From JavaScript to Rust

Mapp common JavaScript and node.js workflows to the Rust ecosystem

By Jarrod Overson, Vino Technologies, and contributors

Table of Contents

Preface

This book started as a series of posts on vino.dev's blog during December 2021. The series generated more interest than we ever expected and it started to take on a life of its own. Since then we've had numerous publishers and volunteers looking to extend this series and its reach. We converted the blog posts to asciidoc and put together the scaffolding necessary to turn it into an e-book. The book's source files and all the project's source code are open source under Creative Commons licenses. You are welcome and encouraged to submit contributions, fixed, translations, or new chapters as you see fit.

Thank you everyone who contributed, provided feedback, and otherwise helped make the effort worthwhile. Special thanks go to my wife Kate Lane and my children Finn, Norah, and Elliot who tolerated me writing all day, every day over the 2021 holidays.

A note on contributions:

The spirit of this book and its original posts can be summed up as "Get to the point. Fast." The JavaScript and node.js community is top-notch when it comes to practical examples and working example code. I missed that when I started working with Rust. I began the original series to help those coming down this same path and I'd like to maintain that spirit as much as possible. If the book is outright **wrong**, then please contribute fixes. Otherwise, please err on the side of "gets the point across" vs 100% technical accuracy.

1. Introduction

1.1. A guide to Rust from a node.js developer's perspective.

Each chapter will take concepts you know in JavaScript and node.js and translate them to their Rust counterparts. The first chapters start with the basics, like getting set up with Rust via a tool similar to `nvm` (`rustup`), using the package manager (`cargo`), and setting up VS Code. Later chapters go over language gotchas, how to perform common JavaScript tasks in Rust, and we'll finish up by touching on solid dependencies to start adding to your projectss.

1.2. Wait, why does anyone need to learn anything but JavaScript?

I *love* JavaScript. I've been coding JavaScript it since I first saw it in Netscape. I've written more lines of JavaScript than any other language. I'm a fan, but I know where the language falls short. It's fast, but not that fast. It's easy to write, but easy to screw up. Large projects become unwieldy fast. TypeScript helps scale JavaScript but it adds its own complexity and still doesn't make anything faster. Server-side JavaScript relies on node.js which is common but not ubiquitous. If you want to distribute something self-contained, there aren't great answers.

When you start stretching passed what JavaScript is best at, it's helpful to have another language to turn to.

1.3. Why Rust?

You could use C, C++, C#, Go, Java, Kotlin, Haskell or a hundred others. Rust is notoriously difficult even for system programmers to get into. Why bother with Rust? Think about your languages as tools in your toolbox. When you fill your toolbox, you don't want 10 tools that solve similar problems. You want tools that complement each other and give you the ability to fix everything an anything. You already have JavaScript, a developer super-tool. It's a high level language that's good enough to run just about everything everywhere. If you're picking up a new language, you might as well go to the extreme and pick a no-compromise, low-level powerhouse.

Also, WebAssembly.

Rust's tooling and support for WebAssembly is better than everything else out there. You can rewrite CPU-heavy JavaScript logic into Rust and run it as WebAssembly. Which basically makes you a superhero. With JavaScript and Rust, there's nothing you can't handle.

1.4. How to use this book

This book is not a deep, comprehensive Rust tutorial. It's meant to bootstrap experienced programmers into Rust. We'll take common node.js workflows and idiomatic JavaScript and TypeScript and map them to their Rust counterparts. This book balances technical accuracy with

readability. It errs on the side of "gets the point across" vs being 100% correct. When something is glossed over, we'll add links for those looking to dive deeper.

2. Installing rust with `rustup`

nvm (or nvm-windows) are indispensible tools. They manage seamlessly installing and switching between versions of node.js on the same system.

The equivalent in Rust's world is rustup.

Rustup manages your Rust installation as well as additonal targets (like WebAssembly) and core tools like `cargo` (Rust's `npm`), `clippy` (Rust's `eslint`), `rustfmt` (Rust's `prettier`).

After installing `rustup`, run it without any subcommands and explore what it has to offer.

```
$ rustup
rustup 1.24.3 (ce5817a94 2021-05-31)
The Rust toolchain installer

USAGE:
    rustup [FLAGS] [+toolchain] <SUBCOMMAND>

FLAGS:
    -v, --verbose    Enable verbose output
    -q, --quiet      Disable progress output
    -h, --help       Prints help information
    -V, --version    Prints version information

ARGS:
    <+toolchain>    release channel (e.g. +stable) or custom toolchain to set override

SUBCOMMANDS:
    show           Show the active and installed toolchains or profiles
    update         Update Rust toolchains and rustup
    check          Check for updates to Rust toolchains and rustup
    default        Set the default toolchain
    toolchain      Modify or query the installed toolchains
    target         Modify a toolchain's supported targets
    component      Modify a toolchain's installed components
    override       Modify directory toolchain overrides
    run            Run a command with an environment configured for a given toolchain
    which          Display which binary will be run for a given command
    doc            Open the documentation for the current toolchain
    man            View the man page for a given command
    self           Modify the rustup installation
    set            Alter rustup settings
    completions    Generate tab-completion scripts for your shell
    help           Prints this message or the help of the given subcommand(s)

DISCUSSION:
    Rustup installs The Rust Programming Language from the official
    release channels, enabling you to easily switch between stable,
    beta, and nightly compilers and keep them updated. It makes
    cross-compiling simpler with binary builds of the standard library
    for common platforms.

    If you are new to Rust consider running `rustup doc --book` to
    learn Rust.
```

`rustup show` will show you what is currently installed.

`rustup completions` will help you enable CLI autocompletion for tools like `rustup` and `cargo`.

`rustup component` lets you add additonal components.

`rustup update` will update you to the latest version.

`rustup install stable|nightly|1.57` will install a specific version or the latest stable/nightly versions.

By default, rustup will install the latest version of `rust` and `cargo` and you should be ready to go right away. Give it a shot with.

```
$ rustc --version
rustc 1.57.0 (59eed8a2a 2021-11-01)

$ cargo --version
cargo 1.56.0 (4ed5d137b 2021-10-04)
```

If it doesn't work, you may need to restart your shell to update your PATH.

2.1. `rust-toolchain.toml`

Specifying your toolchain with rustup is easy enough. As you get deeper, you may get into configurations where different projects require different toolchains or Rust versions. That's where `rust-toolchain.toml` comes into play. Specify your project's required toolchain, targets, and supporting tools here so that `cargo` and `rustup` can work automagically, e.g.

`toml {title = "rust-toolchain.toml"}

channel = "1.56.0" components = ["rustfmt", "clippy"] `

2.2. Next steps

Next up we'll take a look at `cargo`, Rust's `npm` and the additional tools that will help reach parity with common workflows: Chapter 2: From npm to cargo.

3. From npm to cargo

3.1. Introduction

`cargo` is Rust's package manager and operates similarly to `npm` from node's universe. Cargo downloads dependiencs from crates.io by default. You can register an account and publish modules just as you would on npmjs.com. With some minor mapping you can translate almost everything you're used to in node to Rust.

3.2. `npm` to `cargo` mapping

Project settings file

In node.js you have `package.json`. In Rust you have `Cargo.toml`.

Cargo's manifest format is toml rather than the JSON you're used to with npm's package.json. Cargo uses the `Cargo.toml` file to know what dependencies to download, how to run tests, and how to build your projects (among other things).

Bootstrapping new projects

In node.js it's `npm init`. In Rust you have `cargo init` and `cargo new`

`cargo init` will initialize the current directory. `cargo new` initializes projects in a new directory.

Installing dependencies

In node.js it's `npm install [dep]`. In Rust you can use `cargo add [dep]` if you install `cargo-edit` first. Note: ***not*** cargo-add, just in case you come across it.

```
$ cargo install cargo-edit
```

This gives you four new commands: `add`, `rm`, `upgrade`, and `set-version`

Installing tools globally

In node.js it's `npm install --global`. In Rust you have `cargo install`.

Downloading, building, and placing executables in cargo's bin directory is handled with `cargo install`. If you installed rust via `rustup` then these are placed in a local user directory (usually `~/.cargo/bin`). You don't need to `sudo cargo install` anything.

Running tests

In node.js it's `npm test`. In Rust you have `cargo test`.

Cargo automates the running of unit tests, integration tests, and document tests through the `cargo test` command. There's a lot to Rust testing that we'll get to in later chapters.

Publishing modules

In node.js it's `npm publish`. In Rust you have `cargo publish`.

Easy peasy. You'll need to have an account on crates.io and set up the authentication details but cargo will help you there.

Running tasks

In node.js it's `npm run xxx`. In Rust, it depends... You have commands for common tasks but the rest is up to you.

In node.js you might use `npm run start` to run your server or executable. In Rust you would use `cargo run`. You can even use `cargo run --example xxx` to automatically run example code.

In node.js you might use `npm run benchmarks` to profile your code. In Rust you have `cargo bench`.

In node.js you might use `npm run build` to run webpack, tsc, or whatever. In Rust you have `cargo build`.

In node.js you might use `npm run clean` to remove temporary or generated files. In Rust you have `cargo clean` which will wipe away your build folder (`target`, by default).

In node.js you might use `npm run docs` to generate documentation. In Rust you have `cargo doc`.

For code generation or pre-build steps, cargo supports build scripts which run before the main build.

A lot of your use cases are covered by default, but for anything else you have to fend for yourself.

`npm`'s built-in task runner is one of the reasons why you rarely see `Makefile`s in JavaScript projects. In the Rust ecosystem, you're not as lucky. Makefiles are still common but `just` is an attractive option that is gaining adoption. It irons out a lot of the wonkiness of `Makefile`s while keeping a similar syntax.

Install `just` via

```
$ cargo install just
```

Other alternatives include `cargo-make` and `cargo-cmd`. I liked `cargo make` at first but its builtin tasks became just as annoying as `make`'s. I've become skilled writing `Makefile`s but I wish I spent that time learning `just` so take a lesson from me and start there. If you do go the `Makefile` route, check out isaacs's tutorial and read *Your makefiles are wrong*.

3.2.1. Workspaces & monorepos

Both package managers use a workspace concept to help you work with multiple small modules in a large project. In Rust, you create a `Cargo.toml` file in the root directory with a `[workspace]` entry that describes what's included and excluded in the workspace. It could be as simple as

```
[workspace]
members = [
  "crates/*"
]
```

Workspace members that depend on each other can then just point to the local directory as their dependency, e.g.

```
[dependencies]
other-project = { path = "../other-project" }
```

Check `cargo-workspaces` in the next section for a tool to help manage cargo workspaces.

3.2.2. Additional tools

`cargo-edit`

If you skimmed the above portion, make sure you don't miss out on `cargo-edit` which adds `cargo add` and `cargo rm` (among others) to help manage dependencies on the command line.

Install `cargo-edit` via

```
$ cargo install cargo-edit
```

`cargo-workspaces`

`cargo workspaces` (or `cargo ws`) simplifies creating and managing workspaces and their members. It was inspired by node's `lerna` and picks up where `cargo` leaves off. One of its most valuable features is automating the publish of a workspace's members, replacing local dependencies with the published versions.

Install `cargo-workspaces` via

```
$ cargo install cargo-workspaces
```

note: **workspaces** is plural. Don't install cargo-workspace expecting the same functionality.

`cargo-expand`

Macros in Rust are so common that 100% of the logic in your first Hello World app will be wrapped up into one. They're great at hand waving away code you don't want to write repeatedly but they can make code hard to follow and troubleshoot. `cargo expand` helps pull back the curtain.

`cargo-expand` needs a nightly toolchain installed which you can get by running

```
rustup install nightly
```

Install `cargo-expand` via

```
$ cargo install cargo-expand
```

Once installed, you can run `cargo expand [item]` to print out the fully generated source that rustc compiles.

NOTE	`cargo expand` takes a **named item**, not a file path. Running `cargo expand main` doesn't expand `src/main.rs`, it expands the `main()` function in your project's root. With a common layout, to expand a module found in a file like `src/some_module/another.rs`, you'd run `cargo expand some_module::another`. Don't worry, we'll go over the module system in a few days.

If you ran the `cargo new` command above to test it out, this is what your `src/main.rs` probably looks like.

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

`println!()` is a macro. Use `cargo expand` to see what code it generates.

```
$ cargo expand main
fn main() {
    {
        ::std::io::_print(::core::fmt::Arguments::new_v1(
            &["Hello, world!\n"],
            &match () {
                () => [],
            },
        ));
    };
}
```

`tomlq`

While not a `cargo xxx` command, it's useful for querying data in `.toml` files like `Cargo.toml`. It's a less featureful sibling to the amazing `jq`. It's not critical, but it's worth knowing about.

3.3. Wrap-up

The `npm` → `cargo` mapping is straightforward when you add `cargo-edit` and accept the lack of a standard task runner. In the next chapter we'll go over how to get your environment working with

Visual Studio Code: Chapter 3: Setting up VS Code.

4. Setting up Visual Studio Code

4.1. Introduction

Visual Studio Code dominated the JavaScript ecosystem almost on arrival. If you haven't yet given it a shot, you should. You won't find the breadth of plugins with Rust as you do with JavaScript, but it's growing rapidly. The most important pieces are there with features like:

- code completion/intellisense
- inline warnings
- debugger
- automatic refactor actions
- automatic documentation tooltips
- jump to definition, implementation, type, et al

4.2. Core language setup

There are two primary plugins, **rust** (rust-lang.rust) and **rust-analyzer** (matklad.rust-analyzer). They promise similar features but I could never get the **rust** plugin to work reliably. **Rust-analyzer** has been great from day one.

WARNING	The **rust** (rust-lang.rust) and **rust-analyzer** (matklad.rust-analyzer) plugins don't work well together. If you are exploring both, make sure you disable one to get a fair view of the other.

To install **rust-analyzer**, search for it in the extensions pane or press `Ctrl+Shift+P` then enter:

```
ext install matklad.rust-analyzer
```

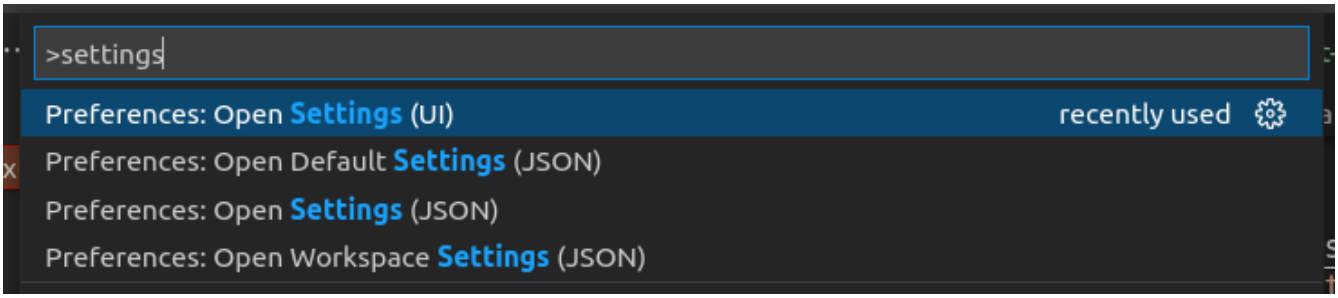
Once you're installed and rust-analyzer has downloaded everything it needs, you should be good to go. Note: you must be in a properly structured Rust project for rust-analyzer to work. You can't open just any `.rs` file and expect full IDE functionality, which brings us to…

4.2.1. If things don't feel like they're working…

If you create a new file in a Rust project, e.g. `my_module.rs`, you'll notice that VS Code & rust-analyzer do *something* but seem to be broken. They do complain about incorrect syntax, but they don't autocomplete anything and don't issue warnings about obviously incorrect code. That's because Rust projects ('crates') rely on a formal structure stemming from the root. That is to say, files can't get checked unless they are imported all the way down to a root source file (e.g. a `main.rs` or `lib.rs`). We'll get into the module system in a later chapter. You can alleviate this now by including your module via a line like `mod my_module`.

4.2.2. Notable rust-analyzer settings

You can edit your settings via a UI or as JSON by opening the command palette with `Ctrl+Shift+P`, typing `settings` and selecting which you want. See the VS Code documentation for more info.



The UI is a great way to explore settings, while JSON is more convenient for rapid editing and sharing. The settings below assume JSON.

Additional linting

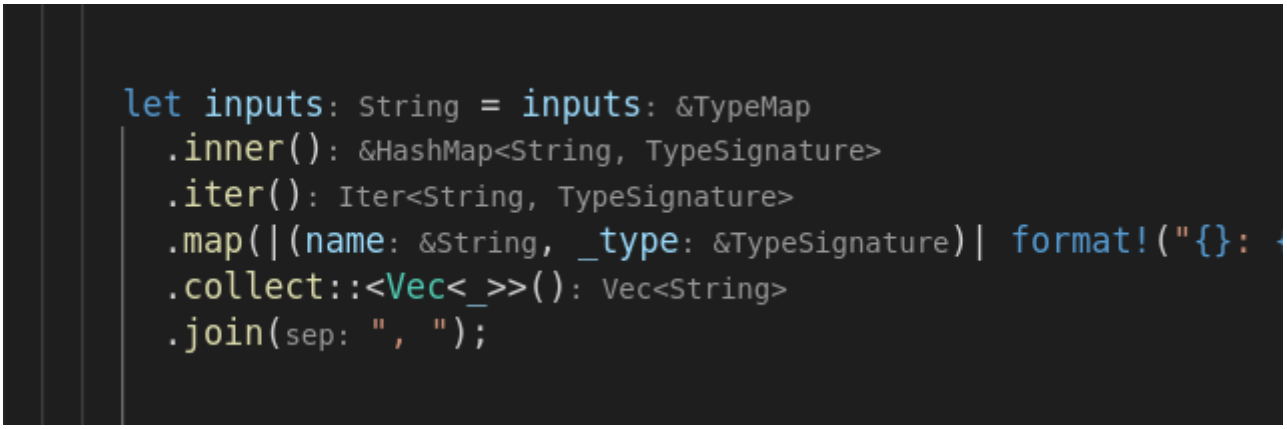
By default, rust-analyzer runs `cargo check` on save to gather project errors and warnings. `cargo check` essentially just compiles your project looking for errors. If you want more, then you're looking for `clippy`. Clippy is like the ESlint of the Rust universe. Get clippy via `rustup component add clippy` (you may notice you have it already).

