C standard library function that performs the bitwise copy.

Case study 1: `String`

`String` is a type that doesn't implement `Copy`.
Why? Because it manages an additional resource: the heap-allocated memory buffer that stores the string's data.

Let's imagine that Rust allowed `String` to implement `Copy`.
Then, when a new `String` instance is created by performing a bitwise copy of the original instance, both the original and the new instance would point to the same memory buffer:

```
              s                                 copied_s
+---------+--------+----------+      +---------+--------+----------+
| pointer | length | capacity |      | pointer | length | capacity |
|  |      |   5    |    5     |      |  |      |   5    |    5     |
+--|------+--------+----------+      +--|------+--------+----------+
   |                                    |
   |                                    |
   v                                    |
 +---+---+---+---+---+                  |
 | H | e | l | l | o |                  |
 +---+---+---+---+---+                  |
   ^                                    |
   |                                    |
   +------------------------------------+
```

This is bad! Both `String` instances would try to free the memory buffer when they go out of scope, leading to a double-free error. You could also create two distinct `&mut String` references that point to the same memory buffer, violating Rust's borrowing rules.

Case study 2: `u32`

`u32` implements `Copy`. All integer types do, in fact.
An integer is "just" the bytes that represent the number in memory. There's nothing more! If you copy those bytes, you get another perfectly valid integer instance. Nothing bad can happen, so Rust allows it.

Case study 3: `&mut u32`

When we introduced ownership and mutable borrows, we stated one rule quite clearly: there can only ever be *one* mutable borrow of a value at any given time. That's why `&mut u32` doesn't implement `Copy`, even though `u32` does.

If `&mut u32` implemented `Copy`, you could create multiple mutable references to the same value and modify it in multiple places at the same time. That'd be a violation of Rust's borrowing rules! It follows that `&mut T` never implements `Copy`, no matter what `T` is.

Implementing Copy

In most cases, you don't need to manually implement `Copy`. You can just derive it, like this:

```rust
struct MyStruct {
    field: u32,
}
```

Exercise

The exercise for this section is located in `04_traits/12_copy`

4.13 The Drop trait

When we introduced destructors, we mentioned that the `drop` function:

1. reclaims the memory occupied by the type (i.e. `std::mem::size_of` bytes)
2. cleans up any additional resources that the value might be managing (e.g. the heap buffer of a `String`)

Step 2. is where the `Drop` trait comes in.

```
pub trait Drop {
    fn drop(&mut self);
}
```

The `Drop` trait is a mechanism for you to define *additional* cleanup logic for your types, beyond what the compiler does for you automatically.
Whatever you put in the `drop` method will be executed when the value goes out of scope.

Drop and Copy

When talking about the `Copy` trait, we said that a type can't implement `Copy` if it manages additional resources beyond the `std::mem::size_of` bytes that it occupies in memory.

You might wonder: how does the compiler know if a type manages additional resources? That's right: `Drop` trait implementations!
If your type has an explicit `Drop` implementation, the compiler will assume that your type has additional resources attached to it and won't allow you to implement `Copy`.

```
// This is a unit struct, i.e. a struct with no fields.
struct MyType;

impl Drop for MyType {
    fn drop(&mut self) {
        // We don't need to do anything here,
        // it's enough to have an "empty" Drop implementation
    }
}
```

The compiler will complain with this error message:

```
error[E0184]: the trait `Copy` cannot be implemented for this type;
              the type has a destructor
 --> src/lib.rs:2:17
  |
2 | #[derive(Clone, Copy)]
  |                 ^^^^ `Copy` not allowed on types with destructors
```

Exercise

The exercise for this section is located in `04_traits/13_drop`

4.14 Wrapping up

We've covered quite a few different traits in this chapter—and we've only scratched the surface! It may feel like you have a lot to remember, but don't worry: you'll bump into these traits so often when writing Rust code that they'll soon become second nature.

Closing thoughts

Traits are powerful, but don't overuse them.
A few guidelines to keep in mind:

- Don't make a function generic if it is always invoked with a single type. It introduces indirection in your codebase, making it harder to understand and maintain.
- Don't create a trait if you only have one implementation. It's a sign that the trait is not needed.
- Implement standard traits for your types (`Debug`, `PartialEq`, etc.) whenever it makes sense. It will make your types more idiomatic and easier to work with, unlocking a lot of functionality provided by the standard library and ecosystem crates.
- Implement traits from third-party crates if you need the functionality they unlock within their ecosystem.
- Beware of making code generic solely to use mocks in your tests. The maintainability cost of this approach can be high, and it's often better to use a different testing strategy. Check out the testing masterclass for details on high-fidelity testing.

Testing your knowledge

Before moving on, let's go through one last exercise to consolidate what we've learned. You'll have minimal guidance this time—just the exercise description and the tests to guide you.

Exercise

The exercise for this section is located in `04_traits/14_outro`

Chapter 5

Modelling A Ticket, pt. 2

The `Ticket` struct we worked on in the previous chapters is a good start, but it still screams "I'm a beginner Rustacean!".

We'll use this chapter to refine our Rust domain modelling skills. We'll need to introduce a few more concepts along the way:

- `enum`s, one of Rust's most powerful features for data modeling
- The `Option` type, to model nullable values
- The `Result` type, to model recoverable errors
- The `Debug` and `Display` traits, for printing
- The `Error` trait, to mark error types
- The `TryFrom` and `TryInto` traits, for fallible conversions
- Rust's package system, explaining what's a library, what's a binary, how to use third-party crates

Exercise

The exercise for this section is located in `05_ticket_v2/00_intro`

5.1 Enumerations

Based on the validation logic you wrote in a previous chapter, there are only a few valid statuses for a ticket: `To-Do`, `InProgress` and `Done`.
This is not obvious if we look at the `status` field in the `Ticket` struct or at the type of the `status` parameter in the `new` method:

```rust
pub struct Ticket {
    title: String,
    description: String,
    status: String,
}

impl Ticket {
    pub fn new(
        title: String,
        description: String,
        status: String
    ) -> Self {
        // [...]
    }
}
```

In both cases we're using `String` to represent the `status` field. `String` is a very general type—it doesn't immediately convey the information that the `status` field has a limited set of possible values. Even worse, the caller of `Ticket::new` will only find out **at runtime** if the status they provided is valid or not.

We can do better than that with **enumerations**.

`enum`

An enumeration is a type that can have a fixed set of values, called **variants**.
In Rust, you define an enumeration using the `enum` keyword:

```rust
enum Status {
    ToDo,
    InProgress,
    Done,
}
```

`enum`, just like `struct`, defines **a new Rust type**.

Exercise

The exercise for this section is located in `05_ticket_v2/01_enum`

5.2 match

You may be wondering—what can you actually **do** with an enum?
The most common operation is to **match** on it.

```
enum Status {
    ToDo,
    InProgress,
    Done
}

impl Status {
    fn is_done(&self) -> bool {
        match self {
            Status::Done => true,
            // The `|` operator lets you match multiple patterns.
            // It reads as "either `Status::ToDo` or `Status::InProgress`".
            Status::InProgress | Status::ToDo => false
        }
    }
}
```

A `match` statement that lets you compare a Rust value against a series of **patterns**. You can think of it as a type-level `if`. If `status` is a `Done` variant, execute the first block; if it's a `InProgress` or `ToDo` variant, execute the second block.

Exhaustiveness

There's one key detail here: `match` is **exhaustive**. You must handle all enum variants. If you forget to handle a variant, Rust will stop you **at compile-time** with an error.

E.g. if we forget to handle the `ToDo` variant:

```
match self {
    Status::Done => true,
    Status::InProgress => false,
}
```

the compiler will complain:

```
error[E0004]: non-exhaustive patterns: `ToDo` not covered
 --> src/main.rs:5:9
  |
5 |     match status {
  |     ^^^^^^^^^^^^ pattern `ToDo` not covered
```

This is a big deal!
Codebases evolve over time—you might add a new status down the line, e.g. `Blocked`. The Rust compiler will emit an error for every single `match` statement that's missing logic for the new variant. That's why Rust developers often sing the praises of "compiler-driven refactoring"—the compiler tells you what to do next, you just have to fix what it reports.

Catch-all

If you don't care about one or more variants, you can use the _ pattern as a catch-all:

```
match status {
    Status::Done => true,
    _ => false
}
```

The _ pattern matches anything that wasn't matched by the previous patterns.

Exercise

The exercise for this section is located in `05_ticket_v2/02_match`

5.3 Variants can hold data

```
enum Status {
    ToDo,
    InProgress,
    Done,
}
```

Our `Status` enum is what's usually called a **C-style enum**.
Each variant is a simple label, a bit like a named constant. You can find this kind of enum in many programming languages, like C, C++, Java, C#, Python, etc.

Rust enums can go further though. We can **attach data to each variant**.

Variants

Let's say that we want to store the name of the person who's currently working on a ticket.
We would only have this information if the ticket is in progress. It wouldn't be there for a to-do ticket or a done ticket. We can model this by attaching a `String` field to the `InProgress` variant:

```
enum Status {
    ToDo,
    InProgress {
        assigned_to: String,
    },
    Done,
}
```

`InProgress` is now a **struct-like variant**.
The syntax mirrors, in fact, the one we used to define a struct—it's just "inlined" inside the enum, as a variant.

Accessing variant data

If we try to access `assigned_to` on a `Status` instance,

```
let status: Status = /* */;

// This won't compile
println!("Assigned to: {}", status.assigned_to);
```

the compiler will stop us:

```
error[E0609]: no field `assigned_to` on type `Status`
 --> src/main.rs:5:40
  |
5 |     println!("Assigned to: {}", status.assigned_to);
  |                                        ^^^^^^^^^^^ unknown field
```

`assigned_to` is **variant-specific**, it's not available on all `Status` instances.
To access `assigned_to`, we need to use **pattern matching**:

```rust
match status {
    Status::InProgress { assigned_to } => {
        println!("Assigned to: {}", assigned_to);
    },
    Status::ToDo | Status::Done => {
        println!("Done");
    }
}
```

Bindings

In the match pattern `Status::InProgress { assigned_to }`, `assigned_to` is a **binding**.
We're **destructuring** the `Status::InProgress` variant and binding the `assigned_to` field to a new variable, also named `assigned_to`.
If we wanted, we could bind the field to a different variable name:

```rust
match status {
    Status::InProgress { assigned_to: person } => {
        println!("Assigned to: {}", person);
    },
    Status::ToDo | Status::Done => {
        println!("Done");
    }
}
```

Exercise

The exercise for this section is located in `05_ticket_v2/03_variants_with_data`

5.4 Concise branching

Your solution to the previous exercise probably looks like this:

```
impl Ticket {
    pub fn assigned_to(&self) -> &str {
        match &self.status {
            Status::InProgress { assigned_to } => assigned_to,
            Status::Done | Status::ToDo => {
                panic!(
                    "Only `In-Progress` tickets can be \
                    assigned to someone"
                )
            }
        }
    }
}
```

You only care about the `Status::InProgress` variant. Do you really need to match on all the other variants?

New constructs to the rescue!

`if let`

The `if let` construct allows you to match on a single variant of an enum, without having to handle all the other variants.

Here's how you can use `if let` to simplify the `assigned_to` method:

```
impl Ticket {
    pub fn assigned_to(&self) -> &str {
        if let Status::InProgress { assigned_to } = &self.status {
            assigned_to
        } else {
            panic!(
                "Only `In-Progress` tickets can be assigned to someone"
            );
        }
    }
}
```

`let/else`

If the `else` branch is meant to return early (a panic counts as returning early!), you can use the `let/else` construct:

```rust
impl Ticket {
    pub fn assigned_to(&self) -> &str {
        let Status::InProgress { assigned_to } = &self.status else {
            panic!(
                "Only `In-Progress` tickets can be assigned to someone"
            );
        };
        assigned_to
    }
}
```

It allows you to assign the destructured variable without incurring any “right drift”, i.e. the variable is assigned at the same indentation level as the code that precedes it.

Style

Both `if let` and `let/else` are idiomatic Rust constructs.
Use them as you see fit to improve the readability of your code, but don’t overdo it: `match` is always there when you need it.

Exercise

The exercise for this section is located in `05_ticket_v2/04_if_let`

5.5 Nullability

Our implementation of the `assigned` method is fairly blunt: panicking for to-do and done tickets is far from ideal.
We can do better using **Rust's `Option` type**.

`Option`

`Option` is a Rust type that represents **nullable values**.
It is an enum, defined in Rust's standard library:

```
enum Option<T> {
    Some(T),
    None,
}
```

`Option` encodes the idea that a value might be present (`Some(T)`) or absent (`None`).
It also forces you to **explicitly handle both cases**. You'll get a compiler error if you are working with a nullable value and you forget to handle the `None` case.
This is a significant improvement over "implicit" nullability in other languages, where you can forget to check for `null` and thus trigger a runtime error.

`Option`'s definition

`Option`'s definition uses a Rust construct that you haven't seen before: **tuple-like variants**.

Tuple-like variants

`Option` has two variants: `Some(T)` and `None`.
`Some` is a **tuple-like variant**: it's a variant that holds **unnamed fields**.

Tuple-like variants are often used when there is a single field to store, especially when we're looking at a "wrapper" type like `Option`.

Tuple-like structs

They're not specific to enums—you can define tuple-like structs too:

```rust
struct Point(i32, i32);
```

You can then access the two fields of a `Point` instance using their positional index:

```rust
let point = Point(3, 4);
let x = point.0;
let y = point.1;
```

Tuples

It's weird to say that something is tuple-like when we haven't seen tuples yet!
Tuples are another example of a primitive Rust type. They group together a fixed number of values with (potentially different) types:

```rust
// Two values, same type
let first: (i32, i32) = (3, 4);
// Three values, different types
let second: (i32, u32, u8) = (-42, 3, 8);
```

The syntax is simple: you list the types of the values between parentheses, separated by commas. You can access the fields of a tuple using the dot notation and the field index:

```rust
assert_eq!(second.0, -42);
assert_eq!(second.1, 3);
assert_eq!(second.2, 8);
```

Tuples are a convenient way of grouping values together when you can't be bothered to define a dedicated struct type.

Exercise

The exercise for this section is located in `05_ticket_v2/05_nullability`

5.6 Fallibility

Let's revisit the `Ticket::new` function from the previous exercise:

```rust
impl Ticket {
    pub fn new(
        title: String,
        description: String,
        status: Status
    ) -> Ticket {
        if title.is_empty() {
            panic!("Title cannot be empty");
        }
        if title.len() > 50 {
            panic!("Title cannot be longer than 50 bytes");
        }
        if description.is_empty() {
            panic!("Description cannot be empty");
        }
        if description.len() > 500 {
            panic!("Description cannot be longer than 500 bytes");
        }

        Ticket {
            title,
            description,
            status,
        }
    }
}
```

As soon as one of the checks fails, the function panics. This is not ideal, as it doesn't give the caller a chance to **handle the error**.

It's time to introduce the `Result` type, Rust's primary mechanism for error handling.

The `Result` type

The `Result` type is an enum defined in the standard library:

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

It has two variants:

- `Ok(T)`: represents a successful operation. It holds `T`, the output of the operation.
- `Err(E)`: represents a failed operation. It holds `E`, the error that occurred.

Both `Ok` and `Err` are generic, allowing you to specify your own types for the success and error cases.

No exceptions

Recoverable errors in Rust are **represented as values**.
They're just an instance of a type, being passed around and manipulated like any other value. This is a significant difference from other languages, such as Python or C#, where **exceptions** are used to signal errors.