You can run `cargo clippy` yourself or set rust-analyzer to run `clippy` on save to get loads of additional warnings and lints. Keep in mind that this takes additional resources and can be a bit slower, but it's worth it. I found `clippy` indispensible when learning Rust. It frequently highlights patterns that could be better replaced with more idiomatic or performant Rust.

```
{
  "rust-analyzer.checkOnSave.command": "clippy"
}
```

Inlay hints can be disabled

For me, rust-analyzer's inlay hints add too much noise. I turn them off.



It's a matter of preference. Just know that it's configurable if you look at your source and think "whoa, what is going on." The first setting here disables everything, but you can disable individual

hints with lines that follow.

```
{
  "rust-analyzer.inlayHints.enable": false,
  "rust-analyzer.inlayHints.chainingHints": false,
  "rust-analyzer.inlayHints.parameterHints": false
}
```

Prompt before downloading updates

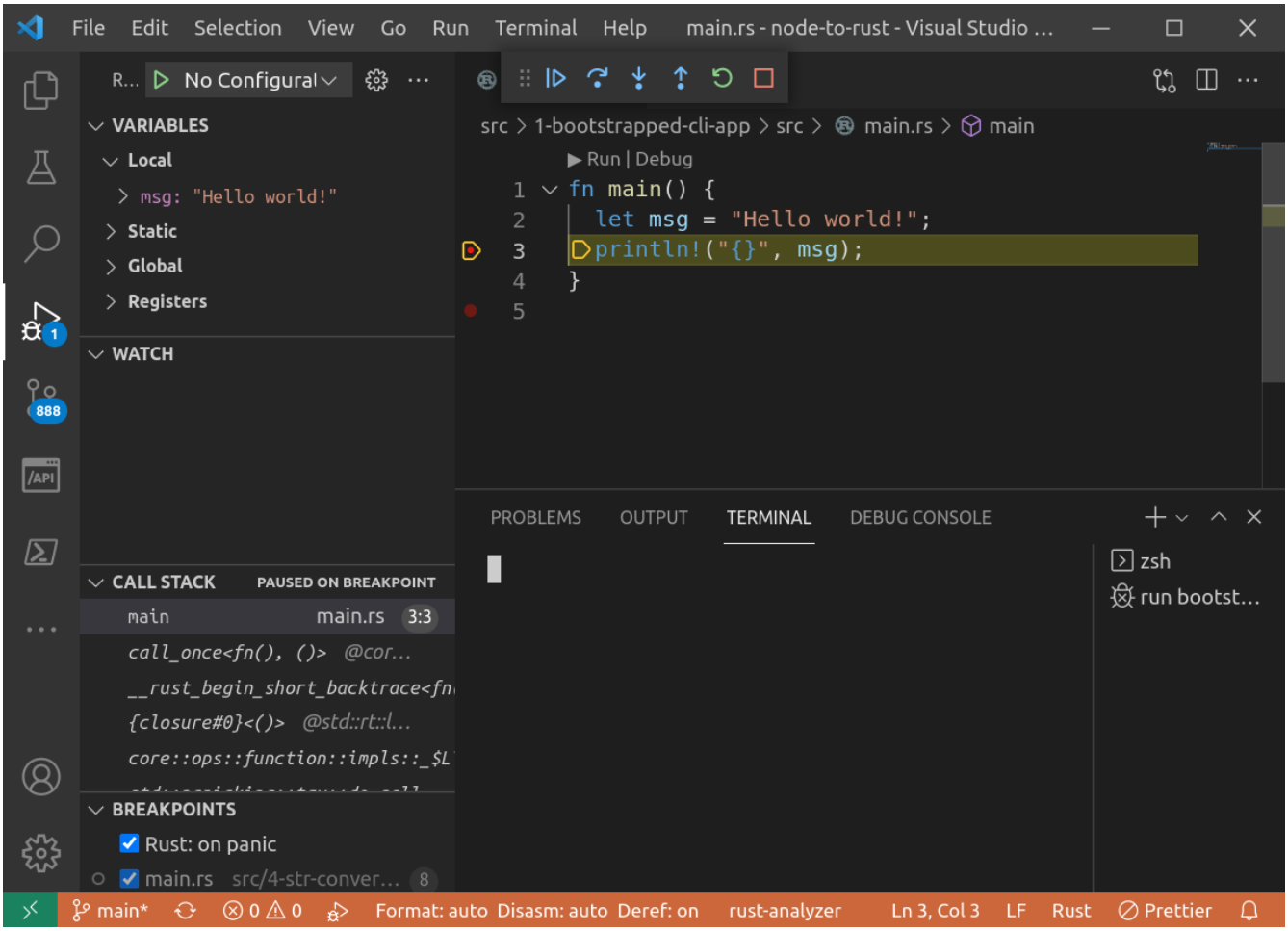
Rust-analyzer keeps itself up to date automatically, but I like controlling that behavior in my applications. Sometimes I'm on a flight's WiFi, a mobile hotspot, or some other shaky internet and it's nice to have the control. You can control this by changing the setting below.

```
{
  "rust-analyzer.updates.askBeforeDownload": true
}
```

4.3. Additional extensions

4.3.1. vscode-lldb

You'll need to install vadimcn.vscode-lldb before you can meaningfully debug Rust applications.



Install with `Ctrl+Shift+P` then:

```
ext install vadimcn.vscode-lldb
```

4.3.2. better-toml

bungcip.better-toml adds syntax highlighting for TOML files

Install with `Ctrl+Shift+P` then:

```
ext install bungcip.better-toml
```

4.3.3. crates

serayuzgur.crates shows you the latest versions of your dependencies and gives you quick access to update them.

image./images/blog/node-to-rust/vs-code-crates.gif[Inline crate versions]

Install with `Ctrl+Shift+P` then:

```
ext install serayuzgur.crates
```

4.3.4. search-crates-io

belfz.search-crates-io attempts to autocomplete dependencies as you write them in your Cargo.toml. I say "attempt" because it doesn't always work, though when it does I appreciate it.

Install with `Ctrl+Shift+P` then:

```
ext install belfz.search-crates-io
```

4.4. Wrap-up

Programming is more than text and an IDE should be more than a text editor. Your IDE should help you stay in flow as much as possible and put exactly what you need at your fingertips. If you have any other tips for how to make VS Code more Rust-friendly, please share them!

5. Hello World (and your first two WTFs)

5.1. Introduction

If you've never worked with a language that compiles down to native binaries, you're going to have fun with Rust. Yeah, it's easy to distribute executables within your chosen community, no matter if its node.js, Python, Ruby, PHP, Java, or something else. It gets crusty fast when you have to explain to outsiders. Think about the last time you enjoyed installing the latest version of Java to run a jar or had to deal with Python's virtual environments to run some random tool you wish was in JavaScript.

With Rust, you can distribute executable binaries like the big kids. Sure you may need to cross-compile it for other architectures and operating systems, but Rust has got you there, too.

5.1.1. Code

The code generated by the commands in this chapter can be found at vinodotdev/node-to-rust

5.1.2. Hello world

Start your first executable project by using `cargo new` with the name of your project like this

```
cargo new my-app
```

By default, `cargo new` uses a template for binary applications. That's what we want today, but keep in mind you can also do `cargo new --lib` to bootstrap a library.

After you execute the command, you'll have the following directory structure:

```
my-app/
 ├──── .git
 ├──── .gitignore
 ├──── Cargo.toml
 └──── src
     └──── main.rs
```

NOTE if you had a version of Rust installed before recently, the `edition` key in `Cargo.toml` is probably set to `2018`. `2021` has since gone stable and is what this book assumes. Editions are kind of like ECMAScript versions. The differences between `2018` and `2021` aren't huge, but it's worth calling out.

Even before you take a look at the source, run your new project with `cargo run`.

```
» cargo run
   Compiling my-app v0.1.0 (./my-app)
    Finished dev [unoptimized + debuginfo] target(s) in 0.89s
     Running `target/debug/my-app`
Hello, world!
```

`cargo run` builds your app with `cargo build` and executes the specified (or default, in this case) binary. After running this, your binary's already made and you can find it at `./target/debug/my-app`. Go, run it directly. It feels good.

If you want to build your app without running it, use `cargo build`. Cargo builds with the `dev` (debug) profile by default which is usually faster. It will retain debug information at the expense of file size and performance. When you are ready to release, you'd build with `cargo build --release` and you'd find your binary in `./target/release/my-app`.

5.1.3. Now to the Rust

Take a look at `src/main.rs` and mentally process this wild new language:

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

Well that's not so bad, right?

The `main()` function is required in standalone executables. It's the entrypoint to your CLI app.

`println!()` is a macro that generates code to print your arguments to STDOUT. If you've never dealt with macros before, they're like inline transpilers that generate code during compilation. We'll get to macros later.

`"Hello, world!"` is a string *slice* which is where things start getting real rusty. Strings are the first major hurdle for new Rust users and we'll tackle those in a later chapter, but let's walk through some of the first WTFs here to set the stage.

5.2. Strings WTF #1

First, assign "Hello, world!" to a variable using `let` and try to print it. Yep, Rust uses `let` and `const` keywords just like JavaScript, though where you want to use `const` just about everywhere in JavaScript, you want to use `let` in most Rust.

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

If you set up VS Code like we did in an earlier day, you'll already see an error. Run it anyway with `cargo run`.

```
$ cargo run
   Compiling day-4-strings-wtf-1 v0.0.0 (/path/node-to-rust/crates/day-4/strings-wtf-
1)
error: format argument must be a string literal
 --> crates/day-4/strings-wtf-1/src/main.rs:3:12
  |
3 |   println!(greeting);
  |            ^^^^^^^^
  |
help: you might be missing a string literal to format with
  |
3 |   println!("{}", greeting);
  |            +++++

error: could not compile `day-4-strings-wtf-1` due to previous error
```

If you expected this to work, you'd be normal. In most languages a string is a string is a string. Not in Rust. Do pay attention to the error message though. Rust's error messages are *leagues* beyond the error messages you're probably used to. This message not only describes the problem, but shows you exactly where it occurs ***AND*** tells you exactly what you need to do to fix it. `println!()` requires a string literal as the first argument and supports a formatting syntax for replacing portions with variables. Change your program to the following to get back on track.

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

5.3. Strings WTF #2

As a seasoned programmer, you know how write reusable code and are probably itching to abstract this complex logic into a reusable function. Take your newfound knowledge of `println!()` formatting syntax and write this beauty below.

```
fn main() {
  greet("World");
}

fn greet(target: String) {
  println!("Hello, {}", target);
}
```

Intuitively, this looks fine. But when you run it…

```
$ cargo run
   Compiling day-4-strings-wtf-2 v0.0.0 (/path/node-to-rust/crates/day-4/strings-wtf-
2)
error[E0308]: mismatched types
 --> crates/day-4/strings-wtf-2/src/main.rs:2:9
  |
2 |   greet("World");
  |         ^^^^^^^- help: try using a conversion method: .to_string()
  |         |
  |         expected struct `String`, found `&str`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `day-4-strings-wtf-2` due to previous error
```

While `rustc`'s error messages do hint at how to get you back up and running, it does little to explain WTF is really going on…

5.4. Wrap-up

Wrapping your head around strings in Rust is important. I know it's a tease to go through stuff like this without an immediate answer, but we'll get to it ASAP. First though, we need to talk about what "ownership" means in Rust in Chapter 5: Borrowing & Ownership.

These questions are why I started this book. Now would be a good time to start searching the web for answers on Rust strings so you have some perspective on things when you come back. If you need a starter, check these out

- Strings in the Rust docs
- Why Are There Two Types of Strings In Rust?
- How do I convert a &str to a String in Rust?
- Rust String vs str slices
- Rust: str vs String
- String vs &str in Rust

6. Borrowing & Ownership

6.1. Introduction

Before we get into strings, we need to talk about ownership. Ownership in Rust is where things start to get complicated. Not because it's hard to understand, but because Rust's rules force you to rethink logic and structure that would work fine elsewhere.

Rust gained popularity and respect because it promised memory safety without a garbage collector. Languages like JavaScript, Go, and many others use garbage collection to manage memory. They keep track of all references to objects (borrowed data) and only release memory when the reference count drops to zero. Garbage collectors make life easy for developers at the expense of resources and performance. Often times it's good enough. When it's not, you're usually out of luck. Troubleshooting and optimizing garbage collection is its own brand of dark magic. When you abide by Rust's rules, you'll achieve memory safety without the overhead of a garbage collector. You get all those resources back for free.

Memory safety is more than just making sure your programs don't crash. It closes the door to a whole class of security vulnerabilities. You've heard of SQL injection, right? If you haven't, it's a vulnerability that stems from database clients that create SQL statements by concatenating unsanitized user input. Adversaries exploit this vulnerability by passing cleverly crafted input that alters the final query and runs new instructions. Luckily, the attack surface is manageable and it's 100% preventable. Even still, it remains the most common vulnerability in web applications today. Memory unsafe code is like having harder to find SQL injection vulnerabilities that can pop up anywhere. Memory safety bugs account for the majority of serious vulnerabilities. Eliminating them altogether with *no* performance impact is an attractive notion.

6.1.1. Required reading

This guide won't duplicate existing content when possible. It's meant to clarify concepts that you have already encountered. Check out these chapters in the Rust book if you're skimming here and aren't following along.

- Rust book Ch.3: Common Programming Concepts
- Rust book Ch.4: Understanding Ownership
- Rust by Example: Variable Bindings
- Rust by Example: Primitives
- Rust by Example: Flow control
- Rust by Example: Functions

6.1.2. Quick sidebar

Variable assignments & mutability

JavaScript assignments fall into two camps, `let` for variables that can be reassigned and `const` for those that can't. While Rust also has `let` and `const`, ignore `const` for now.

Where you want `const` in JavaScript, you want `let` in Rust. Where you'd use `let` in JavaScript, you'd use `let mut` in Rust. The keyword `mut` is required to declare a variable as mutable (changeable). That's right, everything in Rust is immutable (unchangeable) by default. This is a good thing, I promise you. You'll wish this was true in JavaScript by the time you're done here.

In JavaScript you'd write:

```
let one = 1;
console.log({ one });
one = 3;
console.log({ one });
```

The Rust counterpart is:

```
fn main() {
  let mut mutable = 1;
  println!("{}", mutable);
  mutable = 3;
  println!("{}", mutable);
}
```

One major difference with Rust is that you can only reassign a variable with a value of the same type. This won't work:

```
fn main() {
    let mut mutable = 1;
    println!("{}", mutable);

    mutable = "3"; // Notice this isn't a number.

    println!("{}", mutable);
}
```

That said, you *can* assign a different type to a variable with the same name by using another `let` statement

```
fn main() {
    let myvar = 1;
    println!("{}", myvar);
    let myvar = "3";
    println!("{}", myvar);
}
```

6.1.3. Rust's Borrow Checker

Rust guarantees memory safety by enforcing some basic — albeit strict — rules for how you pass data around, how you "borrow" data and who "owns" data.

Rule #1: Ownership

When you pass a value, the calling code can no longer access that data. It's given up ownership. Take a look at the code below and the error that occurs when you try to run it

```
use std::{collections::HashMap, fs::read_to_string};

fn main() {
    let source = read_to_string("./README.md").unwrap();
    let mut files = HashMap::new();
    files.insert("README", source);
    files.insert("README2", source);
}
```

IMPORTANT	You'll see `.unwrap()` a lot in example code but it's not something you should use frequently in production code. We'll go over it in the Result & Option section but the gist is: `.unwrap()` assumes a successful operation and panics (dies) otherwise. It's OK in examples. It's not OK in your applications unless you are sure an operation can't fail.

In your IDE or when you try to run this, notice the error message `use of moved value: source`. You'll see that a lot and it's important to embed its meaning in your brain now.

```
error[E0382]: use of moved value: `source`
  |
4 |     let source = read_to_string("./README.md").unwrap();
  |         ------ move occurs because `source` has type `String`, which does not
implement the `Copy` trait
5 |     let mut files = HashMap::new();
6 |     files.insert("README", source);
  |                            ------ value moved here
7 |     files.insert("README2", source);
  |                             ^^^^^^ value used here after move

For more information about this error, try `rustc --explain E0382`.
```

When we inserted `source` into the HashMap the first time, we gave up ownership. If we want to make the above code compile, we have to clone `source` the first time we give it away.

```
use std::{collections::HashMap, fs::read_to_string};

fn main() {
    let source = read_to_string("./README.md").unwrap();
    let mut files = HashMap::new();
    files.insert("README", source.clone());
    files.insert("README2", source);
}
```

NOTE

You'll see notes in these error messages when your value "does not implement the `Copy` trait". We'll get to traits later but the gist of `Copy` vs `Clone` is that `Copy` is for data that can be reliably, trivially copied. Rust will copy those values automatically for you. `Clone` is for potentially expensive copies and you have to do that yourself.

Rule #2: Borrowing

When borrowing data — when you take a reference to data — you can do it immutably an infinite number of times or mutably *only once*. Typically, you'll take a reference by prefixing a value with an ampersand (`&`). This gives you the ability to pass potentially large chunks of data around without cloning them every time.

```
use std::{collections::HashMap, fs::read_to_string};

fn main() {
    let source = read_to_string("./README.md").unwrap();
    let mut files = HashMap::new();
    files.insert("README", source.clone());
    files.insert("README2", source);

    let files_ref = &files;
    let files_ref2 = &files;

    print_borrowed_map(files_ref);
    print_borrowed_map(files_ref2);
}

fn print_borrowed_map(map: &HashMap<&str, String>) {
    println!("{:?}", map)
}
```

NOTE

The `{:?}` syntax in `println!` is the `Debug` formatter. It's a handy way of outputting data that doesn't necessarily have a human-readable format.

If we needed to take a mutable reference of our map, we would write it as `let files_ref = &mut files;`.

```
use std::{collections::HashMap, fs::read_to_string};

fn main() {
    let source = read_to_string("./README.md").unwrap();
    let mut files = HashMap::new();
    files.insert("README", source.clone());
    files.insert("README2", source);

    let files_ref = &mut files;
    let files_ref2 = &mut files;

    needs_mutable_ref(files_ref);
    needs_mutable_ref(files_ref2);
}

fn needs_mutable_ref(map: &mut HashMap<&str, String>) {}
```

You'll encounter the following error when you compile the above code.

```
error[E0499]: cannot borrow `files` as mutable more than once at a time
   |
9  |     let files_ref = &mut files;
   |                     ---------- first mutable borrow occurs here
10 |     let files_ref2 = &mut files;
   |                      ^^^^^^^^^^ second mutable borrow occurs here
11 |
12 |     needs_mutable_ref(files_ref);
   |                       --------- first borrow later used here

For more information about this error, try `rustc --explain E0499`.
```

The Rust compiler is smart and getting smarter every release, though. If you reorder your borrows so that it can see that one reference will be finished before you use the other, you'll be OK.

```
use std::{collections::HashMap, fs::read_to_string};

fn main() {
    let source = read_to_string("./README.md").unwrap();
    let mut files = HashMap::new();
    files.insert("README", source.clone());
    files.insert("README2", source);

    let files_ref = &mut files;

    needs_mutable_ref(files_ref);

    let files_ref2 = &mut files;

    needs_mutable_ref(files_ref2);
}

fn needs_mutable_ref(map: &mut HashMap<&str, String>) {}
```

As you're starting with Rust, you may find many of your errors can be solved by just switching around around the order of your code. Give it a shot before ripping your hair out.