Exceptions create a separate control flow path that can be hard to reason about.
You don't know, just by looking at a function's signature, if it can throw an exception or not. You don't know, just by looking at a function's signature, **which** exception types it can throw.
You must either read the function's documentation or look at its implementation to find out.

Exception handling logic has very poor locality: the code that throws the exception is far removed from the code that catches it, and there's no direct link between the two.

Fallibility is encoded in the type system

Rust, with `Result`, forces you to **encode fallibility in the function's signature**.
If a function can fail (and you want the caller to have a shot at handling the error), it must return a `Result`.

```rust
// Just by looking at the signature, you know that this function
// can fail. You can also inspect `ParseIntError` to see what
// kind of failures to expect.
fn parse_int(s: &str) -> Result<i32, ParseIntError> {
    // ...
}
```

That's the big advantage of `Result`: it makes fallibility explicit.

Keep in mind, though, that panics exist. They aren't tracked by the type system, just like exceptions in other languages. But they're meant for **unrecoverable errors** and should be used sparingly.

Exercise

The exercise for this section is located in `05_ticket_v2/06_fallibility`

5.7 Unwrapping

`Ticket::new` now returns a `Result` instead of panicking on invalid inputs.
What does this mean for the caller?

Failures can't be (implicitly) ignored

Unlike exceptions, Rust's `Result` forces you to **handle errors at the call site**.
If you call a function that returns a `Result`, Rust won't allow you to implicitly ignore the error case.

```
fn parse_int(s: &str) -> Result<i32, ParseIntError> {
    // ...
}

// This won't compile: we're not handling the error case.
// We must either use `match` or one of the combinators provided by
// `Result` to "unwrap" the success value or handle the error.
let number = parse_int("42") + 2;
```

You got a `Result`. Now what?

When you call a function that returns a `Result`, you have two key options:

- Panic if the operation failed. This is done using either the `unwrap` or `expect` methods.

  ```
  // Panics if `parse_int` returns an `Err`.
  let number = parse_int("42").unwrap();
  // `expect` lets you specify a custom panic message.
  let number = parse_int("42").expect("Failed to parse integer");
  ```

- Destructure the `Result` using a `match` expression to deal with the error case explicitly.

  ```
  match parse_int("42") {
      Ok(number) => println!("Parsed number: {}", number),
      Err(err) => eprintln!("Error: {}", err),
  }
  ```

Exercise

The exercise for this section is located in `05_ticket_v2/07_unwrap`

5.8 Error enums

Your solution to the previous exercise may have felt awkward: matching on strings is not ideal!
A colleague might rework the error messages returned by `Ticket::new` (e.g. to improve readability) and, all of a sudden, your calling code would break.

You already know the machinery required to fix this: enums!

Reacting to errors

When you want to allow the caller to behave differently based on the specific error that occurred, you can use an enum to represent the different error cases:

```
// An error enum to represent the different error cases
// that may occur when parsing a `u32` from a string.
enum U32ParseError {
    NotANumber,
    TooLarge,
    Negative,
}
```

Using an error enum, you're encoding the different error cases in the type system—they become part of the signature of the fallible function.
This simplifies error handling for the caller, as they can use a `match` expression to react to the different error cases:

```
match s.parse_u32() {
    Ok(n) => n,
    Err(U32ParseError::Negative) => 0,
    Err(U32ParseError::TooLarge) => u32::MAX,
    Err(U32ParseError::NotANumber) => {
        panic!("Not a number: {}", s);
    }
}
```

Exercise

The exercise for this section is located in `05_ticket_v2/08_error_enums`

5.9 Error trait

Error reporting

In the previous exercise you had to destructure the `TitleError` variant to extract the error message and pass it to the `panic!` macro.
This is a (rudimentary) example of **error reporting**: transforming an error type into a representation that can be shown to a user, a service operator, or a developer.

It's not practical for each Rust developer to come up with their own error reporting strategy: it'd be a waste of time and it wouldn't compose well across projects. That's why Rust provides the `std::error::Error` trait.

The Error trait

There are no constraints on the type of the `Err` variant in a `Result`, but it's a good practice to use a type that implements the `Error` trait. `Error` is the cornerstone of Rust's error handling story:

```
// Slightly simplified definition of the `Error` trait
pub trait Error: Debug + Display {}
```

You might recall the `:` syntax from the `From` trait—it's used to specify **supertraits**. For `Error`, there are two supertraits: `Debug` and `Display`. If a type wants to implement `Error`, it must also implement `Debug` and `Display`.

`Display` and `Debug`

We've already encountered the `Debug` trait in a previous exercise—it's the trait used by `assert_eq!` to display the values of the variables it's comparing when the assertion fails.

From a "mechanical" perspective, `Display` and `Debug` are identical—they encode how a type should be converted into a string-like representation:

```
// `Debug`
pub trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}

// `Display`
pub trait Display {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

The difference is in their *purpose*: `Display` returns a representation that's meant for "end-users", while `Debug` provides a low-level representation that's more suitable to developers and service operators.
That's why `Debug` can be automatically implemented using the `#[derive(Debug)]` attribute, while `Display` **requires** a manual implementation.

Exercise

The exercise for this section is located in `05_ticket_v2/09_error_trait`

5.10 Libraries and binaries

It took a bit of code to implement the `Error` trait for `TicketNewError`, didn't it?
A manual `Display` implementation, plus an `Error` impl block.

We can remove some of the boilerplate by using `thiserror`, a Rust crate that provides a **procedural macro** to simplify the creation of custom error types.
But we're getting ahead of ourselves: `thiserror` is a third-party crate, it'd be our first dependency!

Let's take a step back to talk about Rust's packaging system before we dive into dependencies.

What is a package?

A Rust package is defined by the `[package]` section in a `Cargo.toml` file, also known as its **manifest**. Within `[package]` you can set the package's metadata, such as its name and version.

Go check the `Cargo.toml` file in the directory of this section's exercise!

What is a crate?

Inside a package, you can have one or more **crates**, also known as **targets**.
The two most common crate types are **binary crates** and **library crates**.

Binaries

A binary is a program that can be compiled to an **executable file**.
It must include a function named `main`—the program's entry point. `main` is invoked when the program is executed.

Libraries

Libraries, on the other hand, are not executable on their own. You can't *run* a library, but you can *import its code* from another package that depends on it.
A library groups together code (i.e. functions, types, etc.) that can be leveraged by other packages as a **dependency**.

All the exercises you've solved so far have been structured as libraries, with a test suite attached to them.

Conventions

There are some conventions around Rust packages that you need to keep in mind:

- The package's source code is usually located in the `src` directory.
- If there's a `src/lib.rs` file, `cargo` will infer that the package contains a library crate.
- If there's a `src/main.rs` file, `cargo` will infer that the package contains a binary crate.

You can override these defaults by explicitly declaring your targets in the `Cargo.toml` file—see cargo's documentation for more details.

Keep in mind that while a package can contain multiple crates, it can only contain one library crate.

Exercise

The exercise for this section is located in `05_ticket_v2/10_packages`

5.11 Dependencies

A package can depend on other packages by listing them in the `[dependencies]` section of its `Cargo.toml` file.
The most common way to specify a dependency is by providing its name and version:

```toml
[dependencies]
thiserror = "1"
```

This will add `thiserror` as a dependency to your package, with a **minimum** version of `1.0.0`. `thiserror` will be pulled from crates.io, Rust's official package registry. When you run `cargo build`, `cargo` will go through a few stages:

- Dependency resolution
- Downloading the dependencies
- Compiling your project (your own code and the dependencies)

Dependency resolution is skipped if your project has a `Cargo.lock` file and your manifest files are unchanged. A lockfile is automatically generated by `cargo` after a successful round of dependency resolution: it contains the exact versions of all dependencies used in your project, and is used to ensure that the same versions are consistently used across different builds (e.g. in CI). If you're working on a project with multiple developers, you should commit the `Cargo.lock` file to your version control system.

You can use `cargo update` to update the `Cargo.lock` file with the latest (compatible) versions of all your dependencies.

Path dependencies

You can also specify a dependency using a **path**. This is useful when you're working on multiple local packages.

```toml
[dependencies]
my-library = { path = "../my-library" }
```

The path is relative to the `Cargo.toml` file of the package that's declaring the dependency.

Other sources

Check out the Cargo documentation for more details on where you can get dependencies from and how to specify them in your `Cargo.toml` file.

Dev dependencies

You can also specify dependencies that are only needed for development—i.e. they only get pulled in when you're running `cargo test`.
They go in the `[dev-dependencies]` section of your `Cargo.toml` file:

```toml
[dev-dependencies]
static_assertions = "1.1.0"
```

We've been using a few of these throughout the book to shorten our tests.

Exercise

The exercise for this section is located in `05_ticket_v2/11_dependencies`

5.12 thiserror

That was a bit of detour, wasn't it? But a necessary one!
Let's get back on track now: custom error types and `thiserror`.

Custom error types

We've seen how to implement the `Error` trait "manually" for a custom error type. Imagine that you have to do this for most error types in your codebase. That's a lot of boilerplate, isn't it?

We can remove some of the boilerplate by using `thiserror`, a Rust crate that provides a **procedural macro** to simplify the creation of custom error types.

```
enum TicketNewError {
    TitleError(String),
    DescriptionError(String),
}
```

You can write your own macros

All the `derive` macros we've seen so far were provided by the Rust standard library. `thiserror::Error` is the first example of a **third-party** `derive` macro.

`derive` macros are a subset of **procedural macros**, a way to generate Rust code at compile time. We won't get into the details of how to write a procedural macro in this course, but it's important to know that you can write your own!
A topic to approach in a more advanced Rust course.

Custom syntax

Each procedural macro can define its own syntax, which is usually explained in the crate's documentation. In the case of `thiserror`, we have:

- `#[derive(thiserror::Error)]`: this is the syntax to derive the `Error` trait for a custom error type, helped by `thiserror`.
- `#[error("{0}")]`: this is the syntax to define a `Display` implementation for each variant of the custom error type. `{0}` is replaced by the zero-th field of the variant (`String`, in this case) when the error is displayed.

Exercise

The exercise for this section is located in `05_ticket_v2/12_thiserror`

5.13 TryFrom and TryInto

In the previous chapter we looked at the From and Into traits, Rust's idiomatic interfaces for **infallible** type conversions.
But what if the conversion is not guaranteed to succeed?

We now know enough about errors to discuss the **fallible** counterparts of From and Into: TryFrom and TryInto.

TryFrom and TryInto

Both TryFrom and TryInto are defined in the std::convert module, just like From and Into.

```
pub trait TryFrom<T>: Sized {
    type Error;
    fn try_from(value: T) -> Result<Self, Self::Error>;
}

pub trait TryInto<T>: Sized {
    type Error;
    fn try_into(self) -> Result<T, Self::Error>;
}
```

The main difference between From/Into and TryFrom/TryInto is that the latter return a Result type.
This allows the conversion to fail, returning an error instead of panicking.

Self::Error

Both TryFrom and TryInto have an associated Error type. This allows each implementation to specify its own error type, ideally the most appropriate for the conversion being attempted.

Self::Error is a way to refer to the Error associated type defined in the trait itself.

Duality

Just like From and Into, TryFrom and TryInto are dual traits.
If you implement TryFrom for a type, you get TryInto for free.

Exercise

The exercise for this section is located in 05_ticket_v2/13_try_from

5.14 Error::source

There's one more thing we need to talk about to complete our coverage of the `Error` trait: the `source` method.

```rust
// Full definition this time!
pub trait Error: Debug + Display {
    fn source(&self) -> Option<&(dyn Error + 'static)> {
        None
    }
}
```

The `source` method is a way to access the **error cause**, if any.
Errors are often chained, meaning that one error is the cause of another: you have a high-level error (e.g. cannot connect to the database) that is caused by a lower-level error (e.g. can't resolve the database hostname). The `source` method allows you to "walk" the full chain of errors, often used when capturing error context in logs.

Implementing source

The `Error` trait provides a default implementation that always returns None (i.e. no underlying cause). That's why you didn't have to care about `source` in the previous exercises.
You can override this default implementation to provide a cause for your error type.

```rust
use std::error::Error;

struct DatabaseError {
    source: std::io::Error
}

impl std::fmt::Display for DatabaseError {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Failed to connect to the database")
    }
}

impl std::error::Error for DatabaseError {
    fn source(&self) -> Option<&(dyn Error + 'static)> {
        Some(&self.source)
    }
}
```

In this example, `DatabaseError` wraps an `std::io::Error` as its source. We then override the `source` method to return this source when called.

`&(dyn Error + 'static)`

What's this `&(dyn Error + 'static)` type?
Let's unpack it:

- `dyn Error` is a **trait object**. It's a way to refer to any type that implements the `Error` trait.
- `'static` is a special **lifetime specifier**. `'static` implies that the reference is valid for "as long as we need it", i.e. the entire program execution.

Combined: `&(dyn Error + 'static)` is a reference to a trait object that implements the `Error` trait and is valid for the entire program execution.

Don't worry too much about either of these concepts for now. We'll cover them in more detail in future chapters.

Implementing `source` using `thiserror`

`thiserror` provides three ways to automatically implement `source` for your error types:

- A field named `source` will automatically be used as the source of the error.

```rust
use thiserror::Error;

pub enum MyError {
    DatabaseError {
        source: std::io::Error
    }
}
```

- A field annotated with the `#[source]` attribute will automatically be used as the source of the error.

```rust
use thiserror::Error;

pub enum MyError {
    DatabaseError {
        inner: std::io::Error
    }
}
```

- A field annotated with the `#[from]` attribute will automatically be used as the source of the error **and** `thiserror` will automatically generate a `From` implementation to convert the annotated type into your error type.