6.1.4. References support session

If you've spent most of your life in JavaScript or had horrible experiences with languages like C, you may be thinking: "References? Whatever. I don't like references and I don't need references." I need to let you in on a secret. You use references literally *all the time* in JavaScript. Every object is a reference. That's how you can pass an object to a function, edit a property, and have that change be reflected after the function finishes. Take this code for example

```
function actOnString(string) {
  string += " What a nice day.";
  console.log(`String in function: ${string}`);
}

const stringValue = "Hello!";
console.log(`String before function: ${stringValue}`);
actOnString(stringValue);
console.log(`String after function: ${stringValue}\n`);

function actOnNumber(number) {
  number++;
  console.log(`Number in function: ${number}`);
}

const numberValue = 2000;
console.log(`Number before function: ${numberValue}`);
actOnNumber(numberValue);
console.log(`Number after function: ${numberValue}\n`);

function actOnObject(object) {
  object.firstName = "Samuel";
  object.lastName = "Clemens";
  console.log(`Object in function: ${objectValue}`);
}

const objectValue = {
  firstName: "Jane",
  lastName: "Doe",
};
objectValue.toString = function () {
  return `${this.firstName} ${this.lastName}`;
};
console.log(`Object before function: ${objectValue}`);
actOnObject(objectValue);
console.log(`Object after function: ${objectValue}`);
```

When you run it you get:

```
String before function: Hello!
String in function: Hello! What a nice day.
String after function: Hello!

Number before function: 2000
Number in function: 2001
Number after function: 2000

Object before function: Jane Doe
Object in function: Samuel Clemens
Object after function: Samuel Clemens
```

Not using references would be like making a deep copy of every `Object` every time you pass it to any function. That would be ridiculous, right? Of course it would.

NOTE

Programmers coming *to* JavaScript look at this behavior as their own "WTF." They're the type of people who interview candidates with questions like "Is JavaScript a pass by value or pass by reference language" while JavaScript programmers hear that question and think "Why are you talking about references and not asking me about React?"

Interview tip: the answer is "JavaScript is pass by value, except for all `Object`s where the value is a reference."

6.2. Wrap-up

Ownership is a core, recurring topic in Rust. We needed to dive into it at a high level before we deal with Strings Chapter 6: Strings, part 1.

7. Strings, Part 1

7.1. Introduction

The first hurdle with Rust and strings comes from misaligned expectations. A string literal (`"Hi!"`) isn't an instance of a `String` in Rust. You don't need to fully understand the code below yet, just know that it outputs the types of the values sent to `print_type_of`.

```
fn main() {
  print_type_of(&"Hi!");
  print_type_of(&String::new());
}

fn print_type_of<T>(_: &T) {
  println!("Type is: {}", std::any::type_name::<T>())
}
```

```
$ cargo run
Type is: &str
Type is: alloc::string::String
```

Fun fact: JavaScript string literals aren't JavaScript `String`s either.

```
"Hi!" === "Hi!";
// > true

"Hi!" === new String("Hi!");
// > false
```

Wait, there's more.

```
typeof "Hi!";
// > "string"

typeof new String("Hi!");
// > "object"

typeof String("Hi!");
// > "string"

"Hi!" === String("Hi!");
// > true

String("Hi!") === new String("Hi!");
// > false
```

That last part is just to point out that if you can learn to love JavaScript, you can learn to love Rust.

JavaScript hand waves away the difference between `string` primitives and `String` instances. It automatically does what you want, when you want it, without incurring the overhead of creating an `Object` for every `string`. When you call a method on a primitive `string`, JavaScript interpreters magically translate it to a method on the `String` prototype.

Rust has similar magic, it just doesn't always do it for you.

7.2. Additional links

There is a *lot* written about Strings. Don't miss the official docs and other great posts out there.

- Strings in the Rust docs
- Why Are There Two Types of Strings In Rust?
- How do I convert a &str to a String in Rust?
- Rust String vs str slices
- Rust: str vs String
- String vs &str in Rust

7.3. Rust strings in a nutshell

7.3.1. `&str`

String literals are *borrowed string slices*. That is to say: they are pointers to a substring in other string data. The Rust compiler puts all of our literal strings in a bucket somewhere and replaces the values with pointers. This lets Rust optimize away duplicate strings and is why you have a pointer to a string **slice**, vs a pointer to a single `String`.

You can verify the optimizations are real, if you don't believe me. Copy-paste the print line below a gazillion times (or less) and see that it only has a minor impact on the executable size.

```
fn main() {
  print("TESTING:12345678901234567890123456789012345678901234567890");
}

fn print(msg: &str) {
  println!("{}", msg);
}
```

You can also run the (not-rust-specific) `strings` command to output all the string data in a binary.

```
$ strings target/release/200-prints | grep TESTING
TESTING:12345678901234567890123456789012345678901234567890
```

If you run that command on the `200-unique-prints` binary in the node-to-rust repo, you'll get much more output.

7.3.2. `String`

`String`s are the strings that you know and love. You can change them, cut them up, shrink them, expand them, all sorts of great stuff. All that brings along additional cost though. Maybe you don't care, maybe you do. It's in your hands now.

7.3.3. How do you make a `&str` a `String`?

In short: use the `.to_owned()` method on a `&str` (a "borrowed" string slice) to turn it into an "owned" `String`, e.g.

```
let my_real_string = "string literal!".to_owned();
```

For what its worth, this method calls the code below under the hood.

```
String::from_utf8_unchecked(self.as_bytes().to_owned())
```

> **NOTE** `self` is Rust's `this`.

This is why we had to go over ownership before we got into strings. String literals start off borrowed. If you need an owned `String`, you have to convert it (copy it, essentially).

7.3.4. You're telling me I need to write `.to_owned()` everywhere?

Yes. And no. Sort of. For now, accept "yes" until we get into Traits and generics.

7.3.5. What about `.to_string()`, `.into()`, `String::from()`, or `format!()`?

All these options also turn a `&str` into a `String`. If this is your first foray into Rust from node.js, don't worry about this section. This is for developers who have read all the other opinions out there and are wondering why other methods aren't the "one true way."

NOTE	A Rust `trait` is sharable behavior. We haven't gotten to them yet, but think of a trait like a `mixin` if you've ever used the mixin pattern in JavaScript.

Why not `.to_string()`?

```
fn main() {
  let real_string: String = "string literal".to_string();

needs_a_string("string literal".to_string());
}

fn needs_a_string(argument: String) {}
```

`something.to_string()` converts *something* into a string. It's commonly implemeted as part of the `Display` trait. You'll see a lot of posts that recommend `.to_string()` and a lot that don't.

The nuances in the recommendation stem from how much you want the compiler to help you. As your applications grow — especially when you start to deal with generics — you'll inevitably refactor some types into other types. A value that was initially a `&str` might end up being refactored into something else. If the new value still implements `Display`, then it has a `.to_string()` method. The compiler won't complain.t

In contrast, `.to_owned()` turns something borrowed into something owned, often by cloning. Turning a borrowed `*not-string*` into an owned `*not-string*` gives the compiler the context necessary to raise an error. If you're OK with the difference, it's easy to change a `.to_owned()` into a `.to_string()`. If you weren't expecting it, then you highlighted an issue before it became a problem.

If you use `.to_string()`, the world won't explode. If you are telling someone they shouldn't use `.to_string()`, you have to be able to explain why. Just like you would if you used the word [octopodes](https://www.dailymotion.com/video/x2voh0q).

NOTE	Clippy has a lint that will alert you if you use `.to_string()` on a `&str`: [clippy::str_to_string](https://rust-lang.github.io/rust-clippy/master/#str_to_string)

7.3.6. Why not `something.into()`?

For example:

```
fn main() {
  let real_string: String = "string literal".into();

  needs_a_string("string literal".into());
}

fn needs_a_string(argument: String) {}
```

`something.into()` will (attempt) to turn *something* into a destination type by calling `[dest_type]::from()`, e.g. `String::from(something)`. If the destination type is a `String` and your *something* is a `&str` then you'll get the behavior you're looking for. The concerns are similar to those above. Are you really trying to turn *something* into *something else*, or are you trying to turn a `&str` into a `String`? If it's the former, then `.into()` works fine, if it's the latter then there are better ways to do it.

Why not String::from()?

```
fn main() {
  let real_string: String = String::from("string literal");

needs_a_string(String::from("string literal"));
}

fn needs_a_string(argument: String) {}
```

`String::from(something)` is more specific than `.into()`. You are explicitly stating your destination type, but it has the same issues as `.to_string()`. All it expresses is that you want a string but you don't care from where.

7.3.7. Why not `format!()`?

```
fn main() {
  let real_string: String = format!("string literal");

  needs_a_string(format!("string literal"));
}

fn needs_a_string(argument: String) {}
```

`format!()` is for formatting. This is the only one you should ***definitely*** not use for simply creating a `String`.

NOTE	Clippy also has a lint for this one: clippy::useless_format

7.3.8. Implementation details

The path to this "one, true answer" is mapped out here. At the end of the road, everything points to `.to_owned()`.

`.to_owned()`

Implemented here

Calls `String::from_utf8_unchecked(self.as_bytes().to_owned())`

`String::from()`

Implemented here

Calls `.to_owned()`

`.to_string()`

Implemented here

Calls `String::from()`

`.into()`

Implemented here

Calls `String::from()`

`format!()`

Implemented here[here]

Calls `Display::fmt` for `str` here

7.4. Wrap-up

Turning `&str` into `String` is the first half of the string issue. The second is which to use in function arguments when you want to create an easy-to-use API that takes either string literals (`&str`) or `String` instances.

8. Language Part 1: Syntax & Differences

8.1. Introduction

All languages have a productivity baseline. That point where you know enough to be confident. After you reach the baseline, mastery is a matter of learning best practices, remembering what's in standard libraries, and expanding your bag of tricks with experience.

Python has a low productivity baseline. The language is easy to grasp and it's a popular one to learn because of it. JavaScript's baseline is a little higher because of the async hurdle. Typed languages start higher by default due to their additional context.

Rust's productivity baseline is more of a productivity roller coaster. Once you think you've figured it out, you peel back the curtain and realize you actually know nothing.

We're still well below the first baseline but congrats to you for getting this far. This section will fill in some of the blanks and skirt you passed some hair pulling episodes where you scream things like "I just want to make an array!!@!"

8.1.1. Notable differences in Rust

Rust programming style

The Rust style differs from JavaScript only slightly.

Variables, functions, and modules are in snake case (e.g. `time_in_millis`) vs camel case (e.g. `timeInMillis`) as in JavaScript.

Structs (a cross between JavaScript objects and classes) are in Pascal case (e.g. `CpuModel`) just like similar structures would be in JavaScript.

Constants are similarly in capital snake case (e.g. `GLOBAL_TIMEOUT`).

Unambiguous parantheses are optional

```
if (x > y) { /* */ }

while (x > y) { /* */ }
```

Can be written as

```
if x > y { /* */ }

while x > y { /* */ }
```

This style is preferred and linters will warn you of it.

Almost everything is an expression

Almost everything complete chunk of code returns a value. Obviously `4 * 2` returns a value (`8`), but so does `if true { 1 } else { 0 }` which returns `1`.

That means you can assign the result of blocks of code to variables or use them as return values, e.g.

```
fn main() {
    let apples = 6;
    let message = if apples > 10 {
        "Lots of apples"
    } else if apples > 4 {
        "A few apples"
    } else {
        "Not many apples at all"
    };

    println!("{}", message) // prints "A few apples"
}
```

Notice how the lines with the strings don't end in a semi-colon. What happens if you add one? What happens if you add a semi-colon to all three?

Spoiler alert: both questions lead to code that won't compile for different reasons. They produce error messages you'll come across frequently. Don't rob yourself the joy of seeing them first hand. It's *exhilirating*. Or just read on.

The unit type (`()`)

Rust has no concept of `null` or `undefined` like JavaScript. That sounds great but it's not like those existed for no reason. They mean something. They mean *nothing*, albeit different kinds of nothing. As such, Rust still needs types that can represent *nothing*.

Try adding a semi-colon the first string above so the `if {} else if {} else {}` looks like this:

```
let message = if apples > 10 {
    "Lots of apples";  // 🤔 Notice the rogue semi-colon
} else if apples > 4 {
    "A few apples"
} else {
    "Not many apples at all"
};
```

Rust won't compile, giving you the error "`if` and `else` have incompatible types." The full output is below.

```
error[E0308]: `if` and `else` have incompatible types
  --> crates/day-7/syntax/src/main.rs:13:12
   |
11 |         let message = if apples > 10 {
   |   ____________________-
12 |  |          "Lots of apples";
   |  |          -----------------
   |  |          |                |
   |  |          |                help: consider removing this semicolon
   |  |          expected because of this
13 |  |      } else if apples > 4 {
   |  |_____________^
14 | ||          "A few apples"
15 | ||      } else {
16 | ||          "Not many apples at all"
17 | ||      };
   | ||      ^
   | ||_____|
   | |______`if` and `else` have incompatible types
   |         expected `()`, found `&str`
```

The helper text tells you that rust "expected `()`, found `&str`." It also mentions (helpfully) that you might consider removing the semicolon. That'll work, but what's going on and what is `()`?

`()` is called the unit type. It essentially means "no value." An expression that ends with a semi-colon returns *no value*, or `()`. Rust sees that the `if {}` part of the conditional returns nothing — or `()` — and expects every other part of the conditional to return a value of the same type, or `()`. When we leave off the semi-colon, the result of that first block is the return value of the expression `"Lots of apples"` which is (naturally) `"Lots of apples"`.

This brings us to...

Implicit returns

We saw how a block can return a value above. Functions are no different. The last line of execution (the "tail") will be used as the return value for a function. You'll frequently see functions that don't have explicit `return` statements, e.g.

```
fn add_numbers(left: i64, right: i64) -> i64 {
    left + right  // 👈 Notice no semi-colon
}
```

Which is equivalent to:

```
fn add_numbers(left: i64, right: i64) -> i64 {
    return left + right;
}
```

If you specify a return type (the `-> i64` above) and accidentally use a semi-colon on your last line, you'll see an error like we saw in the section above:

```
  |
5 | fn add_numbers(left: i64, right: i64) -> i64 {
  |    -----------                           ^^^ expected `i64`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
6 |     left + right;
  |                 - help: consider removing this semicolon
```

It will take some getting used to, but you do get used to it. Whenever you see an error complaining about `()`, it's often because you either need to add or remove a semi-colon (or return type) somewhere.

Arrays

Rust has arrays. If you use them like you want to however, you're going to have an experience just like you did with strings. I won't go into arrays and slices because there is plenty written on the subject (e.g. Rust Book: Ch 4.3 and Rust by Example: Ch 2.3).

The short of Rust arrays is: they must have a known length with all elements initialized.

This won't work.

```
let mut numbers = [1, 2, 3, 4, 5];
numbers.push(7);
println!("{:?}", numbers);
```

The reason it's not worth going into is because you're probably not looking for arrays.

What you're looking for is `Vec` or `VecDeque`. `Vec` is to JavaScript arrays what `String` is to JavaScript strings. `Vec`'s can grow and shrink at the end. `VecDeque` can grow or shrink from either direction.

> **NOTE** `VecDeque` is pronounced vec-deck. Deque stands for "Double ended queue."

Arrays and iterators will have their own section in this guide, but know that there's an easy-to-use macro that gives you a `Vec` with similar syntax you're used to.

```
let mut numbers = vec![1, 2, 3, 4, 5];  // 👀 Notice the vec! macro
numbers.push(7);
println!("{:?}", numbers);
```

8.2. Wrap-up

There is no end to what can trip you up when you try and jump into another language. If you

haven't read The Rust Book, you are going to start having trouble if you haven't already. If you *have* read The Rust Book, read it again. Every time you turn a corner in Rust, you'll start to see things more clearly. Documentation will look different. What didn't land right the first time will start to make sense now.

Next up we'll dive into the basic types and start on Structs. Stay tuned!

9. From objects and classes to HashMaps and structs

9.1. Introduction

In the last chapter we went over some basic differences between JavaScript and Rust and finished with Vectors, the Rust counterpart to JavaScript's arrays. Arrays are a core structure in JavaScript but pale in comparison to the almighty `Object`. The JavaScript `Object` is a *beast*. It takes a hundred concepts and wraps them into one. It could be a map, a dictionary, a tree, a base class, an instance, a bucket for utility functions, and even a serialization format. The next couple sections will unpack the typical use cases of the JavaScript `Object` and translate them to Rust.

NOTE	As we move forward, this guide will start to use more TypeScript than JavaScript. You'll need to have `ts-node` installed (`npm install -g ts-node`) to run the examples. If you want a TypeScript playground, check out my boilerplate project at jsoverson/typescript-boilerplate.

9.1.1. Maps vs Objects

Before ECMAScript 6, JavaScript didn't even have `Map`. It was `Object`s all the way down. That led a whole generation down the path of treating `Object`s as `Map`s and persists today. That's not necessarily a bad thing, but the theme of this guide is: you don't get to magic away the details anymore.

First we need to clarify the difference between a JavaScript `Map` and an `Object`.

A JavaScript `Map` is essentially a key/value store. You store a value under a key (be it a `string` or anything at all) and retrieve that value with the same key.

A JavaScript object is a data structure that has properties (a.k.a. keys) which hold values. You set values via a property (a.k.a key) and retrieve values the same way. While not a term used in JavaScript, an object is a "dictionary." Dictionaries are described as:

NOTE	*A dictionary is also called a hash, a map, a hashmap in different programming languages (and an Object in JavaScript). They're all the same thing: a key-value store.*

I'm glad I could clear things up for you. I'm kidding, but it's to prove a point so bear with me. In JavaScript, the reason to choose a certain type isn't always clear. You can use `Map` and `Object` interchangeably for many purposes. It's not like that in Rust. We need to separate our use cases before moving on. In short:

When you want a keyed collection of values that all have the same type, you want a `Map` type.

When you want an object that has a known set of properties, you want a more structured data type.

A "Map" is a concept and languages usually have many implementations. We'll talk about the `HashMap` type below. The structured data use case usually falls under the category of a language

feature. In Rust's case it's called a `struct`.

9.2. From `Map` to `HashMap`

To store arbitrary values by key, we're going to want a `HashMap`. While there are alternatives, don't worry about them yet.

This is how you'd create a `Map` in TypeScript. JavaScript would be identical minus the `<string, string>`.

```
const map = new Map<string, string>();

map.set("key1", "value1");
map.set("key2", "value2");

console.log(map.get("key1"));
console.log(map.get("key2"));
```

In Rust, you would write:

```
use std::collections::HashMap;

fn main() {
  let mut map = HashMap::new();
  map.insert("key1", "value1");
  map.insert("key2", "value2");

  println!("{:?}", map.get("key1"));
  println!("{:?}", map.get("key2"));
}
```

This looks nearly identical but I'd be dishonest if I moved on quickly. When you run the Rust code, the output is:

```
Some("value1")
Some("value2")
```

Which is *kind of* what we wanted.

9.2.1. `Some()`, `None`, and `Option`

Take a look at that `Some()` craziness. What in the world was that about? `Some` is a variant of the `Option` enum. `Option`s are another way of representing *nothing* like we talked in Chapter 7: Language Part 1: Syntax & Differences.

NOTE An enum (short for enumeration) is a bound list of possible values, or variants. JavaScript doesn't have them, but TypeScript does. Rust enums are cooler, though.

We'll get to enums in time, but think of `Option` as a value that can hold either something or nothing. If we pass a key that doesn't exist in our map, `get()` needs to return *nothing* but we don't have `undefined` in Rust. We could return the unit type (`()`) but we can't write a function that returns `string | undefined` like we could in TypeScript. Instead, Rust has enums. That's where `Option` comes in. The `Option` enum has two variants, it's either `Some()` or `None`.

NOTE You can test an `Option` with `.is_some()`, or `.is_none()`. You can "unwrap" it with `.unwrap()` which will panic if it's `None`. You can unwrap it safely with `.unwrap_or(default_value)`. See the Rust docs on Option for more.

We can rewrite the above to clean up the output.

```
use std::collections::HashMap;

fn main() {
  let mut map = HashMap::new();
  map.insert("key1", "value1");
  map.insert("key2", "value2");

  println!("{}", map.get("key1").unwrap_or(&""));
  println!("{}", map.get("key2").unwrap_or(&""));
}
```

NOTE We *know* that our map contains values for the keys we specify, so we could have used `.unwrap()` without worrying. If we did however, we wouldn't be able to use it for the example below. Such is the life of example code.

Notice how we're using `.unwrap_or(&"")` above, instead of `.unwrap_or("")`. Why? What happens if we write it that way?

```
error[E0308]: mismatched types
 --> crates/day-8/maps/src/main.rs:9:44
  |
9 |   println!("{}", map.get("key2").unwrap_or(""));
  |                                            ^^ expected `&str`, found `str`
  |
  = note: expected reference `&&str`
             found reference `&'static str`

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

These types of errors can be very confusing. The helper text says rust expected a `&str` but found a `str` and then proceeds to note that it actually expected a `&&str` yet found `&'static str`. I already told

you that string literals are the type `&str`, not `str` and never mentioned anything about `'static`. What gives?

Let's break it down.

- First, note that we used string literals for both our HashMap's keys and values. Rust inferred the HashMap's type to be `HashMap<&str, &str>`.
- Second, `.get()` doesn't return an owned value, it returns a borrowed value. That makes sense, right? If it returned an owned value it would either need to give up its ownership (which would mean removing the value from the map) or it would need to clone it. Cloning means extra cycles and memory which is something Rust will *never* do for you automatically. So you get a reference to your value, which was already a reference. A reference to a `&str` has a type of `&&str`.
- Third, `.unwrap_or()` needs to produce the exact same type as the `Option`'s type. In this case, the option's type is `Option<&&str>`. That is to say, the `Option` can either be a `Some(&&str)` (the return type of `.get()`) or `None`. So we need our `.unwrap_or()` to return a `&&str` which means we need to pass it a `&&str`, or `&""`.
- Finally, We haven't talked about lifetimes yet but the `'static` is a lifetime. It means that a reference points to data that will last as long as the program does. String literals will last forever (they have a `static` lifetime) because Rust ensures it. Don't worry about it yet, just know that a `&'static str` means that Rust is probably talking about a string literal.

NOTE	So what's that helper text talking about then? I don't know. It looks wrong. I hadn't thought about it much until you asked. You ask great questions.

9.3. From objects and classes to `structs`

Rust's `structs` are as ubiquitous as JavaScript's objects. They are a cross between plain old objects, TypeScript interfaces, and JavaScript classes. While you frequently use a Rust `struct` with methods (e.g. `some_object.to_string()`) which make them feel like normal class instances, it's more helpful to think of `structs` as pure data to start. Behavior comes later.

An interface you could write as TypeScript like...

```
interface TrafficLight {
  color: string;
}
```

...would be written as a `struct` in Rust like this.

```
struct TrafficLight {
  color: String,
}
```

Instantiating is similar, too:

```
const light: TrafficLight = {
  color: "red",
};
```

```
let light = TrafficLight {
  color: "red".to_owned(), // Note we want an owned String
};
```

But you probably wouldn't write an interface for this in TypeScript. You'd write a class so it can be instantiated with defaults and have methods, right? Something like:

```
class TrafficLight {
  color: string;

  constructor() {
    this.color = "red";
  }
}

const light = new TrafficLight();
```

To do this in Rust, you'd add an implementation to your `struct`.

9.3.1. Adding behavior

To add behavior we add an `impl`.

```
struct TrafficLight {
  color: String,
}

impl TrafficLight {
  pub fn new() -> Self {
    Self {
      color: "red".to_owned(),
    }
  }
}
```

This adds a public function called `new()` that you can execute to get a new `TrafficLight`. `Self` refers to `TrafficLight` here and you could replace one with the other with no change in behavior. There's nothing special about `new` or how you call it. It's not a keyword like in `JavaScript`. It's convention. Call it via `TrafficLight::new()`, e.g.

```
fn main() {
  let light = TrafficLight::new();
}
```

This works but we can't really verify it. You could try printing it but — spoiler alert: it won't compile. You can't even use the debug syntax I mentioned in an earlier chapter.

```
fn main() {
  let light = TrafficLight::new();
  println!("{}", light);
  println!("{:?}", light);
}
```

Both the display formatter (used by `{}`) and the debug formatter (used by `{:?}`) rely on traits that we don't implement.

Traits are like mixins in JavaScript. They are another way of attaching behavior to data. Traits are a big topic that deserve a whole section, but we can add some simple ones today.

```
impl std::fmt::Display for TrafficLight {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    write!(f, "Traffic light is {}", self.color)
  }
}
```

This implements a trait (`Display` found at `std::fmt::Display`) for our `TrafficLight`. Now we can print our traffic light via `println!()`!

```
Traffic light is red
```

Traits can also have default, derivable implementations. This allows you to generalize behavior and reduce boilerplate. If all the fields in your `struct` implement the `Debug` trait, you can derive it with a single line (`#[derive(Debug)]`) and gain debug output for free.

```
#[derive(Debug)]
struct TrafficLight {
  color: String,
}
```

The full source now looks like this:

```
fn main() {
  let light = TrafficLight::new();
  println!("{}", light);
  println!("{:?}", light);
}

impl std::fmt::Display for TrafficLight {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    write!(f, "Traffic light is {}", self.color)
  }
}

#[derive(Debug)]
struct TrafficLight {
  color: String,
}

impl TrafficLight {
  pub fn new() -> Self {
    Self {
      color: "red".to_owned(),
    }
  }
}
```

When you run it, you'll see both our display line and the debug line printed to STDOUT.

```
[snipped]
Traffic light is red
TrafficLight { color: "red" }
```

9.4. Wrap-up

HashMaps are the key to storing and accessing data with a key/value relationship. We'll touch on them more in an upcoming section on Arrays and Iterators.

Structs are how you bring some of the class behavior to Rust. Simple usage of JavaScript and TypeScript classes is easily portable. Tightly coupled relationships and object-oriented patterns aren't. It'll take some time to get used to traits but the benefits of how you structure your code and logic will transfer back to JavaScript.

Traits are powerful and are what give structs their life. The separation of data and behavior is important and takes some practice getting used to it. We'll go over adding methods and more to our structs in the next chapter.

10. Enums and Methods

10.1. Introduction

In this chapter we'll translate method syntax, TypeScript enums, and show how to use Rust's `match` expressions.

NOTE	We've started using more TypeScript than JavaScript. To run examples, install `ts-node` with `npm install -g ts-node`. If you want a TypeScript playground, check out this typescript-boilerplate.

10.1.1. Adding methods to our struct

NOTE	Depending on who you talk to and the time of day you'll get different definitions for words like "function," "method," "procedure," et al. When I use "method" and "function" here, I'm differentiating between a callable subroutine meant to be executed within the context of instantiated data (a method) and one that isn't (a function).

The `new` function we added in the last chapter is like a static function in a TypeScript class. You can call it by name without instantiating a `TrafficLight` first.

```
impl TrafficLight {
  pub fn new() -> Self {
    Self {
      color: "red".to_owned(),
    }
  }
}
// let light = TrafficLight::new()
```

Adding methods to Rust structs is straightforward. Mostly.

Consider how you'd add a method like `getState()` in TypeScript:

```
class TrafficLight {
  color: string;

  constructor() {
    this.color = "red";
  }

  getState(): string {
    return this.color;
  }
}

const light = new TrafficLight();
console.log(light.getState());
```

By default, every method you add to a TypeScript class is public and is added to the prototype to make it available to all instances.

In Rust, every function in an `impl` defaults to private and is just an everyday function. To make something a method, you specify the first argument to be `self`. You won't frequently need to specify the type of `self`. If you write `&self`, the type defaults to a borrowed `Self` (i.e. `&Self`). When you write `self`, the type is `Self`. You will see libraries that specify more exotic types for `self`, but that's for another time.

NOTE

You may find yourself in a state where you have an instance of something and you *know* there's a method, but you just can't call it. It could be two things. 1) You're trying to call a trait method that hasn't been imported. You need to import the trait (i.e. `use [...]::[...]::Trait;`) for it to be callable. 2) Your instance needs to be wrapped with a specific type. If you see a function like `fn work(self: Arc<Self>)`, then you can only call `.work()` on an instance that is wrapped with an `Arc`.

To implement our `getState()` method in Rust, we write:

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

This method *borrows* `self`, has a return type of `&String`, and returns a reference to its internal property `color`. The full code looks like this:

```
fn main() {
  let mut light = TrafficLight::new();
  println!("{:?}", light);
}

impl std::fmt::Display for TrafficLight {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    write!(f, "Traffic light is {}", self.color)
  }
}

#[derive(Debug)]
struct TrafficLight {
  color: String,
}

impl TrafficLight {
  pub fn new() -> Self {
    Self {
      color: "red".to_owned(),
    }
  }

  pub fn get_state(&self) -> &str {
    &self.color
  }
}
```

`&self` vs `self`

I made the point of calling out that the method above *borrows* `self`. The natural corollary is that there must exist methods that *don't* borrow `self`, methods that take ownership of `self`. If a method requires an *owned* `self` then that must mean the calling code gives up ownership when it calls the method.

To re-iterate: the calling code loses the instance when it calls the method.

Don't believe me? Let's see what happens when we change the method to this:

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

And call it twice like:

```rust
fn main() {
  let light = TrafficLight::new();
  light.get_state();
  light.get_state();
}
```

This won't compile. Rust will give you an error message you'll get very used to: "use of moved value."

```
error[E0382]: use of moved value: `light`
  --> crates/day-9/structs/src/main.rs:4:18
   |
2  |   let light = TrafficLight::new();
   |       ----- move occurs because `light` has type `TrafficLight`, which does not
implement the `Copy` trait
3  |   println!("{}", light.get_state());
   |                        ----------- `light` moved due to this method call
4  |   println!("{}", light.get_state());
   |                  ^^^^^ value used here after move
   |
note: this function takes ownership of the receiver `self`, which moves `light`
  --> crates/day-9/structs/src/main.rs:25:20
   |
25 |   pub fn get_state(self) -> String {
   |                    ^^^^

For more information about this error, try `rustc --explain E0382`.
```

Losing a value (having it moved) when you call a method might be hard to wrap your head around at first. You never deal with such a concept in JavaScript. However, you do write code where it would make sense. Scenarios like:

- In conversions: When you take some data and convert it to another. In Rust you literally take (take ownership of) some data and return (give away ownership to) new data.
- In cleanup code. When an object gets destroyed or otherwise cleaned up, the calling code is usually done with the instance.
- In builder patterns or chainable APIs. You can take an owned `self`, mutate it, and return it so the calling code can chain on another method.

There are other use cases and even more that require different ways of thinking about `self`. You'll get there in time.

10.1.2. Mutating state

Things are going swimmingly but our `TrafficLight` isn't very useful. It never changes color. Everything in Rust is immutable by default. Even our own `self`. We need to mark this method as one that can needs a mutable `self`. If we wrote our method like this...

```
pub fn turn_green(&self) {
  self.color = "green".to_owned()
}
```

...Rust would yell at us

```
error[E0594]: cannot assign to `self.color`, which is behind a `&` reference
  --> crates/day-8/structs/src/main.rs:32:5
   |
31 |   pub fn turn_green(&self) {
   |                     ----- help: consider changing this to be a mutable reference:
`&mut self`
32 |     self.color = "green".to_owned()
   |     ^^^^^^^^^^ `self` is a `&` reference, so the data it refers to cannot be
written

For more information about this error, try `rustc --explain E0594`.
```

What we need is a mutable reference (see Chapter 5: Borrowing & Ownership).

We need `&mut self`.

```
pub fn turn_green(&mut self) {
  self.color = "green".to_owned()
}
```

We also need to mark our instance of `TrafficLight` as mutable in the calling code. Otherwise Rust will yell at us again.

In `main()`, change `let light = ...` to `let mut light = ...`.

```
let mut light = TrafficLight::new();
```

Our code now looks like this

```
fn main() {
  let mut light = TrafficLight::new();
  println!("{:?}", light);
  light.turn_green();
  println!("{:?}", light);
}

impl std::fmt::Display for TrafficLight {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    write!(f, "Traffic light is {}", self.color)
  }
}

#[derive(Debug)]
struct TrafficLight {
  color: String,
}

impl TrafficLight {
  pub fn new() -> Self {
    Self {
      color: "red".to_owned(),
    }
  }

  pub fn get_state(&self) -> &str {
    &self.color
  }

  pub fn turn_green(&mut self) {
    self.color = "green".to_owned()
  }
}
```

And it's output is

```
[snipped]
TrafficLight { color: "red" }
TrafficLight { color: "green" }
```

10.1.3. Enums

If you're like me, you were getting itchy seeing "red" and "green" written out as strings. Using data types like strings or numbers to represent a finite set of possibilities (i.e. red, green, and yellow) leaves too much opportunity for failure. This is what enums are for.

To migrate our string to an enum in TypeScript, you'd write this:

```
class TrafficLight {
  color: TrafficLightColor;

constructor() {
    this.color = TrafficLightColor.Red;
  }

getState(): TrafficLightColor {
    return this.color;
  }

turnGreen() {
    this.color = TrafficLightColor.Green;
  }
}

enum TrafficLightColor {
  Red,
  Yellow,
  Green,
}

const light = new TrafficLight();
console.log(light.getState());
light.turnGreen();
console.log(light.getState());
```

This prints

```
0
2
```

TypeScript's default enum value representation is a number but you can change it to a string via:

```
enum TrafficLightColor {
  Red = "red",
  Yellow = "yellow",
  Green = "green",
}
```

In Rust, enums are similarly straightforward:

```
enum TrafficLightColor {
  Red,
  Yellow,
  Green,
}
```

With our struct and implementation changing as such:

```
struct TrafficLight {
  color: TrafficLightColor,
}

impl TrafficLight {
  pub fn new() -> Self {
    Self {
      color: TrafficLightColor::Red,
    }
  }

  pub fn get_state(&self) -> &TrafficLightColor {
    &self.color
  }

  pub fn turn_green(&mut self) {
    self.color = TrafficLightColor::Green
  }
}

enum TrafficLightColor {
  Red,
  Yellow,
  Green,
}
```

Now though, we're bitten by the the traits we implemented and derived. VS Code and rust-analyzer are probably already yelling at you because we just made our `TrafficLight` unprintable and undebuggable because `TrafficLightColor` is both unprintable and undebuggable.

We need to derive `Debug` and implement `Display` for `TrafficLightColor` just as we did with `TrafficLight`. We can derive on an `enum` exactly the same way we did with our `struct`.

Add `#[derive(Debug)]` just before the enum definition.

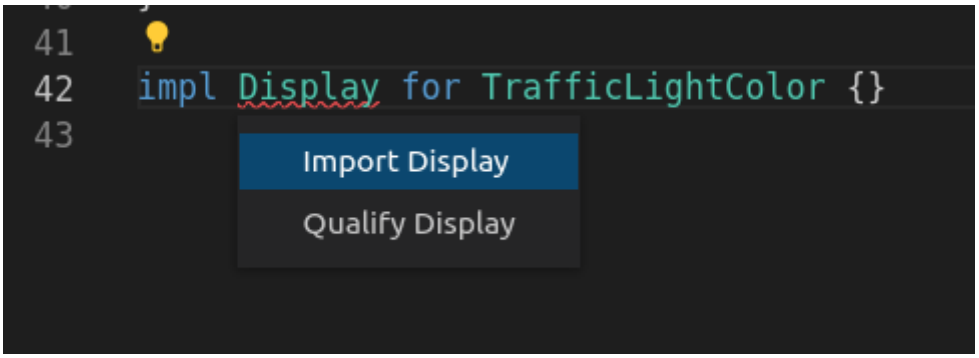
```
#[derive(Debug)]
enum TrafficLightColor {
  Red,
  Yellow,
  Green,
}
```

That took care of one problem. Now we have to implement `Display`. That'll be a little different this time. We want to write out a different string for every variant. To do that we use a match expression. `match` is similar to a `switch/case` except better in every way.

First things first, let's making writing this stuff easier. Write out the impl for `Display` like this:

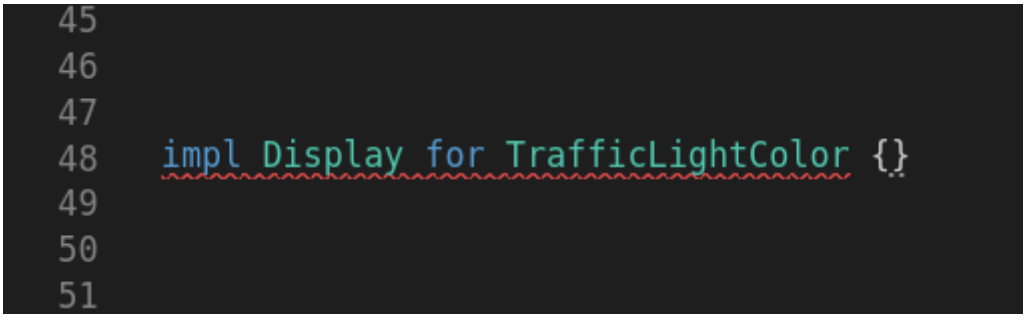
```
impl Display for TrafficLightColor {}
```

If your code follows along with ours, VS Code will complain at `Display`, saying `"cannot find trait `Display` in this scope"`. Place your cursor on display and press `Ctrl+.` (or hover and press "Quick fix"). If rust-analyzer has any suggestions, you'll see them in a drop down menu.



Select `Import Display` and select `std::fmt::Display` from the next drop down. VS Code will take care of adding the `use std::fmt::Display;` line at the top of your file. Nice! Free code!

But now we have an even longer squiggly red line.



Do the `Ctrl+.` shuffle once again, select `Implement missing members` and voila! You've got the boilerplate out of the way.