```rust
use thiserror::Error;

pub enum MyError {
    DatabaseError {
        inner: std::io::Error
    }
}
```

The ? operator

The ? operator is a shorthand for propagating errors.
When used in a function that returns a `Result`, it will return early with an error if the `Result` is `Err`.

For example:

```rust
use std::fs::File;

fn read_file() -> Result<String, std::io::Error> {
    let mut file = File::open("file.txt")?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}
```

is equivalent to:

```rust
use std::fs::File;

fn read_file() -> Result<String, std::io::Error> {
    let mut file = match File::open("file.txt") {
        Ok(file) => file,
        Err(e) => {
            return Err(e);
        }
    };
    let mut contents = String::new();
    match file.read_to_string(&mut contents) {
        Ok(_) => (),
        Err(e) => {
            return Err(e);
        }
    }
    Ok(contents)
}
```

You can use the ? operator to shorten your error handling code significantly.
In particular, the ? operator will automatically convert the error type of the fallible

operation into the error type of the function, if a conversion is possible (i.e. if there is a suitable `From` implementation)

Exercise

The exercise for this section is located in `05_ticket_v2/14_source`

5.15 Wrapping up

When it comes to domain modelling, the devil is in the details.
Rust offers a wide range of tools to help you represent the constraints of your domain directly in the type system, but it takes some practice to get it right and write code that looks idiomatic.

Let's close the chapter with one final refinement of our `Ticket` model.
We'll introduce a new type for each of the fields in `Ticket` to encapsulate the respective constraints.
Every time someone accesses a `Ticket` field, they'll get back a value that's guaranteed to be valid—i.e. a `TicketTitle` instead of a `String`. They won't have to worry about the title being empty elsewhere in the code: as long as they have a `TicketTitle`, they know it's valid **by construction**.

This is just an example of how you can use Rust's type system to make your code safer and more expressive.

Further reading

- Parse, don't validate
- Using types to guarantee domain invariants

Exercise

The exercise for this section is located in `05_ticket_v2/15_outro`

Chapter 6

Intro

In the previous chapter we modelled `Ticket` in a vacuum: we defined its fields and their constraints, we learned how to best represent them in Rust, but we didn't consider how `Ticket` fits into a larger system. We'll use this chapter to build a simple workflow around `Ticket`, introducing a (rudimentary) management system to store and retrieve tickets.

The task will give us an opportunity to explore new Rust concepts, such as:

- Stack-allocated arrays
- `Vec`, a growable array type, and slices
- `Iterator` and `IntoIterator`, for iterating over collections
- Slices (`&[T]`), to work with parts of a collection
- Lifetimes, to describe how long references are valid
- `HashMap` and `BTreeMap`, two key-value data structures
- `Eq` and `Hash`, to compare keys in a `HashMap`
- `Ord` and `PartialOrd`, to work with a `BTreeMap`
- `Index` and `IndexMut`, to access elements in a collection

Exercise

The exercise for this section is located in `06_ticket_management/00_intro`

6.1 Arrays

As soon as we start talking about "ticket management" we need to think about a way to store *multiple* tickets. In turn, this means we need to think about collections. In particular, homogeneous collections: we want to store multiple instances of the same type.

What does Rust have to offer in this regard?

Arrays

A first attempt could be to use an **array**.
Arrays in Rust are fixed-size collections of elements of the same type.

Here's how you can define an array:

```
// Array type syntax: [ <type> ; <number of elements> ]
let numbers: [u32; 3] = [1, 2, 3];
```

This creates an array of 3 integers, initialized with the values 1, 2, and 3.
The type of the array is `[u32; 3]`, which reads as "an array of u32s with a length of 3".

Accessing elements

You can access elements of an array using square brackets:

```
let first = numbers[0];
let second = numbers[1];
let third = numbers[2];
```

The index must be of type `usize`.
Arrays are **zero-indexed**, like everything in Rust. You've seen this before with string slices and field indexing in tuples/tuple-like variants.

Out-of-bounds access

If you try to access an element that's out of bounds, Rust will panic:

```
let numbers: [u32; 3] = [1, 2, 3];
let fourth = numbers[3]; // This will panic
```

This is enforced at runtime using **bounds checking**. It comes with a small performance overhead, but it's how Rust prevents buffer overflows.
In some scenarios the Rust compiler can optimize away bounds checks, especially if iterators are involved—we'll speak more about this later on.

If you don't want to panic, you can use the `get` method, which returns an `Option<&T>`:

```
let numbers: [u32; 3] = [1, 2, 3];
assert_eq!(numbers.get(0), Some(&1));
// You get a `None` if you try to access an out-of-bounds index
// rather than a panic.
assert_eq!(numbers.get(3), None);
```

Performance

Since the size of an array is known at compile-time, the compiler can allocate the array on the stack. If you run the following code:

```
let numbers: [u32; 3] = [1, 2, 3];
```

You'll get the following memory layout:

```
        +---+---+---+
Stack:  | 1 | 2 | 3 |
        +---+---+---+
```

In other words, the size of an array is `std::mem::size_of::<T>() * N`, where `T` is the type of the elements and `N` is the number of elements.
You can access and replace each element in `O(1)` time.

Exercise

The exercise for this section is located in `06_ticket_management/01_arrays`

6.2 Vectors

Arrays' strength is also their weakness: their size must be known upfront, at compile-time. If you try to create an array with a size that's only known at runtime, you'll get a compilation error:

```
let n = 10;
let numbers: [u32; n];
```

```
error[E0435]: attempt to use a non-constant value in a constant
 --> src/main.rs:3:20
  |
2 | let n = 10;
3 | let numbers: [u32; n];
  |                    ^ non-constant value
```

Arrays wouldn't work for our ticket management system—we don't know how many tickets we'll need to store at compile-time. This is where `Vec` comes in.

Vec

`Vec` is a growable array type, provided by the standard library.
You can create an empty array using the `Vec::new` function:

```
let mut numbers: Vec<u32> = Vec::new();
```

You would then push elements into the vector using the `push` method:

```
numbers.push(1);
numbers.push(2);
numbers.push(3);
```

New values are added to the end of the vector.
You can also create an initialized vector using the `vec!` macro, if you know the values at creation time:

```
let numbers = vec![1, 2, 3];
```

Accessing elements

The syntax for accessing elements is the same as with arrays:

```rust
let numbers = vec![1, 2, 3];
let first = numbers[0];
let second = numbers[1];
let third = numbers[2];
```

The index must be of type `usize`.
You can also use the `get` method, which returns an `Option<&T>`:

```rust
let numbers = vec![1, 2, 3];
assert_eq!(numbers.get(0), Some(&1));
// You get a `None` if you try to access an out-of-bounds index
// rather than a panic.
assert_eq!(numbers.get(3), None);
```

Access is bounds-checked, just like element access with arrays. It has O(1) complexity.

Memory layout

`Vec` is a heap-allocated data structure.
When you create a `Vec`, it allocates memory on the heap to store the elements.

If you run the following code:

```rust
let mut numbers = Vec::with_capacity(3);
numbers.push(1);
numbers.push(2);
```

you'll get the following memory layout:

```
      +---------+--------+----------+
Stack | pointer | length | capacity |
      |  |      |   2    |    3     |
      +--|------+--------+----------+
         |
         |
         v
       +---+---+---+
Heap:  | 1 | 2 | ? |
       +---+---+---+
```

`Vec` keeps track of three things:

- The **pointer** to the heap region you reserved.
- The **length** of the vector, i.e. how many elements are in the vector.

- The **capacity** of the vector, i.e. the number of elements that can fit in the space reserved on the heap.

This layout should look familiar: it's exactly the same as `String`!
That's not a coincidence: `String` is defined as a vector of bytes, `Vec<u8>`, under the hood:

```rust
pub struct String {
    vec: Vec<u8>,
}
```

Exercise

The exercise for this section is located in `06_ticket_management/02_vec`

6.3 Resizing

We said that `Vec` is a "growable" vector type, but what does that mean? What happens if you try to insert an element into a `Vec` that's already at maximum capacity?

```
let mut numbers = Vec::with_capacity(3);
numbers.push(1);
numbers.push(2);
numbers.push(3); // Max capacity reached
numbers.push(4); // What happens here?
```

The `Vec` will **resize** itself.
It will ask the allocator for a new (larger) chunk of heap memory, copy the elements over, and deallocate the old memory.

This operation can be expensive, as it involves a new memory allocation and copying all existing elements.

`Vec::with_capacity`

If you have a rough idea of how many elements you'll store in a `Vec`, you can use the `Vec::with_capacity` method to pre-allocate enough memory upfront.
This can avoid a new allocation when the `Vec` grows, but it may waste memory if you overestimate actual usage.

Evaluate on a case-by-case basis.

Exercise

The exercise for this section is located in `06_ticket_management/03_resizing`

6.4 Iteration

During the very first exercises, you learned that Rust lets you iterate over collections using `for` loops. We were looking at ranges at that point (e.g. `0..5`), but the same holds true for collections like arrays and vectors.

```
// It works for `Vec`s
let v = vec![1, 2, 3];
for n in v {
    println!("{}", n);
}

// It also works for arrays
let a: [u32; 3] = [1, 2, 3];
for n in a {
    println!("{}", n);
}
```

It's time to understand how this works under the hood.

`for` desugaring

Every time you write a `for` loop in Rust, the compiler *desugars* it into the following code:

```
let mut iter = IntoIterator::into_iter(v);
loop {
    match iter.next() {
        Some(n) => {
            println!("{}", n);
        }
        None => break,
    }
}
```

`loop` is another looping construct, on top of `for` and `while`.
A `loop` block will run forever, unless you explicitly `break` out of it.

`Iterator` trait

The `next` method in the previous code snippet comes from the `Iterator` trait. The `Iterator` trait is defined in Rust's standard library and provides a shared interface for types that can produce a sequence of values:

```rust
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

The `Item` associated type specifies the type of the values produced by the iterator.

`next` returns the next value in the sequence.
It returns `Some(value)` if there's a value to return, and `None` when there isn't.

Be careful: there is no guarantee that an iterator is exhausted when it returns `None`. That's only guaranteed if the iterator implements the (more restrictive) `FusedIterator` trait.

`IntoIterator` trait

Not all types implement `Iterator`, but many can be converted into a type that does. That's where the `IntoIterator` trait comes in:

```rust
trait IntoIterator {
    type Item;
    type IntoIter: Iterator<Item = Self::Item>;
    fn into_iter(self) -> Self::IntoIter;
}
```

The `into_iter` method consumes the original value and returns an iterator over its elements.
A type can only have one implementation of `IntoIterator`: there can be no ambiguity as to what `for` should desugar to.

One detail: every type that implements `Iterator` automatically implements `IntoIterator` as well. They just return themselves from `into_iter`!

Bounds checks

Iterating over iterators has a nice side effect: you can't go out of bounds, by design. This allows Rust to remove bounds checks from the generated machine code, making iteration faster.

In other words,

```rust
let v = vec![1, 2, 3];
for n in v {
    println!("{}", n);
}
```

is usually faster than

```
let v = vec![1, 2, 3];
for i in 0..v.len() {
    println!("{}", v[i]);
}
```

There are exceptions to this rule: the compiler can sometimes prove that you're not going out of bounds even with manual indexing, thus removing the bounds checks anyway. But in general, prefer iteration to indexing where possible.

Exercise

The exercise for this section is located in `06_ticket_management/04_iterators`

6.5 `.iter()`

`IntoIterator` **consumes** `self` to create an iterator.

This has its benefits: you get **owned** values from the iterator. For example: if you call `.into_iter()` on a `Vec<Ticket>` you'll get an iterator that returns `Ticket` values.

That's also its downside: you can no longer use the original collection after calling `.into_iter()` on it. Quite often you want to iterate over a collection without consuming it, looking at **references** to the values instead. In the case of `Vec<Ticket>`, you'd want to iterate over `&Ticket` values.

Most collections expose a method called `.iter()` that returns an iterator over references to the collection's elements. For example:

```
let numbers: Vec<u32> = vec![1, 2];
// `n` has type `&u32` here
for n in numbers.iter() {
    // [...]
}
```

This pattern can be simplified by implementing `IntoIterator` for a **reference to the collection**. In our example above, that would be `&Vec<Ticket>`.
The standard library does this, that's why the following code works:

```
let numbers: Vec<u32> = vec![1, 2];
// `n` has type `&u32` here
// We didn't have to call `.iter()` explicitly
// It was enough to use `&numbers` in the `for` loop
for n in &numbers {
    // [...]
}
```

It's idiomatic to provide both options:

- An implementation of `IntoIterator` for a reference to the collection.
- An `.iter()` method that returns an iterator over references to the collection's elements.

The former is convenient in `for` loops, the latter is more explicit and can be used in other contexts.

Exercise

The exercise for this section is located in `06_ticket_management/05_iter`

6.6 Lifetimes

Let's try to complete the previous exercise by adding an implementation of `IntoIterator` for `&TicketStore`, for maximum convenience in `for` loops.

Let's start by filling in the most "obvious" parts of the implementation:

```
impl IntoIterator for &TicketStore {
    type Item = &Ticket;
    type IntoIter = // What goes here?

    fn into_iter(self) -> Self::IntoIter {
        self.tickets.iter()
    }
}
```

What should `type IntoIter` be set to?
Intuitively, it should be the type returned by `self.tickets.iter()`, i.e. the type returned by `Vec::iter()`.
If you check the standard library documentation, you'll find that `Vec::iter()` returns an `std::slice::Iter`. The definition of `Iter` is:

```
pub struct Iter<'a, T> { /* fields omitted */ }
```

`'a` is a **lifetime parameter**.

Lifetime parameters

Lifetimes are **labels** used by the Rust compiler to keep track of how long a reference (either mutable or immutable) is valid.
The lifetime of a reference is constrained by the scope of the value it refers to. Rust always makes sure, at compile-time, that references are not used after the value they refer to has been dropped, to avoid dangling pointers and use-after-free bugs.

This should sound familiar: we've already seen these concepts in action when we discussed ownership and borrowing. Lifetimes are just a way to **name** how long a specific reference is valid.

Naming becomes important when you have multiple references and you need to clarify how they **relate to each other**. Let's look at the signature of `Vec::iter()`:

```
impl <T> Vec<T> {
    // Slightly simplified
    pub fn iter<'a>(&'a self) -> Iter<'a, T> {
        // [...]
    }
}
```

`Vec::iter()` is generic over a lifetime parameter, named `'a`.
`'a` is used to **tie together** the lifetime of the `Vec` and the lifetime of the `Iter` returned by `iter()`. In plain English: the `Iter` returned by `iter()` cannot outlive the `Vec` reference (`&self`) it was created from.

This is important because `Vec::iter`, as we discussed, returns an iterator over **references** to the `Vec`'s elements. If the `Vec` is dropped, the references returned by the iterator would be invalid. Rust must make sure this doesn't happen, and lifetimes are the tool it uses to enforce this rule.

Lifetime elision

Rust has a set of rules, called **lifetime elision rules**, that allow you to omit explicit lifetime annotations in many cases. For example, `Vec::iter`'s definition looks like this in `std`'s source code:

```
impl <T> Vec<T> {
    pub fn iter(&self) -> Iter<'_, T> {
        // [...]
    }
}
```

No explicit lifetime parameter is present in the signature of `Vec::iter()`. Elision rules imply that the lifetime of the `Iter` returned by `iter()` is tied to the lifetime of the `&self` reference. You can think of `'_` as a **placeholder** for the lifetime of the `&self` reference.

See the References section for a link to the official documentation on lifetime elision. In most cases, you can rely on the compiler telling you when you need to add explicit lifetime annotations.

References

- std::vec::Vec::iter
- std::slice::Iter
- Lifetime elision rules

Exercise

The exercise for this section is located in `06_ticket_management/06_lifetimes`

6.7 Combinators

Iterators can do so much more than `for` loops!
If you look at the documentation for the `Iterator` trait, you'll find a **vast** collections of methods that you can leverage to transform, filter, and combine iterators in various ways.

Let's mention the most common ones:

- `map` applies a function to each element of the iterator.
- `filter` keeps only the elements that satisfy a predicate.
- `filter_map` combines `filter` and `map` in one step.
- `cloned` converts an iterator of references into an iterator of values, cloning each element.
- `enumerate` returns a new iterator that yields `(index, value)` pairs.
- `skip` skips the first `n` elements of the iterator.
- `take` stops the iterator after `n` elements.
- `chain` combines two iterators into one.

These methods are called **combinators**.
They are usually **chained** together to create complex transformations in a concise and readable way:

```rust
let numbers = vec![1, 2, 3, 4, 5];
// The sum of the squares of the even numbers
let outcome: u32 = numbers.iter()
    .filter(|&n| n % 2 == 0)
    .map(|&n| n * n)
    .sum();
```

Closures

What's going on with the `filter` and `map` methods above?
They take **closures** as arguments.

Closures are **anonymous functions**, i.e. functions that are not defined using the `fn` syntax we are used to.
They are defined using the `|args| body` syntax, where `args` are the arguments and `body` is the function body. `body` can be a block of code or a single expression. For example:

```rust
// An anonymous function that adds 1 to its argument
let add_one = |x| x + 1;
// Could be written with a block too:
let add_one = |x| { x + 1 };
```

Closures can take more than one argument:

```
let add = |x, y| x + y;
let sum = add(1, 2);
```

They can also capture variables from their environment:

```
let x = 42;
let add_x = |y| x + y;
let sum = add_x(1);
```

If necessary, you can specify the types of the arguments and/or the return type:

```
// Just the input type
let add_one = |x: i32| x + 1;
// Or both input and output types, using the `fn` syntax
let add_one: fn(i32) -> i32 = |x| x + 1;
```

`collect`

What happens when you're done transforming an iterator using combinators?
You either iterate over the transformed values using a `for` loop, or you collect them into a collection.