```
impl Display for TrafficLightColor {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    todo!()
  }
}
```

Get used to using this, it's a life saver.

> **NOTE** the `todo!` macro panics. It's a useful, temporary placeholder.

A match expression allows us to match the result of an expression against a pattern. The following code matches the possible values of `TrafficLightColor` against its `self` to produce an appropriate display string.

```
impl Display for TrafficLightColor {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    let color_string = match self {
      TrafficLightColor::Green => "green",
      TrafficLightColor::Red => "red",
      TrafficLightColor::Yellow => "yellow",
    };
    write!(f, "{}", color_string)
  }
}
```

> **NOTE** `write!` is another macro. It takes a formatter + formatting arguments and returns a Result. A `Result` is like an `Option` and we'll get to it soon. Just think of `write!` as the `print!` you use when implementing `Display`.

Our final code looks like this:

```
use std::fmt::Display;

fn main() {
  let mut light = TrafficLight::new();
  println!("{}", light);
  println!("{:?}", light);
  light.turn_green();
  println!("{:?}", light);
}

impl std::fmt::Display for TrafficLight {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    write!(f, "Traffic light is {}", self.color)
  }
}
```

```
#[derive(Debug)]
struct TrafficLight {
  color: TrafficLightColor,
}

impl TrafficLight {
  pub fn new() -> Self {
    Self {
      color: TrafficLightColor::Red,
    }
  }

  pub fn get_state(&self) -> &TrafficLightColor {
    &self.color
  }

  pub fn turn_green(&mut self) {
    self.color = TrafficLightColor::Green
  }
}

#[derive(Debug)]
enum TrafficLightColor {
  Red,
  Yellow,
  Green,
}

impl Display for TrafficLightColor {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    let color_string = match self {
      TrafficLightColor::Green => "green",
      TrafficLightColor::Red => "red",
      TrafficLightColor::Yellow => "yellow",
    };
    write!(f, "{}", color_string)
  }
}
```

And outputs

```
[snipped]
Traffic light is red
TrafficLight { color: Red }
TrafficLight { color: Green }
```

10.2. Wrap-up

Structs and enums are the most important structures you will deal with in Rust. Rust enums encapsulate common usage like above but are also Rust's answer to union types. They can represent much more complex values than TypeScript. Similarly, match expressions are also much more powerful than we let on above. You'll frequently use enums and match expressions hand in hand. Don't ignore them or push off learning more about them. You'll regret it because it changes the way you think about code in Rust.

It's important to take our time going through these sections. It's easier to highlight the nuance and the error messages when there's a natural flow of code progression. Some sections take this route, others are more direct mapping of TypeScript/JavaScript to Rust. If you have comments on what you like better about one style or the other, drop me a line!

11. From Mixins to Traits

11.1. Introduction

Traits are Rust's answer to reusable behavior. They're similar to JavaScript mixins and the mixin pattern. If you're not familiar with JavaScript mixins, it's no more than adding a collection of methods to arbitrary objects. It "mixes in" properties from one object into another, often using `Object.assign()`. e.g.

```
const utilityMixin = {
  prettyPrint() {
    console.log(JSON.stringify(this, null, 2));
  },
};

class Person {
  constructor(first, last) {
    this.firstName = first;
    this.lastName = last;
  }
}

function mixin(base, mixer) {
  Object.assign(base.prototype, mixer);
}

mixin(Person, utilityMixin);

const author = new Person("Jarrod", "Overson");
author.prettyPrint();
```

The above declares a class named `Person` and a plain ol' JavaScript object called `utilityMixin`. The `mixin()` function uses `Object.assign()` to add all the properties from `utilityMixin` to `Person`'s prototype, thus making them available to every instance of `Person`. It's a useful pattern. It's an option that sidesteps long prototype chains or general-purpose classes.

The above is JavaScript. You can use mixins in TypeScript, but it's more complicated. It highlights how being "Just JavaScript, with types" starts to break down.

Rust's traits are very similar to JavaScript's mixins. They're a collection of methods (or method signatures). A lot of documentation compares structs and traits to object oriented paradigms and inheritance. Ignore all of that. It only makes traits harder to understand.

Traits are just a collection of methods.

11.1.1. Stretching our TrafficLight example

In the previous days, we added a `get_state()` method to our `TrafficLight` struct. We're rapidly

becoming experts in lighting management. It's a perferct time to start adding functionality for every light we have. The first light to add is a simple household bulb. It doesn't do much. It's either on or off.

There should be no surprises in the implementation.

```
#[derive(Debug)]
struct HouseLight {
  on: bool,
}

impl Display for HouseLight {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    write!(f, "Houselight is {}", if self.on { "on" } else { "off" })
  }
}

impl HouseLight {
  pub fn new() -> Self {
    Self { on: false }
  }
  pub fn get_state(&self) -> bool {
    self.on
  }
}
```

Now let's make a generic `print_state()` function. We want a function that prints the state of any light we pass to it.

```
fn print_state(light: ???) {
}
```

But what do we take in? We can't take in an arbitrary list of types like we can in TypeScript. We can't write

```
function print(light: TrafficLight | HouseLight) {...}
```

In the last chapter I talked about enums as one way Rust deals with the lack of union types. Another way is with traits. The difference is what we're looking for, the type or a subset of the behavior.

In this example, we don't actually care what kind of light we take in. We only want to query for its name and state. We only want a subset of behavior.

That's where traits come in.

11.1.2. Traits

Trait definitions start with the `trait` keyword and are structured similarly to `impl`s. They consist of methods that look almost identical to what we'd write in an actual `impl`. The one major difference is that you can write trait methods that are missing a body.

NOTE	Trait methods can include a method body which acts as a default implementation. Implementers can choose to override the default implementation.

Let's call this trait `Light` and started filling it out with a `get_name()` method.

```
trait Light {
  fn get_name(&self) -> &str;
}
```

To implement a trait we use `impl` block like we did for our `struct`. This time though, we write `impl [trait] for [struct]` and we're limited to the methods available on the trait.

```
impl Light for HouseLight {
  fn get_name(&self) -> &str {
    "House light"
  }
}

impl Light for TrafficLight {
  fn get_name(&self) -> &str {
    "Traffic light"
  }
}
```

Now we can start to implement a `print_state()` function. To accept an argument that implements a trait you write `impl [trait]`.

```
fn print_state(light: &impl Light) {
  println!("{}", light.get_name());
}
```

When we try to migrate our `get_state()` methods over to the trait, we run into a snag. Each of the light's state has different types. Since we are printing them with the debug formatter right now, your first thought might be to translate what we just did like this:

```
trait Light {
  fn get_name(&self) -> &str;
  fn get_state(&self) -> impl std::fmt::Debug;
}
```

But that won't work. Rust complains with the error `impl `Trait not allowed outside of function and method return types`.

```
error[E0562]: `impl Trait` not allowed outside of function and method return types
  --> crates/day-10/traits/src/main.rs:17:27
   |
17 |   fn get_state(&self) -> impl std::fmt::Debug;
   |                          ^^^^^^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0562`.
```

But... we *are* trying to use it as a method return type... What gives?

11.1.3. `impl` vs `dyn`

To use traits here we need to use `dyn [trait]`. Using `dyn [trait]` vs `impl [trait]` is a matter of whether or not Rust needs or is able to know a value's concrete type at compile time. We can't use `impl std::fmt::Debug` here because every implementation might return a different actual type. Using `dyn` is like crossing a barrier where you trade optimizations for flexibility. Once a value crosses the `dyn` barrier, it loses its type information and is essentially just a blob of data and a pointer to the methods on a trait.

So we change our signature and implementations to:

```
trait Light {
  fn get_name(&self) -> &str;
  fn get_state(&self) -> &dyn std::fmt::Debug;
}

impl Light for HouseLight {
  fn get_name(&self) -> &str {
    "House light"
  }

  fn get_state(&self) -> &dyn std::fmt::Debug {
    &self.on
  }
}

impl Light for TrafficLight {
  fn get_name(&self) -> &str {
    "Traffic light"
  }

  fn get_state(&self) -> &dyn std::fmt::Debug {
    &self.color
  }
}
```

NOTE	Rust must know the size of everything at compile time. It can't do that with `dyn [trait]` values because they don't have a concrete type. With no known size, it's "unsized." What *is* sized is a reference. A reference to a `dyn [trait]`, i.e. `&dyn [trait]` is OK.

Our full code now looks like this:

```
use std::fmt::Display;

fn main() {
  let traffic_light = TrafficLight::new();
  let house_light = HouseLight::new();

  print_state(&traffic_light);
  print_state(&house_light);
}

fn print_state(light: &impl Light) {
  println!("{}'s state is : {:?}", light.get_name(), light.get_state());
}

trait Light {
  fn get_name(&self) -> &str;
  fn get_state(&self) -> &dyn std::fmt::Debug;
}

impl Light for HouseLight {
  fn get_name(&self) -> &str {
    "House light"
  }

  fn get_state(&self) -> &dyn std::fmt::Debug {
    &self.on
  }
}

impl Light for TrafficLight {
  fn get_name(&self) -> &str {
    "Traffic light"
  }

  fn get_state(&self) -> &dyn std::fmt::Debug {
    &self.color
  }
}

impl std::fmt::Display for TrafficLight {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    write!(f, "Traffic light is {}", self.color)
  }
```

```
}

#[derive(Debug)]
struct TrafficLight {
  color: TrafficLightColor,
}

impl TrafficLight {
  pub fn new() -> Self {
    Self {
      color: TrafficLightColor::Red,
    }
  }

  pub fn turn_green(&mut self) {
    self.color = TrafficLightColor::Green
  }
}

#[derive(Debug)]
enum TrafficLightColor {
  Red,
  Yellow,
  Green,
}

impl Display for TrafficLightColor {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    let color_string = match self {
      TrafficLightColor::Green => "green",
      TrafficLightColor::Red => "red",
      TrafficLightColor::Yellow => "yellow",
    };
    write!(f, "{}", color_string)
  }
}

#[derive(Debug)]
struct HouseLight {
  on: bool,
}

impl Display for HouseLight {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    write!(f, "Houselight is {}", if self.on { "on" } else { "off" })
  }
}

impl HouseLight {
  pub fn new() -> Self {
    Self { on: false }
```

```
  }
}
```

Which outputs

```
[snipped]
Traffic light's state is : Red
House light's state is : false
```

The output isn't stellar but we can work on that another time. Our code is getting pretty big to sit in one file now. It's time to start cutting it up.

11.1.4. Additional reading

- The Rust Book: ch 10.02
- Rust by Example: Traits
- Common Rust Traits
- The Rust Reference: Traits

11.2. Wrap-up

Traits are *everywhere* in Rust and it's worth reading Rust code on Github or in the standard library. Some languages (Go, notably) are very straightforward and clear. There is generally one "right" way to do something. Rust is anything but that. There are 800 different ways to do everything and its important to read existing code rather than work in a vacuum.

NOTE	Having 800 ways to do any one thing makes Rust the spiritual successor to perl. Don't say that out loud though. You won't make any friends.

The next chapter goes over Rust's module system. It's straightforward once you "get it," but you're coming from node.js. Node has the simplest module system that I've ever used. Once you get over the first few Rust module WTFs, it won't stand in your way again.

12. The Module System

12.1. Introduction

When released, one of node.js's outstanding features was how simple it was to use. Want to run a JavaScript file? Just use `node file.js`. Want to include a local module? Then `require("./module.js")`. If your `require()` wasn't a relative or absolute path, then node looked in the `node_modules` folder. It was a beautiful time. If you've gotten into node.js within the last few years, you probably have a different view of how simple it is or isn't. It ***was*** simple, once upon a time. I swear.

Rust's module system is a bit more nuanced. Once you "get it," it won't stand in your way. Until then you might be left wondering "WTF" more than once.

12.2. "How do I import a file in Rust?"

The quick answer is, you don't. You don't import files in Rust. You declare modules. If you declare a module with an empty body, then Rust will look for it on the file system according to its resolution algorithm. That resolution algorithm is highly dependent on the project's module structure.

You declare modules with the `mod` keyword. If you write `mod my_module;` then Rust will look for a module named `my_module` which may be found in a file named `my_module.rs`. But it may not.

There isn't a relative file path relationship like you'd have in node.

12.3. "How do I import functions from other modules?"

The `use` statement brings items from a module into scope of the current module so you can use them without qualifying them. This works with both external dependencies and local modules.

These two are largely the same:

```
some_module::some_function();
```

and

```
use some_module::some_function;
some_function()
```

Use `{}` to import multiple items, e.g.

```
use std::io::{Read, Write};
```

Use `super` to import items from a parent module, e.g.

```
fn work() {}

mod test {
    use super::work;
}
```

Use `crate` to import items starting from the crate root, e.g.

```
use crate::some_module::some_function
```

12.4. The pieces of the Rust Module System

12.4.1. Crates and the crate root

A Rust project is called a `crate`. The `crate root` is the source file that the Rust compiler starts with. In node that's commonly called `index.js` and you would define it by setting the `"main"` field in your package.json. In Rust the root is typically `main.rs` for standalone executables and `lib.rs` for libraries. It's configurable in your `Cargo.toml`.

The crate root dictates the root path where Rust will start its search for imported modules.

12.4.2. What is a module?

It's easiest if you start with the understanding that Rust modules have *nothing* to do with file names or paths. You'll see documentation that says otherwise and projects that make this appear to be the case, but it's a lie. Modules are purely a concept within Rust's brain.

A module is like a namespace where you bucket similar things. Modules are barriers for visibility, i.e. public, private, etc. You can have many modules in one file. Even nesting them in a single file is allowed.

```
fn main() {
  my_module::print(my_module::submodule::MSG);
}

mod my_module {
  pub fn print(msg: &str) {
    println!("{}", msg);
  }

  pub mod submodule {
    pub const MSG: &str = "Hello world!";
  }
}
```

Separating logic into different files is important for humans. Not Rust. Luckily Rust has support for

automatically looking up modules on the local filesystem.

Let's say we want to extract `my_module` into another file. We cut and paste its code into a file called `my_module.rs` and keep an empty `mod my_module;` in our `main.js`. The result looks like this:

```
fn main() {
  my_module::print(my_module::submodule::MSG);
}

mod my_module;
```

```
pub fn print(msg: &str) {
  println!("{}", msg);
}

pub mod submodule {
  pub const MSG: &str = "Hello world!";
}
```

Our `main.js` shows we're declaring a module called `my_module`, but it's empty. Rust needs to find it.

12.4.3. How Rust finds files

The resolution algorithm in a nutshell is:

1. Start in a directory with the same path parts as the current module. For example, if you're in the crate root, the path is `./`. If you're in the module `one::two::three`, then start in `./one/two/three`.
2. Look for a file with the name of the imported module (e.g. `mod my_module` would look for `my_module.rs`)
3. Look for a file named `mod.rs` in a directory with the name of the imported module (e.g. `my_module/mod.rs`)
4. If none are found, complain. If both are found, complain. If one is found, use that one.

Note: The `mod.rs` part is technical baggage. It used to be the only way. It's not recommended anymore but you'll still see it on some projects so it's worth knowing about.

What this means is that well-organized projects *seem* like their modules mirror a file system with relative imports. This can bite you if you assume that's always true. The appearance is due to a project being well-organized. It's not a structure Rust imposes.

Take a lone `main.rs` with the code below for example

```rust
fn main() {}

mod module_a;

mod one {
  mod two {
    mod module_b;
  }
}
```

Both `mod module_a` and `mod module_b` statements are in the same file. Where does Rust look for the modules on disk?

Rust looks for `module_a` in `./module_a.rs` or `./module_a/mod.rs`.

Rust looks for `module_b` in `./one/two/module_b.rs` or `./one/two/module_b/mod.rs`. `mod module_b` is declared in `one::two`, so it's namespace parts are `one::two::module_b` and that's what dictates the lookup.

12.4.4. Visibility

By default everything you define is private. BUT — and this is a big BUT — visibility is at the module barrier, not an item's definition. That means everything in a module can access everything else in the same module. If you take the `pub`s off of everything in our traffic light example, it will still work. Sorry to trick you.

NOTE	Everything is private by default except for trait methods and enum variants. All trait methods and enum variants are public by default. In practical terms, that means they have the same visibility as the trait or enum they're defined on. It wouldn't make much sense to have them be anything else.

Visibility also only works up and out. Modules defined closer to the root can't see anything defined deeper unless you change its visibility. You change visibility with the `pub` keyword.

`pub` makes something completely public, but only if it's reachable via the visibility chain. That is, everything leading up to the `pub` item must also be `pub`.

You can tailor the visibility with modifiers on `pub`, e.g.

- `pub(crate)` - public within the crate, i.e. not externally visible.
- `pub(super)` - public to the parent module only.

There are other modifiers, but they're less common. You can learn about them in the [Additional reading section][].

12.4.5. Traffic light exercise

The traffic light example from the past few days has grown pretty large. It's due for some

refactoring. Try to do it yourself. See what hurdles you run into before referring to the code repository to see how I've done it.

Tips:

- Copy/paste is your friend. This won't need to refactor code structure or names.
- You must declare modules from their root module file, they can't all be declared from `main.rs`
- After moving code around, you'll need to adjust visibility to still use the items in `main()`.
- You'll likely need to update or add `use` statements in all the files you create.
- Visual Studio Code and rust-analyzer is your friend. Use the hover tips and "Quick code" fixes liberally.

12.4.6. Additional reading

- The Rust Book: ch 7
- Rust by Example: Modules
- Rust by Example: Crates
- The Rust Reference: Modules

12.5. Wrap-up

Repeat after me: `mod` is not `import`. Again, for those in the back: `mod` is not `import`. Once you get passed that misconception (and the mod.rs wonkiness), the module system is easy to manage. We've finally gone through enough to circle back around to the second part of our Strings guide. We'll get to that immediately in the next chapter.

13. Strings, Part 2

13.1. Introduction

Now that you're getting confident, you are probably itching to start your own projects. You know how to write some basic Rust. You know how to structure code. You're done with contrived examples with traffic lights. As you play around and start writing your API, you'll inevitably write some function that needs an owned `String`. That's easy. We've done that before. Then it hits you…

Your API, dotted with functions like below:

```
fn my_beautiful_function(arg1: String, arg2: String, arg3: String) {
  /* ... */
}
```

forces your users to write code like this

```
my_beautiful_function(
  "first".to_owned(),
  "second".to_owned(),
  "third".to_owned()
);
```

It's so ugly. I just couldn't believe it at first. All I wanted was to create a satisfying API that could take a `String` as well as string literals. I didn't want to make my users do things like add `.to_owned()` all over the place. It's madness. It's hideous.

I spent a long time tracking down what to do. This is my story.

13.2. Should I use `&str` or `String` for my function arguments?

The first question you need to ask is: "Do I need to own or borrow the passed value?"

For those of you still wrapping your head around owning and borrowing, think of this question as: "Do I need my own version of this value or do I just need to look at its data and move on?"

13.2.1. When you're borrowing the value

If you *don't* need your own version, then accept a reference. The main question then becomes "Should I use `&str` or `&String`" and the answer is almost always `&str`. Why? Well look at the function below.

```
fn print_str(msg: &str) {
  println!("{}", msg);
}
```

You can pass a string literal directly, a `&str` assigned to a variable, and you can pass a reference to a `&String` which works automagically.

```
fn main() {
  let string_slice = "String slice assigned to variable";
  let real_string = "Genuine String".to_owned();
  print_str(string_slice);
  print_str("Literal slice");
  print_str(&real_string);
}
```

Why? Because the trait `Borrow<str>` is implemented for `String`. Anywhere you need a borrowed `str`, you can use a borrowed `String` without any hassle. The reverse is not true. If you change the function signature to accept a `&String` and try to pass a `&str`, you'll get a compile error.

13.2.2. When you need an owned value

If you need an owned value then you need a `String` but there are a couple things to consider.

1. It is cumbersome to manually convert loads of `&str`s to `String`. This is a genuine problem when you want to expose a satisfying, easy to use API. The example in the introduction is contrived but it is exactly what you'll deal with frequently.
2. You should let the users of your API decide how to create owned values for you. In other words, you shouldn't simply accept `&str`s everywhere and convert them yourself. The user may have better ways of getting you the data.

Here's where you want to think about your API. Are you looking to own a string that comes from *wherever*? Or do you expect your functions to be called with a mix of string literals and/or `String`s?

If it's the former, then you're not really looking for a `&str` *or* a `String`. You're looking for something that has a `.to_string()` method. That is, you're looking for something that implements `ToString`. Types that implement `ToString` are common, you get it for free by implementing `Display`.

If it's the latter, then you're looking for the common ground between a `String` and `&str`, which we found clues to in the section above.

For functions that accept potentially generated strings

For a function that accepts arbitrary strings from anywhere — generated or non — you can accept arguments that implement `ToString`.

```
use std::str::FromStr; // Imported to use Ipv4Addr::from_str

fn main() {
  let ip_address = std::net::Ipv4Addr::from_str("127.0.0.1").unwrap();
  let string_proper = "String proper".to_owned();
  let string_slice = "string slice";
  needs_string(string_slice);
  needs_string("Literal string");
  needs_string(string_proper);
  needs_string(ip_address);
}

fn needs_string<T: ToString>(almost_string: T) {
  let real_string = almost_string.to_string();
  println!("{}", real_string);
}
```

We talked about `impl [trait]` vs `dyn [trait]` in the last chapter but now I through in new syntax above. This is Rust's generic syntax. The function `needs_string` above could have been written like this:

```
fn needs_string(almost_string: impl ToString) {
  let real_string = almost_string.to_string();
  println!("{}", real_string);
}
```

Nothing would need to change in the code. What's the difference? Very little. `impl [trait]` in argument position is less powerful than generic syntax. Rust also has a `where` keyword which you can use to make the same thing yet another way:

```
fn needs_string<T>(string: T)
where
  T: ToString,
{
  println!("{}", string);
}
```

There's zero difference between the `where` syntax and the `<T: ToString>` syntax. Adding the separate `where` clause is for readability.

For functions that expect a mix of string literals and `String`s

If you didn't read this section, you wouldn't lose much. Using the `ToString` method covers a lot of cases. You do however open your users up to the same concerns we described with using `.to_string()` to convert `&str` to `String` in Chapter 6: Strings, Part 1. Many structs implement `Display`. A user could pass a lot of objects to the function above without the compiler complaining. Additionally, implementations of `.to_string()` may not always be cheap. Your users don't

necessarily know the internals of your code. You might be doing a lot of extra work for no good reason.

Since we already learned that a `&String` can take the place of a `&str`, we can generalize our inputs to anything that can be borrowed like a `str`.

You *could* use anything that implements `Borrow<str>` and the below would work. However, you should accept anything that implements `AsRef<str>` instead. What's the difference? Borrow assumes more and can fail, `AsRef<str>` assumes little and explicitly **must not fail**. There are more differences but they don't matter for our usage here.

The below is extremely similar to the above but notice that our `ip_address` is no longer a valid argument. Just because it has a printable string form doesn't mean it can be trivially taken as a `str` reference.

```
use std::str::FromStr; // Imported to use Ipv4Addr::from_str

fn main() {
  let ip_address = std::net::Ipv4Addr::from_str("127.0.0.1").unwrap();
  let string_slice = "string slice";
  let string_proper = "String proper".to_owned();
  needs_string(string_slice);
  needs_string("Literal string");
  needs_string(string_proper);
  // needs_string(ip_address); // Fails now
}

fn needs_string<T: AsRef<str>>(almost_string: T) {
  let real_string = almost_string.as_ref().to_owned();
  println!("{}", real_string);
}
```

13.2.3. Additional reading

- The Rust Book: ch 10
- Rust by Example: Generics
- The Rust Reference: Generic Parameters

13.3. Wrap-up

Strings used to seem so simple. We were so naive. I bet every time you dive back into node.js your eyes are going to tear up with joy. Eventually you'll have the same feeling with Rust. You'll appreciate how protective Rust is and how fast everything runs. Rust and JavaScript truly are a beautiful pair.

Next up we'll dive into error handling, the `Option` enum, and the `Result` enum.

14. Demystifying Results & Options

14.1. Introduction

I briefly touched on the `Option` enum on in Chapter 8 when we were using `HashMap` methods to look up keys. There's no way for the `HashMap` to guarantee that a key will exist, so its API needs to account for returning *something* as well as *nothing*. Rust has no concept of `undefined` or `null`, like JavaScript. It needs to represent nothingness safely. That's where the `Option` comes in.

Nothingness is like an expected error case for functions that can return either *something* or *nothing*. `try/catch` statements in JavaScript are another way to deal with reasonable error cases. In those cases the answer is either *your result* or *uh oh, something went wrong*. That's what a `Result` is. `Result` and `Option` go hand in hand. They are treated similarly and can be converted from one to another when necessary.

14.2. `Option` recap

If you've gotten this far without exploring enums, you are going to need to brush up quickly with the links in the additional reading section.

Rust enums differ from many other language implementations. They can represent rich and varied values and carry around behavior just like any struct. The `Option` enum is defined in just a few lines here:

```
pub enum Option<T> {
    /// No value
    None,
    /// Some value `T`
    Some(T),
}
```

That's it. It's either `Option::None` and contains no value, or `Option::Some(T)` which contains a `T`. We touched on generic functions briefly in the previous chapter and you can find more information in that day's additional reading section. This is how generic types look on data structures. `Option` doesn't care what `T` is, there are no constraints.

Creating and returning an `Option` is as easy as it gets in Rust.

```
fn main() {
  let some = returns_some();
  println!("{:?}", some);

  let none = returns_none();
  println!("{:?}", none);
}

fn returns_some() -> Option<String> {
  Some("my string".to_owned())
}

fn returns_none() -> Option<String> {
  None
}
```

```
[snipped]
Some("my string")
None
```

NOTE	We still need to specify the value of `T` in the `Option<T>` even if we return a `None`.

NOTE	We can use `Some` & `None` rather than `Option::Some()` and `Option::None` because they are pre-imported by Rust's `prelude` (the common set of imports that you can always rely on). Read up on the `std::prelude` here.

14.3. Result

`Result` is very similar to `Option` except its failure case also contains a value. The value in the `Result::Err` variant usually follows some conventions, but you can see by its implementation it has no constraints.

```
pub enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

You can return your `Ok()` or `Err()` freely. There's nothing special about how they are created.

```
fn main() {
  let value = returns_ok();
  println!("{:?}", value);

  let value = returns_err();
  println!("{:?}", value);
}

fn returns_ok() -> Result<String, MyError> {
  Ok("This turned out great!".to_owned())
}

fn returns_err() -> Result<String, MyError> {
  Err(MyError("This failed horribly.".to_owned()))
}

#[derive(Debug)]
struct MyError(String);
```

```
[snipped]
Ok("This turned out great!")
Err(MyError("This failed horribly."))
```

NOTE I used a custom struct to highlight that an `Err` can contain anything. A basic struct, your struct, a `String`, someone else's error, a `HashMap`, whatever.

14.4. The problem with `.unwrap()`

`Option` and `Result` seem pretty straightforward, don't they?

The confusion comes from how you actually get your damn value out. You've seen the `.unwrap()` method in this guide and you've undoubtedly seen it all over Rust examples. You have probably also seen the warnings against using it. The warning comes for good reason. If you `.unwrap()` a `None` or `Err`, your code will panic and your application will probably die. That's no good.

If we wanted random application deaths due to failure cases we'd just write JavaScript. I'm kidding but also I'm not. With Rust, you can be 99% sure your application will be bullet-proof, **as long as you respect its warnings**. You're not here to write working code fast, you're here to write fast code that always works.

Unfortunately, respecting Rust's warnings can be tedious and repetitive. That's why you see shortcuts like `.unwrap()` in examples. Example code is to get you up and running. It's not there to show you how to handle every possible error case.

14.4.1. So how *do* I get my value out?

Here are your options:

`.unwrap()`

So now that you know you shouldn't use `.unwrap()`, here's how you use `.unwrap()`.

Use `.unwrap()` when you're sure you have a `Some()` or an `Ok()`. Look back at our IP address example from the last chapter:

```
let ip_address = std::net::Ipv4Addr::from_str("127.0.0.1").unwrap();
```

I *know* that `"127.0.0.1"` is a valid IPv4 address and will be parsed successfully. This is example code but it's also OK in production. `from_str()` needs to return a `Result` because there are an infinite number of strings that *won't* parse into an IPv4 address. I'm not passing it any of those.

The IP example relies on my knowledge of IP addresses. You can programmatically generate the confidence necessary to use `.unwrap()` as well. You can use a `HashMaps`'s `.contains_key()` method to ensure there is a key. Then you're free to `.unwrap()` the resulting `.get()` without fear.

That said, it's valuable to always err on the side of caution. Rust won't alert you if a refactor sidesteps your expectations.

`.unwrap_or(value)`

Link to documentation

`.unwrap_or()` is for providing a custom default value in the event of a failure message. The value needs to be the same type (`T`) as `Ok(T)` or `Some(T)`.

```
let default_string = "Default value".to_owned();

let unwrap_or = returns_none().unwrap_or(default_string);

println!("returns_none().unwrap_or(...): {:?}", unwrap_or);
```

`.unwrap_or_else(|| {})`

Link to documentation

`.unwrap_or_else()` is nearly identical to `.unwrap_or` except it takes a function. The return value of the function is used when the `Option` or `Result` is `None` or `Err`. You'd use this in situations where the default value might be expensive to compute and there's no value computing it in advance.

As with `.unwrap_or()`, the return type needs to be the same type of `T`.

```
let unwrap_or_else = returns_none()
  .unwrap_or_else(|| format!("Default value from a function at time {:?}", Instant
::now()));

println!(
  "returns_none().unwrap_or_else(|| {{...}}): {:?}",
  unwrap_or_else
);
```

> **NOTE**
>
> The `|| ...` syntax is Rust's closure syntax.
>
> In JavaScript/TypeScript, you'd have `(arg1: number) => arg1 + 2`. In Rust it is `` `|arg1: i64| arg1 + 2 ` ``. Curly braces are optional when there's a single expression, just like in JavaScript. We'll go over closures in more detail in a later section.

`.unwrap_or_default()`

Link to documentation

`.unwrap_or_default()` defers to a type's `Default` value if none exists. `Default` is a trait like `Debug` or `Display`. It has one method, `default` and takes no arguments. A type that implements `Default` can be instantiated with `[Type]::default()`. In other languages, you might consider this an implementation of the `Null object pattern`. It's what you can resort to when you need a neutral value of a type.

In TypeScript, you might do something like:

```
let my_string = maybe_undefined || "";
```

In Rust, it would be:

```
let my_string = maybe_none.unwrap_or_default(); // Assuming `T` is `String`.
```

> **NOTE**
>
> You can implement `Default` like this:

```
impl Default for MyStruct {
  fn default() -> Self {
    // Return whatever is suitable as a default.
  }
}
```

Pattern matching

We can use the `match` expression to match the enum's variants and return the inner value or a suitable default.

```
let match_value = match returns_some() {
  Some(val) => val,
  None => "My default value".to_owned(),
};

println!("match {{...}}: {:?}", match_value);
```

`if let` expressions

You can enter a block conditionally based off an enum's variant. It's easier to explain with an example:

```
if let Some(val) = returns_some() {
  println!("if let : {:?}", val);
}
```

If the `Option` returned by `returns_some()` is `Some()` then its inner value will be bound to the identifier `val`. It's strange syntax to get used to, but it's useful.

Automagic unwrapping with `?`

Short circuiting, or returning early, is a common way of dealing with error cases. When you get an error or a `None`, return right away and let the caller deal with it. `Rust` embodies this concept into the `?` operator.

The code below shows a few new tricks.

```
use std::fs::read_to_string;

fn main() -> Result<(), std::io::Error> {
  let html = render_markdown("./README.md")?;
  println!("{}", html);
  Ok(())
}

fn render_markdown(file: &str) -> Result<String, std::io::Error> {
  let source = read_to_string(file)?;
  Ok(markdown::to_html(&source))
}
```

- First, we've changed our `main()` to return a `Result` by using `-> Result<(), std::io::Error>` on line 3. Remember, `()` is the unit type. It's another way of saying nothing. You can read a return value like `-> Result<(), ...>` as "Returns nothing, but may fail".
- Second, we're using `std::fs::read_to_string()` on line 10. It takes a path and returns a `Result<String, std::io::Error>`. That is, it returns either the contents of a file as a `String`, or it returns an error of the type `std::io::Error`.

- Third, We automagically unwrap the result into the variable `source` with the `?` operator on line 10. If the result is an error, the `?` returns the result back to the caller, in this case `main()`.
- Fourth, We automatically unwrap the result from `render_markdown` in `main()` with another `?` on line 4. Since there's no caller above `main()`, an error here will kill our program.
- Fifth, We finish our `main()` with `Ok(())` because our return type is `Result<(), ...>`. We don't care about the value we return, but we have to return `Ok()` regardless.

`?` vs `try!`

You may see references to the `try!` macro in some older posts. `try!` was the precursor to `?`. While `try!` is deprecated in favor of `?`, it's still a great way to understand what's happening. The implementation is here and below.

`try!` is a macro and uses macro syntax which will look foreign at first. Macros are beyond the scope of this guide but you're a smart cookie. I bet you can get the gist of what's going on here:

```
macro_rules! r#try {
    ($expr:expr $(,)?) => {
        match $expr {
            $crate::result::Result::Ok(val) => val,
            $crate::result::Result::Err(err) => {
                return $crate::result::Result::Err($crate::convert::From::from(err));
            }
        }
    };
}
```

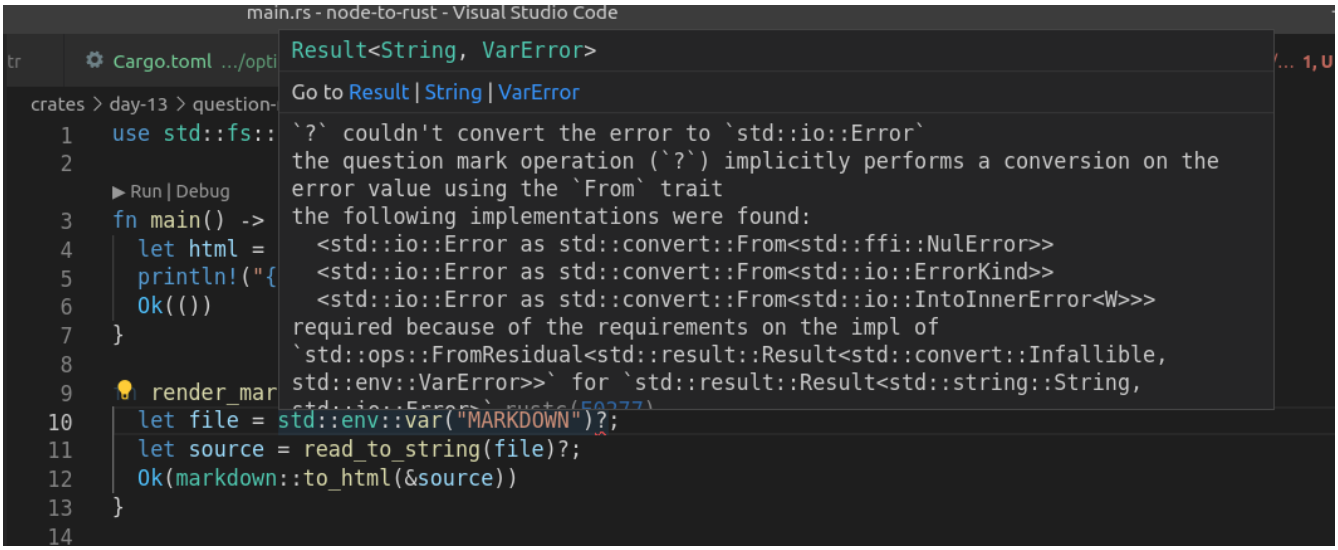
The `try!` macro takes an expression and uses that expression in a `match` statement. If the `expression` is `Ok` it returns the inner value. If it's an error, it returns early *and converts the error into the returning* `Result`*'s Error type*. That last part is an important one. Let's see what happens if we change our example to take our file path from an environment variable rather than a hardcoded string.

```
use std::fs::read_to_string;

fn main() -> Result<(), std::io::Error> {
  let html = render_markdown()?;
  println!("{}", html);
  Ok(())
}

fn render_markdown() -> Result<String, std::io::Error> {
  let file = std::env::var("MARKDOWN")?;
  let source = read_to_string(file)?;
  Ok(markdown::to_html(&source))
}
```

We've added one line to get the value of an environment variable named `"MARKDOWN"`. That function will fail if no such variable exists. We use another question mark (`?`) to short circuit but now we have a compilation error: `` `?` could not convert the error to std::io::Error`...