The latter is done using the `collect` method.
`collect` consumes the iterator and collects its elements into a collection of your choice.

For example, you can collect the squares of the even numbers into a `Vec`:

```
let numbers = vec![1, 2, 3, 4, 5];
let squares_of_evens: Vec<u32> = numbers.iter()
    .filter(|&n| n % 2 == 0)
    .map(|&n| n * n)
    .collect();
```

`collect` is generic over its **return type**.
Therefore you usually need to provide a type hint to help the compiler infer the correct type. In the example above, we annotated the type of `squares_of_evens` to be `Vec<u32>`. Alternatively, you can use the **turbofish syntax** to specify the type:

```
let squares_of_evens = numbers.iter()
    .filter(|&n| n % 2 == 0)
    .map(|&n| n * n)
    // Turbofish syntax: `<method_name>::<type>()`
    // It's called turbofish because `::<>` looks like a fish
    .collect::<Vec<u32>>();
```

Further reading

- `Iterator`'s documentation gives you an overview of the methods available for iterators in `std`.
- The `itertools` crate defines even **more** combinators for iterators.

Exercise

The exercise for this section is located in `06_ticket_management/07_combinators`

6.8 `impl Trait`

`TicketStore::to_dos` returns a `Vec<&Ticket>`.
That signature introduces a new heap allocation every time `to_dos` is called, which may be unnecessary depending on what the caller needs to do with the result. It'd be better if `to_dos` returned an iterator instead of a `Vec`, thus empowering the caller to decide whether to collect the results into a `Vec` or just iterate over them.

That's tricky though! What's the return type of `to_dos`, as implemented below?

```
impl TicketStore {
    pub fn to_dos(&self) -> ??? {
        self.tickets.iter().filter(|t| t.status == Status::ToDo)
    }
}
```

Unnameable types

The `filter` method returns an instance of `std::iter::Filter`, which has the following definition:

```
pub struct Filter<I, P> { /* fields omitted */ }
```

where `I` is the type of the iterator being filtered on and `P` is the predicate used to filter the elements.
We know that `I` is `std::slice::Iter<'_, Ticket>` in this case, but what about `P`?
`P` is a closure, an **anonymous function**. As the name suggests, closures don't have a name, so we can't write them down in our code.

Rust has a solution for this: **impl Trait**.

`impl Trait`

`impl Trait` is a feature that allows you to return a type without specifying its name. You just declare what trait(s) the type implements, and Rust figures out the rest.

In this case, we want to return an iterator of references to `Ticket`s:

```
impl TicketStore {
    pub fn to_dos(&self) -> impl Iterator<Item = &Ticket> {
        self.tickets.iter().filter(|t| t.status == Status::ToDo)
    }
}
```

That's it!

Generic?

`impl Trait` in return position is **not** a generic parameter.

Generics are placeholders for types that are filled in by the caller of the function. A function with a generic parameter is **polymorphic**: it can be called with different types, and the compiler will generate a different implementation for each type.

That's not the case with `impl Trait`. The return type of a function with `impl Trait` is **fixed** at compile time, and the compiler will generate a single implementation for it. This is why `impl Trait` is also called **opaque return type**: the caller doesn't know the exact type of the return value, only that it implements the specified trait(s). But the compiler knows the exact type, there is no polymorphism involved.

RPIT

If you read RFCs or deep-dives about Rust, you might come across the acronym **RPIT**. It stands for **"Return Position Impl Trait"** and refers to the use of `impl Trait` in return position.

Exercise

The exercise for this section is located in `06_ticket_management/08_impl_trait`

6.9 impl Trait in argument position

In the previous section, we saw how `impl Trait` can be used to return a type without specifying its name.
The same syntax can also be used in **argument position**:

```
fn print_iter(iter: impl Iterator<Item = i32>) {
    for i in iter {
        println!("{}", i);
    }
}
```

`print_iter` takes an iterator of `i32`s and prints each element.
When used in **argument position**, `impl Trait` is equivalent to a generic parameter with a trait bound:

```
fn print_iter<T>(iter: T)
where
    T: Iterator<Item = i32>
{
    for i in iter {
        println!("{}", i);
    }
}
```

Downsides

As a rule of thumb, prefer generics over `impl Trait` in argument position.
Generics allow the caller to explicitly specify the type of the argument, using the turbofish syntax (`::<>`), which can be useful for disambiguation. That's not the case with `impl Trait`.

Exercise

The exercise for this section is located in `06_ticket_management/09_impl_trait_2`

6.10 Slices

Let's go back to the memory layout of a `Vec`:

```
let mut numbers = Vec::with_capacity(3);
numbers.push(1);
numbers.push(2);
```

```
      +---------+--------+----------+
Stack | pointer | length | capacity |
      |  |      |   2    |    3     |
      +--|------+--------+----------+
         |
         |
         v
       +---+---+---+
Heap:  | 1 | 2 | ? |
       +---+---+---+
```

We already remarked how `String` is just a `Vec<u8>` in disguise.
The similarity should prompt you to ask: "What's the equivalent of `&str` for `Vec`?"

`&[T]`

`[T]` is a **slice** of a contiguous sequence of elements of type `T`.
It's most commonly used in its borrowed form, `&[T]`.

There are various ways to create a slice reference from a `Vec`:

```
let numbers = vec![1, 2, 3];
// Via index syntax
let slice: &[i32] = &numbers[..];
// Via a method
let slice: &[i32] = numbers.as_slice();
// Or for a subset of the elements
let slice: &[i32] = &numbers[1..];
```

`Vec` implements the `Deref` trait using `[T]` as the target type, so you can use slice methods on a `Vec` directly thanks to deref coercion:

```
let numbers = vec![1, 2, 3];
// Surprise, surprise: `iter` is not a method on `Vec`!
// It's a method on `&[T]`, but you can call it on a `Vec`
// thanks to deref coercion.
let sum: i32 = numbers.iter().sum();
```

Memory layout

A `&[T]` is a **fat pointer**, just like `&str`.
It consists of a pointer to the first element of the slice and the length of the slice.

If you have a `Vec` with three elements:

```rust
let numbers = vec![1, 2, 3];
```

and then create a slice reference:

```rust
let slice: &[i32] = &numbers[1..];
```

you'll get this memory layout:

```
                  numbers                          slice
      +---------+--------+----------+      +---------+--------+
Stack | pointer | length | capacity |      | pointer | length |
      |    |    |   3    |    4     |      |    |    |   2    |
      +----|----+--------+----------+      +----|----+--------+
           |                                    |
           |                                    |
           v                                    |
         +---+---+---+---+                      |
Heap:    | 1 | 2 | 3 | ? |                      |
         +---+---+---+---+                      |
               ^                                |
               |                                |
               +--------------------------------+
```

`&Vec<T>` vs `&[T]`

When you need to pass an immutable reference to a `Vec` to a function, prefer `&[T]` over `&Vec<T>`.
This allows the function to accept any kind of slice, not necessarily one backed by a `Vec`.

For example, you can then pass a subset of the elements in a `Vec`. But it goes further than that—you could also pass a **slice of an array**:

```rust
let array = [1, 2, 3];
let slice: &[i32] = &array;
```

Array slices and `Vec` slices are the same type: they're fat pointers to a contiguous sequence of elements. In the case of arrays, the pointer points to the stack rather than the heap, but that doesn't matter when it comes to using the slice.

Exercise

The exercise for this section is located in `06_ticket_management/10_slices`

6.11 Mutable slices

Every time we've talked about slice types (like `str` and `[T]`), we've used their immutable borrow form (`&str` and `&[T]`).
But slices can also be mutable!

Here's how you create a mutable slice:

```
let mut numbers = vec![1, 2, 3];
let slice: &mut [i32] = &mut numbers;
```

You can then modify the elements in the slice:

```
slice[0] = 42;
```

This will change the first element of the `Vec` to `42`.

Limitations

When working with immutable borrows, the recommendation was clear: prefer slice references over references to the owned type (e.g. `&[T]` over `&Vec<T>`).
That's **not** the case with mutable borrows.

Consider this scenario:

```
let mut numbers = Vec::with_capacity(2);
let mut slice: &mut [i32] = &mut numbers;
slice.push(1);
```

It won't compile!
`push` is a method on `Vec`, not on slices. This is the manifestation of a more general principle: Rust won't allow you to add or remove elements from a slice. You will only be able to modify/replace the elements that are already there.

In this regard, a `&mut Vec` or a `&mut String` are strictly more powerful than a `&mut [T]` or a `&mut str`.
Choose the type that best fits based on the operations you need to perform.

Exercise

The exercise for this section is located in `06_ticket_management/11_mutable_slices`

6.12 Ticket ids

Let's think again about our ticket management system.
Our ticket model right now looks like this:

```
pub struct Ticket {
    pub title: TicketTitle,
    pub description: TicketDescription,
    pub status: Status
}
```

One thing is missing here: an **identifier** to uniquely identify a ticket.
That identifier should be unique for each ticket. That can be guaranteed by generating it automatically when a new ticket is created.

Refining the model

Where should the id be stored?
We could add a new field to the `Ticket` struct:

```
pub struct Ticket {
    pub id: TicketId,
    pub title: TicketTitle,
    pub description: TicketDescription,
    pub status: Status
}
```

But we don't know the id before creating the ticket. So it can't be there from the get-go.
It'd have to be optional:

```
pub struct Ticket {
    pub id: Option<TicketId>,
    pub title: TicketTitle,
    pub description: TicketDescription,
    pub status: Status
}
```

That's also not ideal—we'd have to handle the None case every single time we retrieve a ticket from the store, even though we know that the id should always be there once the ticket has been created.

The best solution is to have two different ticket **states**, represented by two separate types: a `TicketDraft` and a `Ticket`:

```rust
pub struct TicketDraft {
    pub title: TicketTitle,
    pub description: TicketDescription
}

pub struct Ticket {
    pub id: TicketId,
    pub title: TicketTitle,
    pub description: TicketDescription,
    pub status: Status
}
```

A `TicketDraft` is a ticket that hasn't been created yet. It doesn't have an id, and it doesn't have a status.
A `Ticket` is a ticket that has been created. It has an id and a status.
Since each field in `TicketDraft` and `Ticket` embeds its own constraints, we don't have to duplicate logic across the two types.

Exercise

The exercise for this section is located in `06_ticket_management/12_two_states`

6.13 Indexing

`TicketStore::get` returns an `Option<&Ticket>` for a given `TicketId`.
We've seen before how to access elements of arrays and vectors using Rust's indexing syntax:

```
let v = vec![0, 1, 2];
assert_eq!(v[0], 0);
```

How can we provide the same experience for `TicketStore`?
You guessed right: we need to implement a trait, `Index`!

Index

The `Index` trait is defined in Rust's standard library:

```
// Slightly simplified
pub trait Index<Idx>
{
    type Output;

    // Required method
    fn index(&self, index: Idx) -> &Self::Output;
}
```

It has:

- One generic parameter, `Idx`, to represent the index type
- One associated type, `Output`, to represent the type we retrieved using the index

Notice how the `index` method doesn't return an `Option`. The assumption is that `index` will panic if you try to access an element that's not there, as it happens for array and vec indexing.

Exercise

The exercise for this section is located in `06_ticket_management/13_index`

6.14 Mutable indexing

`Index` allows read-only access. It doesn't let you mutate the value you retrieved.

`IndexMut`

If you want to allow mutability, you need to implement the `IndexMut` trait.

```
// Slightly simplified
pub trait IndexMut<Idx>: Index<Idx>
{
    // Required method
    fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

`IndexMut` can only be implemented if the type already implements `Index`, since it unlocks an *additional* capability.

Exercise

The exercise for this section is located in `06_ticket_management/14_index_mut`

6.15 HashMap

Our implementation of `Index`/`IndexMut` is not ideal: we need to iterate over the entire `Vec` to retrieve a ticket by id; the algorithmic complexity is `O(n)`, where `n` is the number of tickets in the store.

We can do better by using a different data structure for storing tickets: a `HashMap<K, V>`.

```
use std::collections::HashMap;

// Type inference lets us omit an explicit type signature (which
// would be `HashMap<String, String>` in this example).
let mut book_reviews = HashMap::new();

book_reviews.insert(
    "Adventures of Huckleberry Finn".to_string(),
    "My favorite book.".to_string(),
);
```

`HashMap` works with key-value pairs. It's generic over both: `K` is the generic parameter for the key type, while `V` is the one for the value type.

The expected cost of insertions, retrievals and removals is **constant**, `O(1)`. That sounds perfect for our usecase, doesn't it?

Key requirements

There are no trait bounds on `HashMap`'s struct definition, but you'll find some on its methods. Let's look at `insert`, for example:

```
// Slightly simplified
impl<K, V> HashMap<K, V>
where
    K: Eq + Hash,
{
    pub fn insert(&mut self, k: K, v: V) -> Option<V> {
        // [...]
    }
}
```

The key type must implement the `Eq` and `Hash` traits.
Let's dig into those two.

Hash

A hashing function (or hasher) maps a potentially infinite set of a values (e.g. all possible strings) to a bounded range (e.g. a u64 value).
There are many different hashing functions around, each with different properties (speed, collision risk, reversibility, etc.).

A `HashMap`, as the name suggests, uses a hashing function behind the scene. It hashes your key and then uses that hash to store/retrieve the associated value. This strategy requires the key type must be hashable, hence the `Hash` trait bound on `K`.

You can find the `Hash` trait in the `std::hash` module:

```
pub trait Hash {
    // Required method
    fn hash<H>(&self, state: &mut H)
       where H: Hasher;
}
```

You will rarely implement `Hash` manually. Most of the times you'll derive it:

```
struct Person {
    id: u32,
    name: String,
}
```

Eq

`HashMap` must be able to compare keys for equality. This is particularly important when dealing with hash collisions—i.e. when two different keys hash to the same value.

You may wonder: isn't that what the `PartialEq` trait is for? Almost!
`PartialEq` is not enough for `HashMap` because it doesn't guarantee reflexivity, i.e. `a == a` is always `true`.
For example, floating point numbers (`f32` and `f64`) implement `PartialEq`, but they don't satisfy the reflexivity property: `f32::NAN == f32::NAN` is `false`.
Reflexivity is crucial for `HashMap` to work correctly: without it, you wouldn't be able to retrieve a value from the map using the same key you used to insert it.

The `Eq` trait extends `PartialEq` with the reflexivity property:

```
pub trait Eq: PartialEq {
    // No additional methods
}
```

It's a marker trait: it doesn't add any new methods, it's just a way for you to say to the compiler that the equality logic implemented in `PartialEq` is reflexive.

You can derive `Eq` automatically when you derive `PartialEq`:

```rust
struct Person {
    id: u32,
    name: String,
}
```

Eq and Hash are linked

There is an implicit contract between `Eq` and `Hash`: if two keys are equal, their hashes must be equal too. This is crucial for `HashMap` to work correctly. If you break this contract, you'll get nonsensical results when using `HashMap`.

Exercise

The exercise for this section is located in `06_ticket_management/15_hashmap`

6.16 Ordering

By moving from a `Vec` to a `HashMap` we have improved the performance of our ticket management system, and simplified our code in the process.
It's not all roses, though. When iterating over a `Vec`-backed store, we could be sure that the tickets would be returned in the order they were added.
That's not the case with a `HashMap`: you can iterate over the tickets, but the order is random.

We can recover a consistent ordering by switching from a `HashMap` to a `BTreeMap`.

BTreeMap

A `BTreeMap` guarantees that entries are sorted by their keys.
This is useful when you need to iterate over the entries in a specific order, or if you need to perform range queries (e.g. "give me all tickets with an id between 10 and 20").

Just like `HashMap`, you won't find trait bounds on the definition of `BTreeMap`. But you'll find trait bounds on its methods. Let's look at `insert`:

```
// `K` and `V` stand for the key and value types, respectively,
// just like in `HashMap`.
impl<K, V> BTreeMap<K, V> {
    pub fn insert(&mut self, key: K, value: V) -> Option<V>
    where
        K: Ord,
    {
        // implementation
    }
}
```

`Hash` is no longer required. Instead, the key type must implement the `Ord` trait.

Ord

The `Ord` trait is used to compare values.
While `PartialEq` is used to compare for equality, `Ord` is used to compare for ordering.

It's defined in `std::cmp`:

```
pub trait Ord: Eq + PartialOrd {
    fn cmp(&self, other: &Self) -> Ordering;
}
```

The `cmp` method returns an `Ordering` enum, which can be one of `Less`, `Equal`, or `Greater`.
`Ord` requires that two other traits are implemented: `Eq` and `PartialOrd`.

`PartialOrd`

`PartialOrd` is a weaker version of `Ord`, just like `PartialEq` is a weaker version of `Eq`. You can see why by looking at its definition:

```
pub trait PartialOrd: PartialEq {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering>;
}
```

`PartialOrd::partial_cmp` returns an `Option`—it is not guaranteed that two values can be compared.
For example, `f32` doesn't implement `Ord` because `NaN` values are not comparable, the same reason why `f32` doesn't implement `Eq`.

Implementing `Ord` and `PartialOrd`

Both `Ord` and `PartialOrd` can be derived for your types:

```
// You need to add `Eq` and `PartialEq` too,
// since `Ord` requires them.
struct TicketId(u64);
```

If you choose (or need) to implement them manually, be careful:

- `Ord` and `PartialOrd` must be consistent with `Eq` and `PartialEq`.
- `Ord` and `PartialOrd` must be consistent with each other.

Exercise

The exercise for this section is located in `06_ticket_management/16_btreemap`

Chapter 7

Intro

One of Rust's big promises is *fearless concurrency*: making it easier to write safe, concurrent programs. We haven't seen much of that yet. All the work we've done so far has been single-threaded. Time to change that!

In this chapter we'll make our ticket store multithreaded.
We'll have the opportunity to touch most of Rust's core concurrency features, including:

- Threads, using the `std::thread` module
- Message passing, using channels
- Shared state, using `Arc`, `Mutex` and `RwLock`
- `Send` and `Sync`, the traits that encode Rust's concurrency guarantees

We'll also discuss various design patterns for multithreaded systems and some of their trade-offs.

Exercise

The exercise for this section is located in `07_threads/00_intro`

7.1 Threads

Before we start writing multithreaded code, let's take a step back and talk about what threads are and why we might want to use them.

What is a thread?

A **thread** is an execution context managed by the underlying operating system. Each thread has its own stack and instruction pointer.

A single **process** can manage multiple threads. These threads share the same memory space, which means they can access the same data.

Threads are a **logical** construct. In the end, you can only run one set of instructions at a time on a CPU core, the **physical** execution unit.
Since there can be many more threads than there are CPU cores, the operating system's **scheduler** is in charge of deciding which thread to run at any given time, partitioning CPU time among them to maximize throughput and responsiveness.

`main`

When a Rust program starts, it runs on a single thread, the **main thread**.
This thread is created by the operating system and is responsible for running the `main` function.

```rust
use std::thread;
use std::time::Duration;

fn main() {
    loop {
        thread::sleep(Duration::from_secs(2));
        println!("Hello from the main thread!");
    }
}
```

`std::thread`

Rust's standard library provides a module, `std::thread`, that allows you to create and manage threads.

`spawn`

You can use `std::thread::spawn` to create new threads and execute code on them.

For example:

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        loop {
            thread::sleep(Duration::from_secs(1));
            println!("Hello from a thread!");
        }
    });

    loop {
        thread::sleep(Duration::from_secs(2));
        println!("Hello from the main thread!");
    }
}
```

If you execute this program on the Rust playground you'll see that the main thread and the spawned thread run concurrently.
Each thread makes progress independently of the other.

Process termination

When the main thread finishes, the overall process will exit.
A spawned thread will continue running until it finishes or the main thread finishes.

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        loop {
            thread::sleep(Duration::from_secs(1));
            println!("Hello from a thread!");
        }
    });

    thread::sleep(Duration::from_secs(5));
}
```

In the example above, you can expect to see the message "Hello from a thread!" printed roughly five times.

Then the main thread will finish (when the `sleep` call returns), and the spawned thread will be terminated since the overall process exits.

`join`

You can also wait for a spawned thread to finish by calling the `join` method on the `JoinHandle` that `spawn` returns.

```rust
use std::thread;
fn main() {
    let handle = thread::spawn(|| {
        println!("Hello from a thread!");
    });

    handle.join().unwrap();
}
```

In this example, the main thread will wait for the spawned thread to finish before exiting.
This introduces a form of **synchronization** between the two threads: you're guaranteed to see the message "Hello from a thread!" printed before the program exits, because the main thread won't exit until the spawned thread has finished.

Exercise

The exercise for this section is located in `07_threads/01_threads`

7.2 'static

If you tried to borrow a slice from the vector in the previous exercise, you probably got a compiler error that looks something like this:

```
error[E0597]: `v` does not live long enough
   |
11 | pub fn sum(v: Vec<i32>) -> i32 {
   |            - binding `v` declared here
...
15 |     let right = &v[split_point..];
   |                  ^ borrowed value does not live long enough
16 |     let left_handle = spawn(move || left.iter().sum::<i32>());
   |                             --------------------------------
                    argument requires that `v` is borrowed for `'static`
19 | }
   |  - `v` dropped here while still borrowed
```

`argument requires that v is borrowed for 'static`, what does that mean?

The `'static` lifetime is a special lifetime in Rust.
It means that the value will be valid for the entire duration of the program.

Detached threads

A thread launched via `thread::spawn` can **outlive** the thread that spawned it.
For example:

```rust
use std::thread;

fn f() {
    thread::spawn(|| {
        thread::spawn(|| {
            loop {
                thread::sleep(std::time::Duration::from_secs(1));
                println!("Hello from the detached thread!");
            }
        });
    });
}
```

In this example, the first spawned thread will in turn spawn a child thread that prints a message every second.
The first thread will then finish and exit. When that happens, its child thread will **continue running** for as long as the overall process is running.
In Rust's lingo, we say that the child thread has **outlived** its parent.

'static lifetime

Since a spawned thread can:

- outlive the thread that spawned it (its parent thread)
- run until the program exits

it must not borrow any values that might be dropped before the program exits; violating this constraint would expose us to a use-after-free bug.
That's why `std::thread::spawn`'s signature requires that the closure passed to it has the `'static` lifetime:

```rust
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static
{
    // [..]
}
```

'static is not (just) about references

All values in Rust have a lifetime, not just references.

In particular, a type that owns its data (like a `Vec` or a `String`) satisfies the `'static` constraint: if you own it, you can keep working with it for as long as you want, even after the function that originally created it has returned.

You can thus interpret `'static` as a way to say:

- Give me an owned value
- Give me a reference that's valid for the entire duration of the program

The first approach is how you solved the issue in the previous exercise: by allocating new vectors to hold the left and right parts of the original vector, which were then moved into the spawned threads.

'static references

Let's talk about the second case, references that are valid for the entire duration of the program.

Static data

The most common case is a reference to **static data**, such as string literals:

```rust
let s: &'static str = "Hello world!";
```

Since string literals are known at compile-time, Rust stores them *inside* your executable, in a region known as **read-only data segment**. All references pointing to that region will therefore be valid for as long as the program runs; they satisfy the `'static` contract.

Further reading

- The data segment

Exercise

The exercise for this section is located in `07_threads/02_static`

7.3 Leaking data

The main concern around passing references to spawned threads is use-after-free bugs: accessing data using a pointer to a memory region that's already been freed/de-allocated.
If you're working with heap-allocated data, you can avoid the issue by telling Rust that you'll never reclaim that memory: you choose to **leak memory**, intentionally.

This can be done, for example, using the `Box::leak` method from Rust's standard library:

```rust
// Allocate a `u32` on the heap, by wrapping it in a `Box`.
let x = Box::new(41u32);
// Tell Rust that you'll never free that heap allocation
// using `Box::leak`. You can thus get back a 'static reference.
let static_ref: &'static mut u32 = Box::leak(x);
```

Data leakage is process-scoped

Leaking data is dangerous: if you keep leaking memory, you'll eventually run out and crash with an out-of-memory error.

```rust
// If you leave this running for a while,
// it'll eventually use all the available memory.
fn oom_trigger() {
    loop {
        let v: Vec<usize> = Vec::with_capacity(1024);
        v.leak();
    }
}
```

At the same time, memory leaked via `leak` method is not truly forgotten.
The operating system can map each memory region to the process responsible for it.
When the process exits, the operating system will reclaim that memory.

Keeping this in mind, it can be OK to leak memory when:

- The amount of memory you need to leak is not unbounded/known upfront, or
- Your process is short-lived and you're confident you won't exhaust all the available memory before it exits

"Let the OS deal with it" is a perfectly valid memory management strategy if your usecase allows for it.

Exercise

The exercise for this section is located in `07_threads/03_leak`

7.4 Scoped threads

All the lifetime issues we discussed so far have a common source: the spawned thread can outlive its parent.
We can sidestep this issue by using **scoped threads**.

```
let v = vec![1, 2, 3];
let midpoint = v.len() / 2;

std::thread::scope(|scope| {
    scope.spawn(|| {
        let first = &v[..midpoint];
        println!("Here's the first half of v: {first:?}");
    });
    scope.spawn(|| {
        let second = &v[midpoint..];
        println!("Here's the second half of v: {second:?}");
    });
});

println!("Here's v: {v:?}");
```

Let's unpack what's happening.

`scope`

The `std::thread::scope` function creates a new **scope**.
`std::thread::scope` takes as input a closure, with a single argument: a `Scope` instance.

Scoped spawns

`Scope` exposes a `spawn` method.
Unlike `std::thread::spawn`, all threads spawned using a `Scope` will be **automatically joined** when the scope ends.

If we were to "translate" the previous example to `std::thread::spawn`, it'd look like this:

```
let v = vec![1, 2, 3];
let midpoint = v.len() / 2;

let handle1 = std::thread::spawn(|| {
    let first = &v[..midpoint];
    println!("Here's the first half of v: {first:?}");
```

```
});
let handle2 = std::thread::spawn(|| {
    let second = &v[midpoint..];
    println!("Here's the second half of v: {second:?}");
});

handle1.join().unwrap();
handle2.join().unwrap();

println!("Here's v: {v:?}");
```

Borrowing from the environment

The translated example wouldn't compile, though: the compiler would complain that `&v` can't be used from our spawned threads since its lifetime isn't `'static`.

That's not an issue with `std::thread::scope`—you can **safely borrow from the environment**.

In our example, `v` is created before the spawning points. It will only be dropped *after* `scope` returns. At the same time, all threads spawned inside `scope` are guaranteed to finish *before* `scope` returns, therefore there is no risk of having dangling references.

The compiler won't complain!

Exercise

The exercise for this section is located in `07_threads/04_scoped_threads`

7.5 Channels

All our spawned threads have been fairly short-lived so far.
Get some input, run a computation, return the result, shut down.

For our ticket management system, we want to do something different: a client-server architecture.

We will have **one long-running server thread**, responsible for managing our state, the stored tickets.

We will then have **multiple client threads**.
Each client will be able to send **commands** and **queries** to the stateful thread, in order to change its state (e.g. add a new ticket) or retrieve information (e.g. get the status of a ticket).
Client threads will run concurrently.

Communication

So far we've only had very limited parent-child communication:

- The spawned thread borrowed/consumed data from the parent context
- The spawned thread returned data to the parent when joined

This isn't enough for a client-server design.
Clients need to be able to send and receive data from the server thread *after* it has been launched.

We can solve the issue using **channels**.

Channels

Rust's standard library provides **multi-producer, single-consumer** (mpsc) channels in its `std::sync::mpsc` module.
There are two channel flavours: bounded and unbounded. We'll stick to the unbounded version for now, but we'll discuss the pros and cons later on.

Channel creation looks like this:

```rust
use std::sync::mpsc::channel;

let (sender, receiver) = channel();
```

You get a sender and a receiver.
You call `send` on the sender to push data into the channel.
You call `recv` on the receiver to pull data from the channel.

Multiple senders

`Sender` is clonable: we can create multiple senders (e.g. one for each client thread) and they will all push data into the same channel.

`Receiver`, instead, is not clonable: there can only be a single receiver for a given channel.

That's what **mpsc** (multi-producer single-consumer) stands for!

Message type

Both `Sender` and `Receiver` are generic over a type parameter `T`.
That's the type of the *messages* that can travel on our channel.

It could be a `u64`, a struct, an enum, etc.

Errors

Both `send` and `recv` can fail.
`send` returns an error if the receiver has been dropped.
`recv` returns an error if all senders have been dropped and the channel is empty.

In other words, `send` and `recv` error when the channel is effectively closed.

Exercise

The exercise for this section is located in `07_threads/05_channels`

7.6 Interior mutability

Let's take a moment to reason about the signature of `Sender`'s `send`:

```rust
impl<T> Sender<T> {
    pub fn send(&self, t: T) -> Result<(), SendError<T>> {
        // [...]
    }
}
```

`send` takes `&self` as its argument.
But it's clearly causing a mutation: it's adding a new message to the channel. What's even more interesting is that `Sender` is cloneable: we can have multiple instances of `Sender` trying to modify the channel state **at the same time**, from different threads.

That's the key property we are using to build this client-server architecture. But why does it work? Doesn't it violate Rust's rules about borrowing? How are we performing mutations via an *immutable* reference?

Shared rather than immutable references

When we introduced the borrow-checker, we named the two types of references we can have in Rust:

- immutable references (`&T`)
- mutable references (`&mut T`)

It would have been more accurate to name them:

- shared references (`&T`)
- exclusive references (`&mut T`)

Immutable/mutable is a mental model that works for the vast majority of cases, and it's a great one to get started with Rust. But it's not the whole story, as you've just seen: `&T` doesn't actually guarantee that the data it points to is immutable.
Don't worry, though: Rust is still keeping its promises. It's just that the terms are a bit more nuanced than they might seem at first.

`UnsafeCell`

Whenever a type allows you to mutate data through a shared reference, you're dealing with **interior mutability**.

By default, the Rust compiler assumes that shared references are immutable. It **optimises your code** based on that assumption.
The compiler can reorder operations, cache values, and do all sorts of magic to make your code faster.

You can tell the compiler "No, this shared reference is actually mutable" by wrapping the data in an `UnsafeCell`.
Every time you see a type that allows interior mutability, you can be certain that `UnsafeCell` is involved, either directly or indirectly.
Using `UnsafeCell`, raw pointers and `unsafe` code, you can mutate data through shared references.

Let's be clear, though: `UnsafeCell` isn't a magic wand that allows you to ignore the borrow-checker!
`unsafe` code is still subject to Rust's rules about borrowing and aliasing. It's an (advanced) tool that you can leverage to build **safe abstractions** whose safety can't be directly expressed in Rust's type system. Whenever you use the `unsafe` keyword you're telling the compiler: "I know what I'm doing, I won't violate your invariants, trust me."

Every time you call an `unsafe` function, there will be documentation explaining its **safety preconditions**: under what circumstances it's safe to execute its `unsafe` block. You can find the ones for `UnsafeCell` in `std`'s documentation.

We won't be using `UnsafeCell` directly in this course, nor will we be writing `unsafe` code. But it's important to know that it's there, why it exists and how it relates to the types you use every day in Rust.

Key examples

Let's go through a couple of important `std` types that leverage interior mutability.
These are types that you'll encounter somewhat often in Rust code, especially if you peek under the hood of some the libraries you use.

Reference counting

`Rc` is a reference-counted pointer.
It wraps around a value and keeps track of how many references to the value exist.
When the last reference is dropped, the value is deallocated.
The value wrapped in an `Rc` is immutable: you can only get shared references to it.