That error leaves us at another Rust **WTF**-juncture. A WTFuncture. A Rust-T-F. You understand `Option`s and `Result`s. They're not that scary, but how the heck do you deal with all the different errors? We'll get to that in the next chapter.

14.4.2. Additional reading

- The Rust Book: ch 06 - Enums
- Rust by Example: Enums
- The Rust Reference: Enumerations
- Rust by Example: match
- Rust by Example: if let
- Rust docs: Result
- Rust docs: Option

14.5. Wrap-up

`Option`s and `Result`s are everywhere in Rust. You should try thinking in terms of them right away. Enums themselves are everywhere, for that matter. You will often find its better to return or accept values in terms of enums instead of magic values like strings or numbers and booleans that mean more than just `true` or `false`.

This section was all lead-in to the next part of the guide where we go over how to deal with the `Err` side of the `Result`.

15. Managing Errors

15.1. Introduction

So much of Rust's documentation is explanation-heavy vs JavaScript's example-oriented culture. The examples that *do* exist often involve unrelated concepts that add no value to the topic in question. I still aim to submit a PR to fix this example for read_to_string. That example inexplicably relies on the contents of a file to be a valid socket address to complete successfully. Maybe I expect too much, but I'm old and unlikely to change. I learn from example, from trial and error. I *could* sit through an explaination about how combustion is an exothermic chemical reaction and that a humans' epidermis starts getting damaged at temperatures above 118 degrees Fahrenheit. *Or* I could touch a stove once.

I've read a lot of Rust while I learned. I kept trying to figure out the "right way" to do things. Judging from a lot of the code in public projects, I'm not the only one who wasn't able to get comfortable with Rust right away. I've seen many crates I thought ***must*** be examples of good, practical code only to learn that the authors were on the same journey I was. It's difficult to write good Rust when real world code is a mess of people trying to figure things out and documentation amounts to a pile of Lego without instructions.

Error handling is a good example of this problem turned up to 11. The gap from "This is what a `Result` is and how `?` works" to being useful is massive. It's not something you can ignore, either. It's a major hurdle to being productive.

15.1.1. Error handling in Rust

The one thing you need to know right now is that you **must** start caring more about errors than you probably ever have before.

When you start a project, the first thing you think about needs to be "How do I manage errors?"

15.1.2. Dealing with multiple error types

Let's take a look at the markdown renderer that wouldn't compile from the last chapter.

```
use std::fs::read_to_string;

fn main() -> Result<(), std::io::Error> {
  let html = render_markdown()?;
  println!("{}", html);
  Ok(())
}

fn render_markdown() -> Result<String, std::io::Error> {
  let file = std::env::var("MARKDOWN")?;
  let source = read_to_string(file)?;
  Ok(markdown::to_html(&source))
}
```

The problem with this code stems from mismatched types. We return a `Result` with an error type of `io::Error` but we're using `?` in two places, one of which returns a different kind of error. `read_to_string` returns a `Result<String, io::Error>` which is great, but `env::var()` returns a `Result<String, env::VarError>`.

We need a general type that matches multiple errors. If you've been following along day-by-day, then you know this means one of two things: traits or enums.

Option 1: `Box<dyn Error>`

NOTE	This option is a good learning exercise. It's not code you should write in any meaningful project.

Boxing your errors relies on those errors implementing the `Error` trait.

What's a `Box`?

Rust must know the size of everything at compile time. Since a `dyn [trait]` value has lost its concrete type (see: Chapter 10: From Mixins to Traits), Rust can't know its size. It's "unsized." A reference, on the other hand, does have a concrete size. It's the size of a pointer. On a 32-bit machine its 32 bits. A 64-bit machine, it's 64 bits. I know, lin./images/blog/whoa.gif[whoa].

But we can't simply return a reference willy nilly. Referenced data has to live somewhere. If it lives in (is owned by) our function then Rust won't let us return a reference at all. The value's lifetime will be too short. We've dealt with lifetimes *all over* this guide, they've just been hidden. We haven't had to tackle them directly but we're getting closer.

Think of `Box`-ing something as taking a value, putting it somewhere where it'll live for a long time, and holding a pointer to that location. It's how we get around wanting or needing to return a reference for a value that we would normally drop.

The code to make this work is below:

```rust
use std::{error::Error, fs::read_to_string};

fn main() -> Result<(), Box<dyn Error>> {
  let html = render_markdown()?;
  println!("{}", html);
  Ok(())
}

fn render_markdown() -> Result<String, Box<dyn Error>> {
  let file = std::env::var("MARKDOWN")?;
  let source = read_to_string(file)?;
  Ok(markdown::to_html(&source))
}
```

This often works, but if you remember from Chapter 13: Results & Options, the `Result` type doesn't constrain the error's type. You will eventually encounter errors that do not implement the `Error` trait and this method falls flat.

Option 2: Create your own custom Error type

Using `dyn [trait]` crosses the `dyn` barrier which results in lost type information (as you remember). It's a handy way of getting code running but it's not a longterm solution.

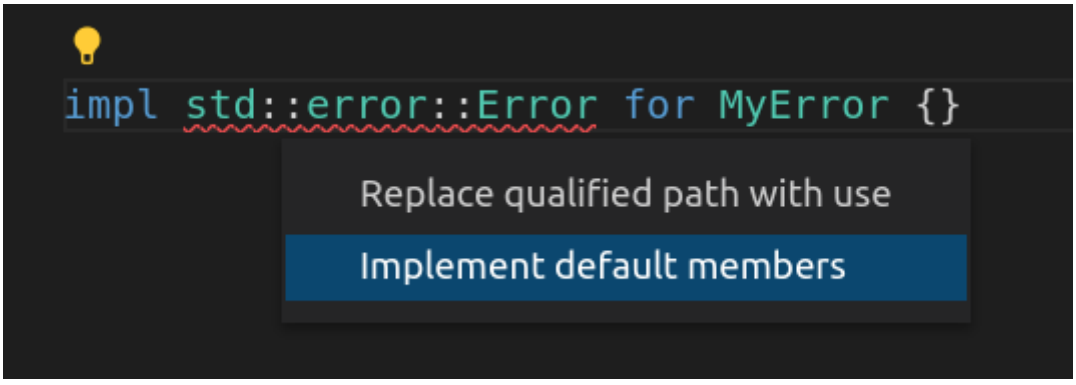
Creating your own error type gives you more control over what you want to expose externally or handle explicitly. Error types can be structs or enums. Custom errors that account for multiple errors will frequently be an enum or involve an enum internally (often called `ErrorKind`).

```rust
enum MyError {}
```

A good Rust citizen produces errors that implement the `Error` trait, which you can start by writing:

```rust
impl std::error::Error for MyError {}
```

Have you committed VS Code's `Quick fix` we talked about it in Chapter 9 to muscle memory yet?



If so then that red squiggly line might have triggered you to go ahead and run the *Implement default members* action.

```
impl std::error::Error for MyError {
  fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
    None
  }

  fn type_id(&self, _: private::Internal) -> std::any::TypeId
  where
    Self: 'static,
  {
    std::any::TypeId::of::<Self>()
  }

  fn backtrace(&self) -> Option<&std::backtrace::Backtrace> {
    None
  }

  fn description(&self) -> &str {
    "description() is deprecated; use Display"
  }

  fn cause(&self) -> Option<&dyn std::error::Error> {
    self.source()
  }
}
```

Yikes. Good news, though: you can delete it. We don't need it. Those defaults are fine. The red squiggly line was actually because the `Error` trait has two supertraits, `Display` and `Debug`, that need to be implemented.

NOTE A supertrait refers to a trait that is a "superset" of another trait. Supertraits confused me at first for two reasons.

1. *Super [anything]* invokes imagery of something out of the ordinary, something special. They're not.
2. In programming, `super` is frequently coupled with inheritance which Rust tells you time and time again it doesn't have.

The important thing to remember is that a supertrait is an additional, required trait. You must implement the supertraits in addition to the desired trait. We declare a trait with supertraits when we want to be able to use the supertrait's methods from within our trait.

Implementing `Debug` and `Display` is straightforward and similar to what we've seen before:

```
#[derive(Debug)]
enum MyError {}

impl std::error::Error for MyError {}

impl std::fmt::Display for MyError {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    write!(f, "Error!") // We have nothing useful to display yet.
  }
}
```

NOTE In this guide I frequently prefix items with their full namespace, i.e. `std::fmt::Display` vs `Display`. That's not necessary. I do it as a compromise between clarity and terseness.

After changing all of our `Result`s to return `MyError`, our code now looks like this:

```
use std::fs::read_to_string;

fn main() -> Result<(), MyError> {
  let html = render_markdown()?;
  println!("{}", html);
  Ok(())
}

fn render_markdown() -> Result<String, MyError> {
  let file = std::env::var("MARKDOWN")?;
  let source = read_to_string(file)?;
  Ok(markdown::to_html(&source))
}

#[derive(Debug)]
enum MyError {}

impl std::error::Error for MyError {}

impl std::fmt::Display for MyError {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    write!(f, "Error!")
  }
}
```

This code doesn't yet compile though. Rust outputs two errors, both of which are the same: `? coudn't convert the error to `MyError``

```
[snipped]
error[E0277]: `?` couldn't convert the error to `MyError`
   --> crates/day-14/custom-error-type/src/main.rs:10:39
    |
9   | fn render_markdown() -> Result<String, MyError> {
    |                         ---------------------- expected `MyError` because of
this
10  |   let file = std::env::var("MARKDOWN")?;
    |                                       ^ the trait `From<VarError>` is not
implemented for `MyError`
    |
    = note: the question mark operation (`?`) implicitly performs a conversion on the
error value using the `From` trait
    = note: required because of the requirements on the impl of
`FromResidual<Result<Infallible, VarError>>` for `Result<String, MyError>`
note: required by `from_residual`
[snipped]
```

Just because we have a custom error type doesn't mean that Rust knows how to convert other errors into it. The helper text shows us just what we need to do, though. We need to implement `From<env::VarError>` and `From<io::Error>` for `MyError`.

The `From`, `Into`, `TryFrom`, and `TryInto` traits

The `From`, `Into`, `TryFrom`, and `TryInto` traits are the root of many magical conversions. Whenever you see `.into()`, you're (usually) seeing the result of implementing one or several of these traits.

Implementing `From` gives you the inverse `Into` for free. `TryFrom` does the same for `TryInto`. The `Try*` traits are for conversions that can fail. They return a `Result`.

The implementations for `MyError` are below. Notice that we're adding variants to `MyError` to denote the error kind and also that our `IOError` variant wraps the original `std::io::Error`.

```
#[derive(Debug)]
enum MyError {
  EnvironmentVariableNotFound,
  IOError(std::io::Error),
}

impl From<std::env::VarError> for MyError {
  fn from(_: std::env::VarError) -> Self {
    Self::EnvironmentVariableNotFound
  }
}

impl From<std::io::Error> for MyError {
  fn from(value: std::io::Error) -> Self {
    Self::IOError(value)
  }
}
```

The complete implementation is below. Take note that we fleshed out the `Display` implementation now that we have variants to distinguish from:

```rust
use std::fs::read_to_string;

fn main() -> Result<(), MyError> {
  let html = render_markdown()?;
  println!("{}", html);
  Ok(())
}

fn render_markdown() -> Result<String, MyError> {
  let file = std::env::var("MARKDOWN")?;
  let source = read_to_string(file)?;
  Ok(markdown::to_html(&source))
}

#[derive(Debug)]
enum MyError {
  EnvironmentVariableNotFound,
  IOError(std::io::Error),
}

impl From<std::env::VarError> for MyError {
  fn from(_: std::env::VarError) -> Self {
    Self::EnvironmentVariableNotFound
  }
}

impl From<std::io::Error> for MyError {
  fn from(value: std::io::Error) -> Self {
    Self::IOError(value)
  }
}

impl std::error::Error for MyError {}

impl std::fmt::Display for MyError {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    match self {
      MyError::EnvironmentVariableNotFound => write!(f, "Environment variable not
found"),
      MyError::IOError(err) => write!(f, "IO Error: {}", err.to_string()),
    }
  }
}
```

It's a lot of code just to use a question mark that's supposed to make our lives easier...

Option 3: Use a crate

Every Rust programmer deals with errors and there's loads of precedent out there. There's no need to reinvent the wheel at this stage of your Rust journey. It's much easier to leave it to a crate.

thiserror

thiserror (crates.io) gives you all of Option 2 with less headache and more functionality. The code below is a complete implementation that mimics the behavior in our custom error example.

```
use std::fs::read_to_string;

fn main() -> Result<(), MyError> {
  let html = render_markdown()?;
  println!("{}", html);
  Ok(())
}

fn render_markdown() -> Result<String, MyError> {
  let file = std::env::var("MARKDOWN")?;
  let source = read_to_string(file)?;
  Ok(markdown::to_html(&source))
}

#[derive(thiserror::Error, Debug)]
enum MyError {
  #[error("Environment variable not found")]
  EnvironmentVariableNotFound(#[from] std::env::VarError),
  #[error(transparent)]
  IOError(#[from] std::io::Error),
}
```

error-chain

NOTE	error-chain is no longer maintained and is marked as deprecated. It's still heavily relied upon and works fine the cases where I've used it. It makes basic error handling so simple that I think it is still worth mentioning. Getting passed the early frustration with error handling is more important than finding the perfect crate right away.

Another great option is error-chain (crates.io). error-chain gives you a lot more options and makes creating errors as easy as:

```
error_chain::error_chain!{}
```

Really, that's it. You get an Error struct, an ErrorKind enum, a custom Result type aliased to return your Error, and more.

Below is a sample implementation for our example program.

```rust
use std::fs::read_to_string;

error_chain::error_chain! {
  foreign_links {
    EnvironmentVariableNotFound(::std::env::VarError);
    IOError(::std::io::Error);
  }
}

fn main() -> Result<()> {
  let html = render_markdown()?;
  println!("{}", html);
  Ok(())
}

fn render_markdown() -> Result<String> {
  let file = std::env::var("MARKDOWN")?;
  let source = read_to_string(file)?;
  Ok(markdown::to_html(&source))
}
```

Honorable mention: `anyhow`

The author of `thiserror` also publishes another popular error crate called `anyhow`. His words on the difference between `anyhow` and `thiserror`:

> **NOTE** Use thiserror if you care about designing your own dedicated error type(s) so that the caller receives exactly the information that you choose in the event of failure. This most often applies to library-like code. Use Anyhow if you don't care what error type your functions return, you just want it to be easy. This is common in application-like code._

I've used `anyhow` frequently and agree with the distinction above. `anyhow` is great for building command line utilities and other projects that won't be used like a library.

15.1.3. Additional reading

- The Rust Book: ch 9.02 - Recoverable Errors with Result
- Rust by Example: Multiple Error Types

Other crates

There are many other popular crates that make error handling less cumbersome. I haven't used any of these in large projects and can't give an opinion.

- Snafu (crates.io.)
- quick-error (crates.io)
- failure (crates.io)

- err-derive (crates.io)

15.2. Wrap-up

Rust makes errors a priority. Once you start respecting them the way Rust forces you too, you'll understand why. Robust error handling is one of the most valuable things you can take back to your JavaScript projects. You'll learn how to isolate code that can fail and generate more meaningful error messages and fallbacks.

You can't go wrong with using `thiserror` or `error-chain` for libraries. I use `anyhow` in my tests and for CLI projects frequently. They are all quality options and will turn error handling into one of the most frustrating parts of Rust into one of the things you love most.

16. Closures

16.1. Introduction

Closures are a natural part of JavaScript. It's hard to imagine what programming is like without them. Luckily, you don't have to. The behavior of Rust's closures is similar enough to JavaScript's that you will be able to retain most of what you're comfortable with.

NOTE	Closures are defined as functions that retain references to (enclose) its surrounding state. I use the term "closure" here as a general term to mean an anonymous function, regardless of whether or not it references external variables.

16.2. Closure syntax comparison

NOTE	If you haven't been following along with the code repository, now is a good time to start. This day's code is available in Chapter 15 of the project on github.

This section maps JavaScript/TypeScript closures to the equivalent Rust syntax without much explanation. If you get lost, please reach out. Your perspective on what is confusing will help make this book better. Don't hesitate to submit issues or pull requests!

16.2.1. Basic closure syntax

This closure prints `Hi! I'm in a closure`.

```
let closure = () \=> {
  console.log("Hi! I'm in a closure");
};
closure();
```

```
let closure = || {
  println!("Hi! I'm in a closure");
};
closure();
```

```
Hi! I'm in a closure
```

NOTE	Rust uses pipes instead of parentheses for arguments and does not have a separator between the arguments and the body.

16.2.2. Closures with a single expression body

These closures show how you can omit the curly braces `{}` for closures that consist of a single

expression.

```
let double = (num: number) \=> num + num;
let num = 4;
console.log(`+${num} + ${num} = ${double(num)}+`);
```

```
let double = |num: i64| num + num;
let num = 4;
println!("{} + {} = {}", num, num, double(num));
```

```
4 + 4 = 8
```

NOTE	Both JavaScript and Rust can omit the curly braces if the body consists of a single expression.