```rust
use std::rc::Rc;

let a: Rc<String> = Rc::new("My string".to_string());
// Only one reference to the string data exists.
```

```rust
assert_eq!(Rc::strong_count(&a), 1);

// When we call `clone`, the string data is not copied!
// Instead, the reference count for `Rc` is incremented.
let b = Rc::clone(&a);
assert_eq!(Rc::strong_count(&a), 2);
assert_eq!(Rc::strong_count(&b), 2);
// ^ Both `a` and `b` point to the same string data
//   and share the same reference counter.
```

`Rc` uses `UnsafeCell` internally to allow shared references to increment and decrement the reference count.

`RefCell`

`RefCell` is one of the most common examples of interior mutability in Rust. It allows you to mutate the value wrapped in a `RefCell` even if you only have an immutable reference to the `RefCell` itself.

This is done via **runtime borrow checking**. The `RefCell` keeps track of the number (and type) of references to the value it contains at runtime. If you try to borrow the value mutably while it's already borrowed immutably, the program will panic, ensuring that Rust's borrowing rules are always enforced.

```rust
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow(); // Immutable borrow
let z = x.borrow_mut(); // Panics! There is an active immutable borrow.
```

Exercise

The exercise for this section is located in `07_threads/06_interior_mutability`

7.7 Two-way communication

In our current client-server implementation, communication flows in one direction: from the client to the server.
The client has no way of knowing if the server received the message, executed it successfully, or failed. That's not ideal.

To solve this issue, we can introduce a two-way communication system.

Response channel

We need a way for the server to send a response back to the client.
There are various ways to do this, but the simplest option is to include a `Sender` channel in the message that the client sends to the server. After processing the message, the server can use this channel to send a response back to the client.

This is a fairly common pattern in Rust applications built on top of message-passing primitives.

Exercise

The exercise for this section is located in `07_threads/07_ack`

7.8 A dedicated `Client` type

All the interactions from the client side have been fairly low-level: you have to manually create a response channel, build the command, send it to the server, and then call `recv` on the response channel to get the response.

This is a lot of boilerplate code that could be abstracted away, and that's exactly what we're going to do in this exercise.

Exercise

The exercise for this section is located in `07_threads/08_client`

7.9 Bounded vs unbounded channels

So far we've been using unbounded channels.
You can send as many messages as you want, and the channel will grow to accommodate them.
In a multi-producer single-consumer scenario, this can be problematic: if the producers enqueue messages at a faster rate than the consumer can process them, the channel will keep growing, potentially consuming all available memory.

Our recommendation is to **never** use an unbounded channel in a production system. You should always enforce an upper limit on the number of messages that can be enqueued using a **bounded channel**.

Bounded channels

A bounded channel has a fixed capacity.
You can create one by calling `sync_channel` with a capacity greater than zero:

```
use std::sync::mpsc::sync_channel;

let (sender, receiver) = sync_channel(10);
```

`receiver` has the same type as before, `Receiver<T>`.
`sender`, instead, is an instance of `SyncSender<T>`.

Sending messages

You have two different methods to send messages through a `SyncSender`:

- `send`: if there is space in the channel, it will enqueue the message and return `Ok(())`.
  If the channel is full, it will block and wait until there is space available.
- `try_send`: if there is space in the channel, it will enqueue the message and return `Ok(())`.
  If the channel is full, it will return `Err(TrySendError::Full(value))`, where `value` is the message that couldn't be sent.

Depending on your use case, you might want to use one or the other.

Backpressure

The main advantage of using bounded channels is that they provide a form of **backpressure**.
They force the producers to slow down if the consumer can't keep up. The backpressure can then propagate through the system, potentially affecting the whole architecture and preventing end users from overwhelming the system with requests.

Exercise

The exercise for this section is located in `07_threads/09_bounded`

7.10 Update operations

So far we've implemented only insertion and retrieval operations.
Let's see how we can expand the system to provide an update operation.

Legacy updates

In the non-threaded version of the system, updates were fairly straightforward: `TicketStore` exposed a `get_mut` method that allowed the caller to obtain a mutable reference to a ticket, and then modify it.

Multithreaded updates

The same strategy won't work in the current multithreaded version. The borrow checker would stop us: `SyncSender<&mut Ticket>` isn't `'static` because `&mut Ticket` doesn't satisfy the `'static` lifetime, therefore they can't be captured by the closure that gets passed to `std::thread::spawn`.

There are a few ways to work around this limitation. We'll explore a few of them in the following exercises.

Patching

We can't send a `&mut Ticket` over a channel, therefore we can't mutate on the client-side.
Can we mutate on the server-side?

We can, if we tell the server what needs to be changed. In other words, if we send a **patch** to the server:

```
struct TicketPatch {
    id: TicketId,
    title: Option<TicketTitle>,
    description: Option<TicketDescription>,
    status: Option<TicketStatus>,
}
```

The `id` field is mandatory, since it's required to identify the ticket that needs to be updated.
All other fields are optional:

- If a field is `None`, it means that the field should not be changed.
- If a field is `Some(value)`, it means that the field should be changed to `value`.

Exercise

The exercise for this section is located in `07_threads/10_patch`

7.11 Locks, Send and Arc

The patching strategy you just implemented has a major drawback: it's racy.
If two clients send patches for the same ticket roughly at same time, the server will apply them in an arbitrary order. Whoever enqueues their patch last will overwrite the changes made by the other client.

Version numbers

We could try to fix this by using a **version number**.
Each ticket gets assigned a version number upon creation, set to 0.
Whenever a client sends a patch, they must include the current version number of the ticket alongside the desired changes. The server will only apply the patch if the version number matches the one it has stored.

In the scenario described above, the server would reject the second patch, because the version number would have been incremented by the first patch and thus wouldn't match the one sent by the second client.

This approach is fairly common in distributed systems (e.g. when client and servers don't share memory), and it is known as **optimistic concurrency control**.
The idea is that most of the time, conflicts won't happen, so we can optimize for the common case. You know enough about Rust by now to implement this strategy on your own as a bonus exercise, if you want to.

Locking

We can also fix the race condition by introducing a **lock**.
Whenever a client wants to update a ticket, they must first acquire a lock on it. While the lock is active, no other client can modify the ticket.

Rust's standard library provides two different locking primitives: Mutex<T> and RwLock<T>.
Let's start with Mutex<T>. It stands for **mut**ual **ex**clusion, and it's the simplest kind of lock: it allows only one thread to access the data, no matter if it's for reading or writing.

Mutex<T> wraps the data it protects, and it's therefore generic over the type of the data.
You can't access the data directly: the type system forces you to acquire a lock first using either Mutex::lock or Mutex::try_lock. The former blocks until the lock is acquired, the latter returns immediately with an error if the lock can't be acquired.
Both methods return a guard object that dereferences to the data, allowing you to modify it. The lock is released when the guard is dropped.

```rust
use std::sync::Mutex;

// An integer protected by a mutex lock
let lock = Mutex::new(0);

// Acquire a lock on the mutex
let mut guard = lock.lock().unwrap();

// Modify the data through the guard,
// leveraging its `Deref` implementation
*guard += 1;

// The lock is released when `data` goes out of scope
// This can be done explicitly by dropping the guard
// or happen implicitly when the guard goes out of scope
drop(guard)
```

Locking granularity

What should our `Mutex` wrap?
The simplest option would be the wrap the entire `TicketStore` in a single `Mutex`.
This would work, but it would severely limit the system's performance: you wouldn't be able to read tickets in parallel, because every read would have to wait for the lock to be released.
This is known as **coarse-grained locking**.

It would be better to use **fine-grained locking**, where each ticket is protected by its own lock. This way, clients can keep working with tickets in parallel, as long as they aren't trying to access the same ticket.

```rust
// The new structure, with a lock for each ticket
struct TicketStore {
    tickets: BTreeMap<TicketId, Mutex<Ticket>>,
}
```

This approach is more efficient, but it has a downside: `TicketStore` has to become **aware** of the multithreaded nature of the system; up until now, `TicketStore` has been blissfully ignoring the existence of threads.
Let's go for it anyway.

Who holds the lock?

For the whole scheme to work, the lock must be passed to the client that wants to modify the ticket.

The client can then directly modify the ticket (as if they had a `&mut Ticket`) and release the lock when they're done.

This is a bit tricky.
We can't send a `Mutex<Ticket>` over a channel, because `Mutex` is not `Clone` and we can't move it out of the `TicketStore`. Could we send the `MutexGuard` instead?

Let's test the idea with a small example:

```rust
use std::thread::spawn;
use std::sync::Mutex;
use std::sync::mpsc::sync_channel;

fn main() {
    let lock = Mutex::new(0);
    let (sender, receiver) = sync_channel(1);
    let guard = lock.lock().unwrap();

    spawn(move || {
        receiver.recv().unwrap();
    });

    // Try to send the guard over the channel
    // to another thread
    sender.send(guard);
}
```

The compiler is not happy with this code:

```
error[E0277]: `MutexGuard<'_, i32>` cannot be sent between
              threads safely
   --> src/main.rs:10:7
    |
10  |   spawn(move || {
    |  _-----_^
    | | |
    | | required by a bound introduced by this call
11  | |     receiver.recv().unwrap();
12  | | });
    | |_^ `MutexGuard<'_, i32>` cannot be sent between threads safely
    |
    = help: the trait `Send` is not implemented for
            `MutexGuard<'_, i32>`, which is required by
            `{closure@src/main.rs:10:7: 10:14}: Send`
    = note: required for `Receiver<MutexGuard<'_, i32>>`
            to implement `Send`
note: required because it's used within this closure
```

`MutexGuard<'_, i32>` is not `Send`: what does it mean?

`Send`

`Send` is a marker trait that indicates that a type can be safely transferred from one thread to another.
`Send` is also an auto-trait, just like `Sized`; it's automatically implemented (or not implemented) for your type by the compiler, based on its definition.
You can also implement `Send` manually for your types, but it requires `unsafe` since you have to guarantee that the type is indeed safe to send between threads for reasons that the compiler can't automatically verify.

Channel requirements

`Sender<T>`, `SyncSender<T>` and `Receiver<T>` are `Send` if and only if `T` is `Send`.
That's because they are used to send values between threads, and if the value itself is not `Send`, it would be unsafe to send it between threads.

`MutexGuard`

`MutexGuard` is not `Send` because the underlying operating system primitives that `Mutex` uses to implement the lock require (on some platforms) that the lock must be released by the same thread that acquired it.
If we were to send a `MutexGuard` to another thread, the lock would be released by a different thread, which would lead to undefined behavior.

Our challenges

Summing it up:

- We can't send a `MutexGuard` over a channel. So we can't lock on the server-side and then modify the ticket on the client-side.
- We can send a `Mutex` over a channel because it's `Send` as long as the data it protects is `Send`, which is the case for `Ticket`. At the same time, we can't move the `Mutex` out of the `TicketStore` nor clone it.

How can we solve this conundrum?
We need to look at the problem from a different angle. To lock a `Mutex`, we don't need an owned value. A shared reference is enough, since `Mutex` uses internal mutability:

```rust
impl<T> Mutex<T> {
    // `&self`, not `self`!
    pub fn lock(&self) -> LockResult<MutexGuard<'_, T>> {
```

```rust
        // Implementation details
    }
}
```

It is therefore enough to send a shared reference to the client.
We can't do that directly, though, because the reference would have to be `'static` and that's not the case.
In a way, we need an "owned shared reference". It turns out that Rust has a type that fits the bill: `Arc`.

Arc to the rescue

`Arc` stands for **atomic reference counting**.
`Arc` wraps around a value and keeps track of how many references to the value exist. When the last reference is dropped, the value is deallocated.
The value wrapped in an `Arc` is immutable: you can only get shared references to it.

```rust
use std::sync::Arc;

let data: Arc<u32> = Arc::new(0);
let data_clone = Arc::clone(&data);

// `Arc<T>` implements `Deref<T>`, so can convert
// a `&Arc<T>` to a `&T` using deref coercion
let data_ref: &u32 = &data;
```

If you're having a déjà vu moment, you're right: `Arc` sounds very similar to `Rc`, the reference-counted pointer we introduced when talking about interior mutability. The difference is thread-safety: `Rc` is not `Send`, while `Arc` is. It boils down to the way the reference count is implemented: `Rc` uses a "normal" integer, while `Arc` uses an **atomic** integer, which can be safely shared and modified across threads.

`Arc<Mutex<T>>`

If we pair `Arc` with `Mutex`, we finally get a type that:

- Can be sent between threads, because:
  - `Arc` is `Send` if `T` is `Send`, and
  - `Mutex` is `Send` if `T` is `Send`.
  - `T` is `Ticket`, which is `Send`.
- Can be cloned, because `Arc` is `Clone` no matter what `T` is. Cloning an `Arc` increments the reference count, the data is not copied.

- Can be used to modify the data it wraps, because `Arc` lets you get a shared reference to `Mutex<T>` which can in turn be used to acquire a lock.

We have all the pieces we need to implement the locking strategy for our ticket store.

Further reading

- We won't be covering the details of atomic operations in this course, but you can find more information in the `std` documentation as well as in the "Rust atomics and locks" book.

Exercise

The exercise for this section is located in `07_threads/11_locks`

7.12 Readers and writers

Our new `TicketStore` works, but its read performance is not great: there can only be one client at a time reading a specific ticket, because `Mutex<T>` doesn't distinguish between readers and writers.

We can solve the issue by using a different locking primitive: `RwLock<T>`.
`RwLock<T>` stands for **read-write lock**. It allows **multiple readers** to access the data simultaneously, but only one writer at a time.

`RwLock<T>` has two methods to acquire a lock: `read` and `write`.
`read` returns a guard that allows you to read the data, while `write` returns a guard that allows you to modify it.

```rust
use std::sync::RwLock;

// An integer protected by a read-write lock
let lock = RwLock::new(0);

// Acquire a read lock on the RwLock
let guard1 = lock.read().unwrap();

// Acquire a **second** read lock
// while the first one is still active
let guard2 = lock.read().unwrap();
```

Trade-offs

On the surface, `RwLock<T>` seems like a no-brainer: it provides a superset of the functionality of `Mutex<T>`. Why would you ever use `Mutex<T>` if you can use `RwLock<T>` instead?

There are two key reasons:

- Locking a `RwLock<T>` is more expensive than locking a `Mutex<T>`.
  This is because `RwLock<T>` has to keep track of the number of active readers and writers, while `Mutex<T>` only has to keep track of whether the lock is held or not. This performance overhead is not an issue if there are more readers than writers, but if the workload is write-heavy `Mutex<T>` might be a better choice.
- `RwLock<T>` can cause **writer starvation**.
  If there are always readers waiting to acquire the lock, writers might never get a chance to run.
  `RwLock<T>` doesn't provide any guarantees about the order in which readers and writers are granted access to the lock. It depends on the policy implemented by the underlying OS, which might not be fair to writers.

In our case, we can expect the workload to be read-heavy (since most clients will be reading tickets, not modifying them), so `RwLock<T>` is a good choice.

Exercise

The exercise for this section is located in `07_threads/12_rw_lock`

7.13 Design review

Let's take a moment to review the journey we've been through.

Lockless with channel serialization

Our first implementation of a multithreaded ticket store used:

- a single long-lived thread (server), to hold the shared state
- multiple clients sending requests to it via channels from their own threads.

No locking of the state was necessary, since the server was the only one modifying the state. That's because the "inbox" channel naturally **serialized** incoming requests: the server would process them one by one.
We've already discussed the limitations of this approach when it comes to patching behaviour, but we didn't discuss the performance implications of the original design: the server could only process one request at a time, including reads.

Fine-grained locking

We then moved to a more sophisticated design, where each ticket was protected by its own lock and clients could independently decide if they wanted to read or atomically modify a ticket, acquiring the appropriate lock.

This design allows for better parallelism (i.e. multiple clients can read tickets at the same time), but it is still fundamentally **serial**: the server processes commands one by one. In particular, it hands out locks to clients one by one.

Could we remove the channels entirely and allow clients to directly access the `TicketStore`, relying exclusively on locks to synchronize access?

Removing channels

We have two problems to solve:

- Sharing `TicketStore` across threads
- Synchronizing access to the store

Sharing `TicketStore` across threads

We want all threads to refer to the same state, otherwise we don't really have a multithreaded system—we're just running multiple single-threaded systems in parallel.
We've already encountered this problem when we tried to share a lock across threads: we can use an `Arc`.

Synchronizing access to the store

There is one interaction that's still lockless thanks to the serialization provided by the channels: inserting (or removing) a ticket from the store.
If we remove the channels, we need to introduce (another) lock to synchronize access to the `TicketStore` itself.

If we use a `Mutex`, then it makes no sense to use an additional `RwLock` for each ticket: the `Mutex` will already serialize access to the entire store, so we wouldn't be able to read tickets in parallel anyway.
If we use a `RwLock`, instead, we can read tickets in parallel. We just need to pause all reads while inserting or removing a ticket.

Let's go down this path and see where it leads us.

Exercise

The exercise for this section is located in `07_threads/13_without_channels`

7.14 Sync

Before we wrap up this chapter, let's talk about another key trait in Rust's standard library: `Sync`.

`Sync` is an auto trait, just like `Send`.
It is automatically implemented by all types that can be safely **shared** between threads.

In order words: `T: Sync` means that `&T` is `Send`.

Sync doesn't imply Send

It's important to note that `Sync` doesn't imply `Send`.
For example: `MutexGuard` is not `Send`, but it is `Sync`.

It isn't `Send` because the lock must be released on the same thread that acquired it, therefore we don't want `MutexGuard` to be dropped on a different thread.
But it is `Sync`, because giving a `&MutexGuard` to another thread has no impact on where the lock is released.

Send doesn't imply Sync

The opposite is also true: `Send` doesn't imply `Sync`.
For example: `RefCell<T>` is `Send` (if `T` is `Send`), but it is not `Sync`.

`RefCell<T>` performs runtime borrow checking, but the counters it uses to track borrows are not thread-safe. Therefore, having multiple threads holding a `&RefCell` would lead to a data race, with potentially multiple threads obtaining mutable references to the same data. Hence `RefCell` is not `Sync`.
`Send` is fine, instead, because when we send a `RefCell` to another thread we're not leaving behind any references to the data it contains, hence no risk of concurrent mutable access.

Exercise

The exercise for this section is located in `07_threads/14_sync`

Chapter 8

Async Rust

Threads are not the only way to write concurrent programs in Rust.
In this chapter we'll explore another approach: **asynchronous programming**.

In particular, you'll get an introduction to:

- The `async`/`.await` keywords, to write asynchronous code effortlessly
- The `Future` trait, to represent computations that may not be complete yet
- `tokio`, the most popular runtime for running asynchronous code
- The cooperative nature of Rust asynchronous model, and how this affects your code

Exercise

The exercise for this section is located in `08_futures/00_intro`

8.1 Asynchronous functions

All the functions and methods you've written so far were eager.
Nothing happened until you invoked them. But once you did, they ran to completion: they did **all** their work, and then returned their output.

Sometimes that's undesirable.
For example, if you're writing an HTTP server, there might be a lot of **waiting**: waiting for the request body to arrive, waiting for the database to respond, waiting for a downstream service to reply, etc.

What if you could do something else while you're waiting?
What if you could choose to give up midway through a computation?
What if you could choose to prioritise another task over the current one?

That's where **asynchronous functions** come in.

`async fn`

You use the `async` keyword to define an asynchronous function:

```rust
use tokio::net::TcpListener;

// This function is asynchronous
async fn bind_random() -> TcpListener {
    // [...]
}
```

What happens if you call `bind_random` as you would a regular function?

```rust
fn run() {
    // Invoke `bind_random`
    let listener = bind_random();
    // Now what?
}
```

Nothing happens!
Rust doesn't start executing `bind_random` when you call it, not even as a background task (as you might expect based on your experience with other languages). Asynchronous functions in Rust are **lazy**: they don't do any work until you explicitly ask them to. Using Rust's terminology, we say that `bind_random` returns a **future**, a type that represents a computation that may complete later. They're called futures because they implement the `Future` trait, an interface that we'll examine in detail later on in this chapter.

`.await`

The most common way to ask an asynchronous function to do some work is to use the `.await` keyword:

```rust
use tokio::net::TcpListener;

async fn bind_random() -> TcpListener {
    // [...]
}

async fn run() {
    // Invoke `bind_random` and wait for it to complete
    let listener = bind_random().await;
    // Now `listener` is ready
}
```

`.await` doesn't return control to the caller until the asynchronous function has run to completion—e.g. until the `TcpListener` has been created in the example above.

Runtimes

If you're puzzled, you're right to be!
We've just said that the perk of asynchronous functions is that they don't do **all** their work at once. We then introduced `.await`, which doesn't return until the asynchronous function has run to completion. Haven't we just re-introduced the problem we were trying to solve? What's the point?

Not quite! A lot happens behind the scenes when you call `.await`!
You're yielding control to an **async runtime**, also known as an **async executor**. Executors are where the magic happens: they are in charge of managing all your ongoing asynchronous **tasks**. In particular, they balance two different goals:

- **Progress**: they make sure that tasks make progress whenever they can.
- **Efficiency**: if a task is waiting for something, they try to make sure that another task can run in the meantime, fully utilising the available resources.

No default runtime

Rust is fairly unique in its approach to asynchronous programing: there is no default runtime. The standard library doesn't ship with one. You need to bring your own!

In most cases, you'll choose one of the options available in the ecosystem. Some runtimes are designed to be broadly applicable, a solid option for most applications.

tokio and async-std belong to this category. Other runtimes are optimised for specific use cases—e.g. embassy for embedded systems.

Throughout this course we'll rely on tokio, the most popular runtime for general-purpose asynchronous programming in Rust.

#[tokio::main]

The entrypoint of your executable, the main function, must be a synchronous function. That's where you're supposed to set up and launch your chosen async runtime.

Most runtimes provides a macro to make this easier. For tokio, it's tokio::main:

```rust
async fn main() {
    // Your async code goes here
}
```

which expands to:

```rust
fn main() {
    let rt = tokio::runtime::Runtime::new().unwrap();
    rt.block_on(
        // Your async function goes here
        // [...]
    );
}
```

#[tokio::test]

The same goes for tests: they must be synchronous functions.
Each test function is run in its own thread, and you're responsible for setting up and launching an async runtime if you need to run async code in your tests.
tokio provides a #[tokio::test] macro to make this easier:

```rust
async fn my_test() {
    // Your async test code goes here
}
```

Exercise

The exercise for this section is located in 08_futures/01_async_fn

8.2 Spawning tasks

Your solution to the previous exercise should look something like this:

```rust
pub async fn echo(listener: TcpListener) -> Result<(), anyhow::Error> {
    loop {
        let (mut socket, _) = listener.accept().await?;
        let (mut reader, mut writer) = socket.split();
        tokio::io::copy(&mut reader, &mut writer).await?;
    }
}
```

This is not bad!
If a long time passes between two incoming connections, the `echo` function will be idle (since `TcpListener::accept` is an asynchronous function), thus allowing the executor to run other tasks in the meantime.

But how can we actually have multiple tasks running concurrently?
If we always run our asynchronous functions until completion (by using `.await`), we'll never have more than one task running at a time.

This is where the `tokio::spawn` function comes in.

`tokio::spawn`

`tokio::spawn` allows you to hand off a task to the executor, **without waiting for it to complete**.
Whenever you invoke `tokio::spawn`, you're telling `tokio` to continue running the spawned task, in the background, **concurrently** with the task that spawned it.

Here's how you can use it to process multiple connections concurrently:

```rust
use tokio::net::TcpListener;

pub async fn echo(listener: TcpListener) -> Result<(), anyhow::Error> {
    loop {
        let (mut socket, _) = listener.accept().await?;
        // Spawn a background task to handle the connection
        // thus allowing the main task to immediately start
        // accepting new connections
        tokio::spawn(async move {
            let (mut reader, mut writer) = socket.split();
            tokio::io::copy(&mut reader, &mut writer).await?;
        });
    }
}
```

Asynchronous blocks

In this example, we've passed an **asynchronous block** to `tokio::spawn`: `async move { /* */ }` Asynchronous blocks are a quick way to mark a region of code as asynchronous without having to define a separate async function.

`JoinHandle`

`tokio::spawn` returns a `JoinHandle`.
You can use `JoinHandle` to `.await` the background task, in the same way we used `join` for spawned threads.

```
pub async fn run() {
    // Spawn a background task to ship telemetry data
    // to a remote server
    let handle = tokio::spawn(emit_telemetry());
    // In the meantime, do some other useful work
    do_work().await;
    // But don't return to the caller until
    // the telemetry data has been successfully delivered
    handle.await;
}

pub async fn emit_telemetry() {
    // [...]
}

pub async fn do_work() {
    // [...]
}
```

Panic boundary

If a task spawned with `tokio::spawn` panics, the panic will be caught by the executor. If you don't `.await` the corresponding `JoinHandle`, the panic won't be propagated to the spawner. Even if you do `.await` the `JoinHandle`, the panic won't be propagated automatically. Awaiting a `JoinHandle` returns a `Result`, with `JoinError` as its error type. You can then check if the task panicked by calling `JoinError::is_panic` and choose what to do with the panic—either log it, ignore it, or propagate it.

```
use tokio::task::JoinError;

pub async fn run() {
    let handle = tokio::spawn(work());
    if let Err(e) = handle.await {
```

```rust
        if let Ok(reason) = e.try_into_panic() {
            // The task has panicked
            // We resume unwinding the panic,
            // thus propagating it to the current task
            panic::resume_unwind(reason);
        }
    }
}

pub async fn work() {
    // [...]
}
```

`std::thread::spawn` vs `tokio::spawn`

You can think of `tokio::spawn` as the asynchronous sibling of `std::thread::spawn`.

Notice a key difference: with `std::thread::spawn`, you're delegating control to the OS scheduler. You're not in control of how threads are scheduled.

With `tokio::spawn`, you're delegating to an async executor that runs entirely in user space. The underlying OS scheduler is not involved in the decision of which task to run next. We're in charge of that decision now, via the executor we chose to use.

Exercise

The exercise for this section is located in `08_futures/02_spawn`

8.3 Runtime architecture

So far we've been talking about async runtimes as an abstract concept. Let's dig a bit deeper into the way they are implemented—as you'll see soon enough, it has an impact on our code.

Flavors

`tokio` ships two different runtime *flavors*.

You can configure your runtime via `tokio::runtime::Builder`:

- `Builder::new_multi_thread` gives you a **multithreaded `tokio` runtime**
- `Builder::new_current_thread` will instead rely on the **current thread** for execution.

`#[tokio::main]` returns a multithreaded runtime by default, while `#[tokio::test]` uses a current thread runtime out of the box.

Current thread runtime

The current-thread runtime, as the name implies, relies exclusively on the OS thread it was launched on to schedule and execute tasks.
When using the current-thread runtime, you have **concurrency** but no **parallelism**: asynchronous tasks will be interleaved, but there will always be at most one task running at any given time.

Multithreaded runtime

When using the multithreaded runtime, instead, there can up to `N` tasks running *in parallel* at any given time, where `N` is the number of threads used by the runtime. By default, `N` matches the number of available CPU cores.

There's more: `tokio` performs **work-stealing**.
If a thread is idle, it won't wait around: it'll try to find a new task that's ready for execution, either from a global queue or by stealing it from the local queue of another thread.
Work-stealing can have significant performance benefits, especially on tail latencies, whenever your application is dealing with workloads that are not perfectly balanced across threads.

Implications

`tokio::spawn` is flavor-agnostic: it'll work no matter if you're running on the multi-threaded or current-thread runtime. The downside is that the signature assumes the worst case (i.e. multithreaded) and is constrained accordingly:

```rust
pub fn spawn<F>(future: F) -> JoinHandle<F::Output>
where
    F: Future + Send + 'static,
    F::Output: Send + 'static,
{ /* */ }
```

Let's ignore the `Future` trait for now to focus on the rest.
`spawn` is asking all its inputs to be `Send` and have a `'static` lifetime.

The `'static` constraint follows the same rationale of the `'static` constraint on `std::thread::spawn`: the spawned task may outlive the context it was spawned from, therefore it shouldn't depend on any local data that may be de-allocated after the spawning context is destroyed.

```rust
fn spawner() {
    let v = vec![1, 2, 3];
    // This won't work, since `&v` doesn't
    // live long enough.
    tokio::spawn(async {
        for x in &v {
            println!("{x}")
        }
    })
}
```

`Send`, on the other hand, is a direct consequence of `tokio`'s work-stealing strategy: a task that was spawned on thread A may end up being moved to thread B if that's idle, thus requiring a `Send` bound since we're crossing thread boundaries.

```rust
fn spawner(input: Rc<u64>) {
    // This won't work either, because
    // `Rc` isn't `Send`.
    tokio::spawn(async move {
        println!("{}", input);
    })
}
```

Exercise

The exercise for this section is located in `08_futures/03_runtime`

8.4 The Future trait

The local `Rc` problem

Let's go back to `tokio::spawn`'s signature:

```
pub fn spawn<F>(future: F) -> JoinHandle<F::Output>
    where
        F: Future + Send + 'static,
        F::Output: Send + 'static,
{ /* */ }
```

What does it *actually* mean for `F` to be `Send`?
It implies, as we saw in the previous section, that whatever value it captures from the spawning environment has to be `Send`. But it goes further than that.

Any value that's *held across a .await point* has to be `Send`.
Let's look at an example:

```
use std::rc::Rc;
use tokio::task::yield_now;

fn spawner() {
    tokio::spawn(example());
}

async fn example() {
    // A value that's not `Send`,
    // created _inside_ the async function
    let non_send = Rc::new(1);

    // A `.await` point that does nothing
    yield_now().await;

    // The local non-`Send` value is still needed
    // after the `.await`
    println!("{}", non_send);
}
```

The compiler will reject this code:

```
error: future cannot be sent between threads safely
    |
5   |     tokio::spawn(example());
    |                  ^^^^^^^^^
    | future returned by `example` is not `Send`
    |
```

```
note: future is not `Send` as this value is used across an await
    |
11  |     let non_send = Rc::new(1);
    |         -------- has type `Rc<i32>` which is not `Send`
12  |     // A `.await` point
13  |     yield_now().await;
    |                 ^^^^^
    |   await occurs here, with `non_send` maybe used later
note: required by a bound in `tokio::spawn`
    |
164 |     pub fn spawn<F>(future: F) -> JoinHandle<F::Output>
    |            ----- required by a bound in this function
165 |     where
166 |         F: Future + Send + 'static,
    |                     ^^^^ required by this bound in `spawn`
```

To understand why that's the case, we need to refine our understanding of Rust's asynchronous model.

The Future trait

We stated early on that `async` functions return **futures**, types that implement the `Future` trait. You can think of a future as a **state machine**. It's in one of two states:

- **pending**: the computation has not finished yet.
- **ready**: the computation has finished, here's the output.

This is encoded in the trait definition:

```rust
trait Future {
    type Output;

    // Ignore `Pin` and `Context` for now
    fn poll(
      self: Pin<&mut Self>,
      cx: &mut Context<'_>
    ) -> Poll<Self::Output>;
}
```

`poll`

The `poll` method is the heart of the `Future` trait.
A future on its own doesn't do anything. It needs to be **polled** to make progress.
When you call `poll`, you're asking the future to do some work. `poll` tries to make progress, and then returns one of the following:

- `Poll::Pending`: the future is not ready yet. You need to call `poll` again later.
- `Poll::Ready(value)`: the future has finished. `value` is the result of the computation, of type `Self::Output`.

Once `Future::poll` returns `Poll::Ready`, it should not be polled again: the future has completed, there's nothing left to do.

The role of the runtime

You'll rarely, if ever, be calling poll directly.
That's the job of your async runtime: it has all the required information (the `Context` in `poll`'s signature) to ensure that your futures are making progress whenever they can.

`async fn` and futures

We've worked with the high-level interface, asynchronous functions.
We've now looked at the low-level primitive, the `Future trait`.

How are they related?

Every time you mark a function as asynchronous, that function will return a future. The compiler will transform the body of your asynchronous function into a **state machine**: one state for each `.await` point.

Going back to our `Rc` example:

```rust
use std::rc::Rc;
use tokio::task::yield_now;

async fn example() {
    let non_send = Rc::new(1);
    yield_now().await;
    println!("{}", non_send);
}
```

The compiler would transform it into an enum that looks somewhat like this:

```rust
pub enum ExampleFuture {
    NotStarted,
    YieldNow(Rc<i32>),
    Terminated,
}
```

When `example` is called, it returns `ExampleFuture::NotStarted`. The future has never been polled yet, so nothing has happened.
When the runtime polls it the first time, `ExampleFuture` will advance until the next `.await` point: it'll stop at the `ExampleFuture::YieldNow(Rc<i32>)` stage of the state machine, returning `Poll::Pending`.
When it's polled again, it'll execute the remaining code (`println!`) and return `Poll::Ready(())`.

When you look at its state machine representation, `ExampleFuture`, it is now clear why `example` is not `Send`: it holds an `Rc`, therefore it cannot be `Send`.

Yield points

As you've just seen with `example`, every `.await` point creates a new intermediate state in the lifecycle of a future.
That's why `.await` points are also known as **yield points**: your future *yields control* back to the runtime that was polling it, allowing the runtime to pause it and (if necessary) schedule another task for execution, thus making progress on multiple fronts concurrently.

We'll come back to the importance of yielding in a later section.

Exercise

The exercise for this section is located in `08_futures/04_future`

8.5 Don't block the runtime

Let's circle back to yield points.
Unlike threads, **Rust tasks cannot be preempted**.

`tokio` cannot, on its own, decide to pause a task and run another one in its place. The control goes back to the executor **exclusively** when the task yields—i.e. when `Future::poll` returns `Poll::Pending` or, in the case of `async fn`, when you `.await` a future.

This exposes the runtime to a risk: if a task never yields, the runtime will never be able to run another task. This is called **blocking the runtime**.

What is blocking?

How long is too long? How much time can a task spend without yielding before it becomes a problem?

It depends on the runtime, the application, the number of in-flight tasks, and many other factors. But, as a general rule of thumb, try to spend less than 100 microseconds between yield points.

Consequences

Blocking the runtime can lead to:

- **Deadlocks**: if the task that's not yielding is waiting for another task to complete, and that task is waiting for the first one to yield, you have a deadlock. No progress can be made, unless the runtime is able to schedule the other task on a different thread.
- **Starvation**: other tasks might not be able to run, or might run after a long delay, which can lead to poor performances (e.g. high tail latencies).

Blocking is not always obvious

Some types of operations should generally be avoided in async code, like:

- Synchronous I/O. You can't predict how long it will take, and it's likely to be longer than 100 microseconds.
- Expensive CPU-bound computations.

The latter category is not always obvious though. For example, sorting a vector with a few elements is not a problem; that evaluation changes if the vector has billions of entries.

How to avoid blocking

OK, so how do you avoid blocking the runtime assuming you *must* perform an operation that qualifies or risks qualifying as blocking?
You need to move the work to a different thread. You don't want to use the so-called runtime threads, the ones used by `tokio` to run tasks.

`tokio` provides a dedicated threadpool for this purpose, called the **blocking pool**. You can spawn a synchronous operation on the blocking pool using the `tokio::task::spawn_blocking` function. `spawn_blocking` returns a future that resolves to the result of the operation when it completes.

```rust
use tokio::task;

fn expensive_computation() -> u64 {
    // [...]
}

async fn run() {
    let handle = task::spawn_blocking(expensive_computation);
    // Do other stuff in the meantime
    let result = handle.await.unwrap();
}
```

The blocking pool is long-lived. `spawn_blocking` should be faster than creating a new thread directly via `std::thread::spawn` because the cost of thread initialization is amortized over multiple calls.

Further reading

- Check out Alice Ryhl's blog post on the topic.

Exercise

The exercise for this section is located in `08_futures/05_blocking`

8.6 Async-aware primitives

If you browse `tokio`'s documentation, you'll notice that it provides a lot of types that "mirror" the ones in the standard library, but with an asynchronous twist: locks, channels, timers, and more.

When working in an asynchronous context, you should prefer these asynchronous alternatives to their synchronous counterparts.

To understand why, let's take a look at `Mutex`, the mutually exclusive lock we explored in the previous chapter.

Case study: `Mutex`

Let's look at a simple example:

```
use std::sync::{Arc, Mutex};

async fn run(m: Arc<Mutex<Vec<u64>>>) {
    let guard = m.lock().unwrap();
    http_call(&guard).await;
    println!("Sent {:?} to the server", &guard);
    // `guard` is dropped here
}

/// Use `v` as the body of an HTTP call.
async fn http_call(v: &[u64]) {
  // [...]
}
```

`std::sync::MutexGuard` and yield points

This code will compile, but it's dangerous.

We try to acquire a lock over a `Mutex` from `std` in an asynchronous context. We then hold on to the resulting `MutexGuard` across a yield point (the `.await` on `http_call`).

Let's imagine that there are two tasks executing `run`, concurrently, on a single-threaded runtime. We observe the following sequence of scheduling events:

```
     Task A          Task B
        |
  Acquire lock
Yields to runtime
        |
        +---------------+
```

```
                       |
             Tries to acquire lock
```

We have a deadlock. Task B will never manage to acquire the lock, because the lock is currently held by task A, which has yielded to the runtime before releasing the lock and won't be scheduled again because the runtime cannot preempt task B.

`tokio::sync::Mutex`

You can solve the issue by switching to `tokio::sync::Mutex`:

```rust
use std::sync::Arc;
use tokio::sync::Mutex;

async fn run(m: Arc<Mutex<Vec<u64>>>) {
    let guard = m.lock().await;
    http_call(&guard).await;
    println!("Sent {:?} to the server", &guard);
    // `guard` is dropped here
}
```

Acquiring the lock is now an asynchronous operation, which yields back to the runtime if it can't make progress.
Going back to the previous scenario, the following would happen:

```
       Task A          Task B
          |
  Acquires the lock
  Starts `http_call`
  Yields to runtime
          |
          +--------------+
                         |
             Tries to acquire the lock
              Cannot acquire the lock
                 Yields to runtime
                         |
          +--------------+
          |
`http_call` completes
  Releases the lock
   Yield to runtime
          |
          +--------------+
                         |
                 Acquires the lock
                       [...]
```

All good!

Multithreaded won't save you

We've used a single-threaded runtime as the execution context in our previous example, but the same risk persists even when using a multithreaded runtime.
The only difference is in the number of concurrent tasks required to create the deadlock: in a single-threaded runtime, 2 are enough; in a multithreaded runtime, we would need `N+1` tasks, where `N` is the number of runtime threads.

Downsides

Having an async-aware `Mutex` comes with a performance penalty.
If you're confident that the lock isn't under significant contention *and* you're careful to never hold it across a yield point, you can still use `std::sync::Mutex` in an asynchronous context.

But weigh the performance benefit against the liveness risk you will incur.

Other primitives

We used `Mutex` as an example, but the same applies to `RwLock`, semaphores, etc.
Prefer async-aware versions when working in an asynchronous context to minimise the risk of issues.

Exercise

The exercise for this section is located in `08_futures/06_async_aware_primitives`

8.7 Cancellation

What happens when a pending future is dropped?
The runtime will no longer poll it, therefore it won't make any further progress. In other words, its execution has been **cancelled**.

In the wild, this often happens when working with timeouts. For example:

```rust
use tokio::time::timeout;
use tokio::sync::oneshot;
use std::time::Duration;

async fn http_call() {
    // [...]
}

async fn run() {
    // Wrap the future with a `Timeout` set to expire in 10 milliseconds.
    let duration = Duration::from_millis(10);
    if let Err(_) = timeout(duration, http_call()).await {
        println!("Didn't receive a value within 10 ms");
    }
}
```

When the timeout expires, the future returned by `http_call` will be cancelled. Let's imagine that this is `http_call`'s body:

```rust
use std::net::TcpStream;

async fn http_call() {
    let (stream, _) = TcpStream::connect(/* */).await.unwrap();
    let request: Vec<u8> = /* */;
    stream.write_all(&request).await.unwrap();
}
```

Each yield point becomes a **cancellation point**.
`http_call` can't be preempted by the runtime, so it can only be discarded after it has yielded control back to the executor via `.await`. This applies recursively—e.g. `stream.write_all(&request)` is likely to have multiple yield points in its implementation. It is perfectly possible to see `http_call` pushing a *partial* request before being cancelled, thus dropping the connection and never finishing transmitting the body.

Clean up

Rust's cancellation mechanism is quite powerful—it allows the caller to cancel an ongoing task without needing any form of cooperation from the task itself.

At the same time, this can be quite dangerous. It may be desirable to perform a **graceful cancellation**, to ensure that some clean-up tasks are performed before aborting the operation.

For example, consider this fictional API for a SQL transaction:

```rust
async fn transfer_money(
    connection: SqlConnection,
    payer_id: u64,
    payee_id: u64,
    amount: u64
) -> Result<(), anyhow::Error> {
    let transaction = connection.begin_transaction().await?;
    update_balance(payer_id, amount, &transaction).await?;
    decrease_balance(payee_id, amount, &transaction).await?;
    transaction.commit().await?;
}
```

On cancellation, it'd be ideal to explicitly abort the pending transaction rather than leaving it hanging. Rust, unfortunately, doesn't provide a bullet-proof mechanism for this kind of **asynchronous** clean up operations.

The most common strategy is to rely on the `Drop` trait to schedule the required clean-up work. This can be by:

- Spawning a new task on the runtime
- Enqueueing a message on a channel
- Spawning a background thread

The optimal choice is contextual.

Cancelling spawned tasks

When you spawn a task using `tokio::spawn`, you can no longer drop it; it belongs to the runtime.
Nonetheless, you can use its `JoinHandle` to cancel it if needed:

```rust
async fn run() {
    let handle = tokio::spawn(/* some async task */);
    // Cancel the spawned task
    handle.abort();
}
```

Further reading

- Be extremely careful when using `tokio`'s `select!` macro to "race" two different futures. Retrying the same task in a loop is dangerous unless you can ensure **cancellation safety**. Check out `select!`'s documentation for more details.
  If you need to interleave two asynchronous streams of data (e.g. a socket and a channel), prefer using `StreamExt::merge` instead.
- A `CancellationToken` may be preferable to `JoinHandle::abort` in some cases.

Exercise

The exercise for this section is located in `08_futures/07_cancellation`

8.8 Outro

Rust's asynchronous model is quite powerful, but it does introduce additional complexity. Take time to know your tools: dive deep into `tokio`'s documentation and get familiar with its primitives to make the most out of it.

Keep in mind, as well, that there is ongoing work at the language and `std` level to streamline and "complete" Rust's asynchronous story. You may experience some rough edges in your day-to-day work due to some of these missing pieces.

A few recommendations for a mostly-pain-free async experience:

- **Pick a runtime and stick to it.**
  Some primitives (e.g. timers, I/O) are not portable across runtimes. Trying to mix runtimes is likely to cause you pain. Trying to write code that's runtime agnostic can significantly increase the complexity of your codebase. Avoid it if you can.
- **There is no stable `Stream`/`AsyncIterator` interface yet.**
  An `AsyncIterator` is, conceptually, an iterator that yields new items asynchronously. There is ongoing design work, but no consensus (yet). If you're using `tokio`, refer to `tokio_stream` as your go-to interface.
- **Be careful with buffering.**
  It is often the cause of subtle bugs. Check out "Barbara battles buffered streams" for more details.
- **There is no equivalent of scoped threads for asynchronous tasks**.
  Check out "The scoped task trilemma" for more details.

Don't let these caveats scare you: asynchronous Rust is being used effectively at *massive* scale (e.g. AWS, Meta) to power foundational services.
You will have to master it if you're planning building networked applications in Rust.

Exercise

The exercise for this section is located in `08_futures/08_outro`

Chapter 9

Epilogue

Our tour of Rust ends here.
It has been quite extensive, but by no means exhaustive: Rust is a language with a large surface area, and an even larger ecosystem!
Don't let this scare you, though: there's **no need to learn everything**. You'll pick up whatever is necessary to be effective in the domain (backend, embedded, CLIs, GUIs, etc.) **while working on your projects**.

In the end, there are no shortcuts: if you want to get good at something, you need to do it, over and over again. Throughout this course you wrote a fair amount of Rust, enough to get the language and its syntax flowing under your fingers. It'll take many more lines of code to feel it "yours", but that moment will come without a doubt if you keep practicing.

Going further

Let's close with some pointers to additional resources that you might find useful as you move forward in your journey with Rust.

Exercises

You can find more exercises to practice Rust in the `rustlings` project and on exercism.io's Rust track.

Introductory material

Check out the Rust book and "Programming Rust" if you're looking for a different perspective on the same concepts we covered throughout this course. You'll certainly learn something new since they don't cover exactly the same topics; Rust has a lot of surface area!

Advanced material

If you want to dive deeper into the language, refer to the Rustonomicon and "Rust for Rustaceans".
The "Decrusted" series is another excellent resource to learn more about the internals of many of the most popular Rust libraries.

Domain-specific material

If you want to use Rust for backend development, check out "Zero to Production in Rust".
If you want to use Rust for embedded development, check out the Embedded Rust book.

Masterclasses

You can then find resources on key topics that cut across domains.
For testing, check out "Advanced testing, going beyond the basics".
For telemetry, check out "You can't fix what you can't see".