16.2.3. Closures referencing external variables

A proper closure references variables from its parent's scope. That's no problem in Rust.

```
let name = "Rebecca";
closure = () \=> {
  console.log(`+Hi, ${name}.+`);
};
closure();
```

```
let name = "Rebecca";
let closure = || {
  println!("Hi, {}.", name);
};
closure();
```

```
Hi, Rebecca.
```

NOTE	Mutable variables need a mutable closure! The state of your closure is part of the closure. If you mutate a variable in a closure then the closure itself must be made mutable.

These closures increment a counter when executed.

```
let counter = 0;
closure = () \=> {
  counter += 1;
  console.log(`+This closure has a counter. I've been run ${counter} times.+`);
};
closure();
closure();
closure();
console.log(`+The closure was called a total of ${counter} times+`);
```

```
let mut counter = 0;

let mut closure = || {
  counter += 1;
  println!(
    "This closure has a counter. I've been run {} times.",
    counter
  );
};
closure();
closure();
closure();
println!("The closure was called a total of {} times", counter);
```

```
This closure has a counter. I've been run 1 times.
This closure has a counter. I've been run 2 times.
This closure has a counter. I've been run 3 times.
The closure was called a total of 3 times
```

16.2.4. Returning a closure

Generating closures dynamically is straightforward once you get over the nuances in the different traits. The make-adder functions take in an addend and generate a closure that takes in a second number and sums the enclosed value with the passed value.

```
function makeAdder(left: number): (left: number) \=> number {
  return (right: number) \=> {
    console.log(`+${left} + ${right} is ${left + right}+`);
    return left + right;
  };
}

let plusTwo = makeAdder(2);
plusTwo(23);
```

```
fn make_adder(left: i32) -> impl Fn(i32) -> i32 {
  move |right: i32| {
    println!("{} + {} is {}", left, right, left + right);
    left + right
  }
}

let plus_two = make_adder(2);
plus_two(23);
```

```
2 + 23 is 25
```

The `Fn`, `FnMut`, and `FnOnce` traits

Functions come in three flavors.

- `Fn`: a function that immutably borrows any variables it closes over.
- `FnMut`: a function that mutably borrows variables it closes over.
- `FnOnce`: a function that consumes (loses ownership of) of its values and thus can only be run once, e.g.

```
let name = "Dwayne".to_owned();
let consuming_closure = || name.into_bytes();
let bytes = consuming_closure();
let bytes = consuming_closure(); // This is a compilation error
```

The `move` keyword

The move keyword tells Rust that the following block or closure takes ownership of any variables it references. It's necessary above because we're returning a closure that references `left` which would normally be dropped when the function ends. When we `move` it into the closure, we can return the closure without issue.

16.2.5. Composing functions

The `compose` function takes two functions and returns a closure that runs both and pipes the output of one into the other.

Each input closure takes one argument of the generic type `T` and returns a value also of type `T`. The first of the two closures is the `plus_two` closure from above. The second closure, `times_two`, multiplies its input by two.

The generated closure, `double_plus_two`, composes the original two into one.

```
function compose<T>(f: (left: T) \=> T, g: (left: T) \=> T): (left: T) \=> T { return
(right: T) \=> f(g(right)); }

let plusTwo = makeAdder(2); // ← makeAdder from above
let timesTwo = (i: number) \=> i * 2;
let doublePlusTwo = compose(plusTwo, timesTwo);
console.log(`+${10} * 2 + 2 = ${doublePlusTwo(10)}+`);
```

```
fn compose<T>(f: impl Fn(T) -> T, g: impl Fn(T) -> T) -> impl Fn(T) -> T {
  move |i: T| f(g(i))
}

let plus_two = make_adder(2);  // ← make_adder from above
let times_two = |i: i32| i * 2;
let double_plus_two = compose(plus_two, times_two);
println!("{} * 2 + 2 = {}", 10, double_plus_two(10));
```

```
10 * 2 + 2 = 22
```

16.2.6. Regular function references

This section shows how you can treat any function as a first-class citizen in Rust.

```
function regularFunction() {
  console.log("I'm a regular function");
}

let fnRef = regularFunction;
fnRef();
```

```
fn regular_function() {
  println!("I'm a regular function");
}

let fn_ref = regular_function;
fn_ref();
```

```
I'm a regular function
```

16.2.7. Storing closures in a struct

Storing functions can be a little trickier due to the different `Fn*` traits and the `dyn [trait]` behavior.

This code creates a class or struct that you instantiate with a closure. You can then call `.run()` from the resulting instance to execute the stored closure.

```
class DynamicBehavior<T>{ closure: (num: T) \=> T; constructor(closure: (num: T) \=>
T) { this.closure = closure; } run(arg: T): T { return this.closure(arg); } }

let square = new DynamicBehavior((num: number) \=> num * num);
console.log(`+${5} squared is ${square.run(5)}+`);
```

```
struct DynamicBehavior<T> {
  closure: Box<dyn Fn(T) -> T>,
}

impl<T> DynamicBehavior<T> {
  fn new(closure: Box<dyn Fn(T) -> T>) -> Self {
    Self { closure }
  }
  fn run(&self, arg: T) -> T {
    (self.closure)(arg)
  }
}

let square = DynamicBehavior::new(Box::new(|num: i64| num * num));
println!("{} squared is {}", 5, square.run(5))
```

NOTE	Remember we can't use `impl [trait]` outside of a function's parameters or return value, so to store a closure we need to store it as a `dyn [trait]`. Also remember that `dyn [trait]` is unsized and Rust doesn't like that. We can `Box` it to move passed Rust's complaints (see Chapter 14 : What's a box?).

16.2.8. Additional reading

- The Rust Book: ch 13.01 - Closures
- The Rust Book: ch 19.05 - Advanced Functions and Closures
- Rust by Example: Closures
- Rust Reference: Closure expressions

16.3. Wrap-up

Rust's closures are not as terrifying as some people make them out to be. You will eventually get to some gotchas and hairy parts, but we'll tackle those when we deal with async. First though, we've put off lifetimes for long enough. We'll get deeper into Rust's borrow checker in the next chapter before moving on to Arrays, iterators, async, and more.

17. Lifetimes, References, and 'static

17.1. Introduction

Lifetimes are Rust's way of avoiding dangling references, pointers to memory that has been deallocated. To a user, this might look like a failed operation or a crashed program. To a malicious actor, it's an opening. Accessing the memory behind a dangling reference produces undefined behavior. That is another way of saying it produces behavior that is open for others to define.

NOTE

Bugs like dangling references account for the vast majority of major software vulnerabilities. An analysis from the Chromium team showed that 70% of high severity issues were due to memory safety bugs. If this sort of thing interests you, the phrase you need to search for is "use after free" or UAF vulnerabilities. This is a good introduction to how they're exploited in browsers. It's *fascinating*.

This might sound a little nutty if you've never used a language like C where you manage your own memory. Languages like JavaScript and Java manage memory for you and use garbage collectors to avoid dangling references. Garbage collectors track every time you take a reference to data. They keep that memory allocated as long as the reference count is greater than zero.

Garbage collection was a revolutionary invention *63 years ago*. But it comes with a cost. In extreme cases, garbage collection can freeze your application for full *seconds*. Garbage-minded applications rarely see that high of an impact, but the GC always looms in the background. Like a vampire.

Rust doesn't use a garbage collector yet still avoids dangling references. You get the raw power of a language like C with the safety of a garbage collected language like JavaScript.

NOTE

There are dozens, if not hundreds, of excellent articles written about Rust lifetimes. This chapter will address some of the confusion that arises once you're in the weeds. The additional reading in this section should be considered a prerequisite.

17.2. Lifetimes vs lifetime annotations

You will frequently come across posts, questions, and answers where the phrase "lifetime annotations" is shortened to "lifetimes" which only adds confusion.

A ***lifetime*** is a construct within Rust's borrow checker. Every value has a point where its created and a point where its dropped. That's its lifetime.

A ***lifetime annotation*** is Rust syntax you can add to a reference to give its lifetime a named tag. You must use lifetime annotations in situations where there are multiple references and Rust can't disambiguate them on its own.

Every time you read a sentence like "you must specify a lifetime" or "give the reference a lifetime," it's referring to a lifetime annotation. You can't give a value a new lifetime.

17.3. Lifetime elision

Every reference has a lifetime, even if you don't see annotations. Just because you're not writing annotations on your references doesn't mean you're avoiding them.

This function:

```
fn omits_annotations(list: &[String]) -> Option<&String> {
  list.get(0)
}
```

is equivalent to this function:

```
fn has_annotations<'a>(list: &'a [String]) -> Option<&'a String> {
  list.get(1)
}
```

Both compile and act like you'd expect.

```
fn main() {
  let authors = vec!["Samuel Clemens".to_owned(), "Jane Austen".to_owned()];
  let value = omits_annotations(&authors).unwrap();
  println!("The first author is '{}'", value);
  let value = has_annotations(&authors).unwrap();
  println!("The second author is '{}'", value);
}
```

```
The first author is 'Samuel Clemens'
The second author is 'Jane Austen'
```

The reason Rust can do this for you is that there is *one* reference in the arguments and *one* reference in the return value. The lifetime of the returned reference therefore *must* be the same as the lifetime in the passed reference.

This is Rust trying to be helpful. It handles the lifetime annotations for simple cases. The problem is: if Rust handles the trivial cases, the first ones you have to deal with are — by definition — non-trivial. It can feel daunting. Don't stress about it. You've got this. The best way to go forward is to go backward. Write out by hand what Rust was doing for you. It'll give you a feel for what is expected.

17.4. The `'static` lifetime

`'static` is described in great detail in every corner of Rust's community. The time spent explaining `'static` makes it seem more complex than it is. I'll try to be as succinct as possible.

There are two ways you'll typically see `'static` used.

1. As the explicit lifetime annotation on a reference, e.g.:

```
fn main() {
  let mark_twain = "Samuel Clemens";
  print_author(mark_twain);
}
fn print_author(author: &'static str) {
  println!("{}", author);
}
```

1. Or as the lifetime bounds on generic type parameters, e.g.:

```
fn print<T: Display + 'static>(message: &T) {
  println!("{}", message);
}
```

`&'static` means this reference ***is*** valid for the rest of the program. The data it's pointing to ***will not*** move ***nor*** change. It will always be available. This is why string literals are `&'static`. The data is baked into the program and will never be dropped.

NOTE	`&'static` says nothing about the lifetime of the variable holding the reference, however. The program below exemplifies this behavior.

The `get_memory_location()` function returns the pointer and length of a literal string, a `&'static str`, that it immediately drops. The `get_str_at_location()` function takes a pointer plus a length and attempts to read that memory location back as a `&str`. This works just fine, though we're using dangerous functions and have to mark them as such with the `unsafe` keyword.

Uncomment the final line in `main()` to see why they are unsafe.

```
use std::{slice::from_raw_parts, str::from_utf8_unchecked};

fn get_memory_location() -> (usize, usize) {
  let string = "Hello World!";
  let pointer = string.as_ptr() as usize;
  let length = string.len();
  (pointer, length)
  // `string` is dropped here.
  // It's no longer accessible, but the data lives on.
}

fn get_str_at_location(pointer: usize, length: usize) -> &'static str {
  // Notice the `unsafe {}` block. We can't do things like this without
  // acknowledging to Rust that we know this is dangerous.
  unsafe { from_utf8_unchecked(from_raw_parts(pointer as *const u8, length)) }
}

fn main() {
  let (pointer, length) = get_memory_location();
  let message = get_str_at_location(pointer, length);
  println!(
    "The {} bytes at 0x{:X} stored: {}",
    length, pointer, message
  );
  // If you want to see why dealing with raw pointers is dangerous,
  // uncomment this line.
  // let message = get_str_at_location(1000, 10);
}
```

```
The 12 bytes at 0x562037200057 stored: Hello World!
```

On the other hand, adding `'static` as a bound is like telling Rust "I want a type that *could* last forever, if I needed it to." It's *not* telling Rust you only want data that does live forever.

In friendly terms: `&'static` **!==** `T: 'static`

The code below illustrates how a type like `String` in the second block can satisfy a `'static` constraint for the `static_bound()` function yet not retain the same properties as the `&'static` references in the first block.

```
use std::fmt::Display;

fn main() {
  let r1;
  let r2;
  {
    static STATIC_EXAMPLE: i32 = 42;
    r1 = &STATIC_EXAMPLE;
    let x = "&'static str";
    r2 = x;
  }
  println!("&'static i32: {}", r1);
  println!("&'static str: {}", r2);

let r3;

{
    let string = "String".to_owned();

 static_bound(&string); // This is *not* an error
 r3 = &string; // *This* is   }   println!("{}", r3); }

fn static_bound<T: Display + 'static>(t: &T) {
  println!("{}", t);
}
```

```
error[E0597]: `string` does not live long enough
  --> crates/day-16/static/src/main.rs:21:10
   |
21 |     r3 = &string;
   |          ^^^^^^^ borrowed value does not live long enough
22 |   }
   |   - `string` dropped here while still borrowed
23 |   println!("{}", r3);
   |                  -- borrow later used here

For more information about this error, try `rustc --explain E0597`.
```

NOTE The project day-16-static-bounds in the code repository further illustrates the differences between `&'static` and `T: 'static`.

While the two usages are related, the spirit behind each is different.

As a rule: if you need to add a `&'static` to make things work, you might want to rethink things. If you need to add a `'static` bound to make Rust happy (e.g. `T: 'static` or `+ 'static`), it's probably OK.

NOTE	Because I was curious: Rust's standard library has 48 instances of `&'static` (minus `&'static str`) and 112 instances of `'static` as a constraint.

17.4.1. Additional reading

- The Rust Book: ch 10.03 - Validating References with Lifetimes
- Rust by Example: Lifetimes
- Rust Reference: Trait and lifetime bounds
- Rust Reference: Lifetime elision
- Rustonomicon: Lifetimes
- Common Rust Lifetime Misconceptions
- rustviz: Rust lifetime visualizer
- Understanding lifetimes in Rust

17.5. Wrap-up

This chapter went through five iterations before ending with this version. One of the biggest source of headaches with generic types, lifetimes, and references is how you use them with popular third party libraries and async code. We haven't yet hit those topics yet. We'll circle back around to common errors and issues when we move into more complex code.

18. Arrays, Loops, and Iterators

18.1. Introduction

Translating common JavaScript use cases for Arrays and loops requires learning some new concepts and data types. You'll also need to get comfortable reading and writing more code than you're used to. In some ways, Rust is more succinct than JavaScript. In others, what could have been a one-liner in JavaScript may be ten times more code in Rust.

We touched on `Vec` and `VecDeque` in Chapter 7 which will be your go-to list structures. The next hurdle is wrapping your head around Iterators. Many of the Array methods you're used to using in JavaScript exist in Rust, but they are wrapped in a lazy `Iterator` construct.

18.2. Recap: `vec![]`, `Vec`, and `VecDeque`

Actual Rust arrays must have a known length with all elements initialized. You can mutate the internal elements, but you can't grow or shrink them.

This won't work.

```
let mut numbers = [1, 2, 3, 4, 5];
numbers.push(7); // no method named `push` found for array `[{integer}; 5]`
println!("{:?}", numbers);
```

To get the flexible lists you're used to, you need a `Vec` or `VecDeque`. `Vec` is to JavaScript arrays what `String` is to JavaScript strings. `Vec`'s can only grow and shrink at the end. `VecDeque` can grow or shrink from either direction.

Creating a `Vec` is easy with the `vec![]` macro. Add it to your cheatsheet. You'll use it frequently.

```
let mut numbers = vec![1, 2, 3, 4, 5];  // 🔑 Notice the vec! macro
numbers.push(7);
println!("", numbers);
```

18.3. Loops

18.3.1. `for ( … ; … ; … )`

Rust does not have a for loop like this for good reason. It's cumbersome syntax for a general loop that mostly gets used one way:

1. We initialize a single variable (e.g. `i`) and assign it the minimum value in a range (e.g. `0`).
2. We test if the variable is greater than the maximum value in the range (e.g. `i < max`).
3. Every loop we increment the variable by `1`.

Rust generalizes this use case into its `for…in` expression combined with the [range operator](https://doc.rust-lang.org/std/ops/struct.Range.html) (e.g. `0..10`).

```
let max = 4;
for (let i = 0; i < max; i++) {
  console.log(i);
}
```

```
let max = 4;
for i in 0..max {
  println!("{}", i);
}
```

```
1
2
3
```

18.3.2. `for…in`

Rust doesn't have the same kinds of `Object` types that JavaScript has. There's no real way of iterating over the properties of arbitrary objects. Rust does have Map types like `HashMap` though (see Chapter 8). You can use the `.keys()` method to get an iterator over the map's keys.

> **NOTE** `.keys()` visits keys in arbitrary order. You don't get a say in this.

```
let obj: any = {
  key1: "value1",
  key2: "value2",
};
for (let prop in obj) {
  console.log(`+${prop}: ${obj[prop]}+`);
}
```

```
let obj = HashMap::from([
  ("key1", "value1"),
  ("key2", "value2")
]);
for prop in obj.keys() {
  println!("{}: {}", prop, obj.get(prop).unwrap());
}
```

```
key1: value1
key2: value2
```

18.3.3. `for⋯of`

JavaScript's `for⋯of` translates almost 1-to-1 with Rust's `for⋯in`

```
let numbers = [1, 2, 3, 4, 5];
for (let number of numbers) {
  console.log(number);
}
```

```
let numbers = [1, 2, 3, 4, 5];
for number in numbers {
  println!("{}", number);
}
```

```
1
2
3
4
5
```

18.3.4. `while` loops

While loops are straightforward to understand, but there are common cases that Rust handles better than a straight translation.

`while (!done)`…

One such case is using a `while` to loop until something is "done," whatever done might mean.

```
let obj = {
  data: ["a", "b", "c"],
  doWork() {
    return this.data.pop();
  },
};

let data;
while ((data = obj.doWork())) {
  console.log(data);
}
```

The Rust counterpart uses `while let` syntax to match against the return value and conditonally continue the loop. The loop continues as long as `.doWork()` returns a `Some()`. You can do this similarly with `Result`s and `Ok` and any other type.

```
struct Worker {
  data: Vec<&'static str>,
}
impl Worker {
  fn doWork(&mut self) -> Option<&'static str> {
    self.data.pop()
  }
}
let mut obj = Worker {
  data: vec!["a", "b", "c"],
};

while let Some(data) = obj.doWork() {
  println!("{}", data);
}
```

```
c
b
a
```

18.3.5. do ⋯ while

Rust has no `do⋯while` loop. You can get similar behavior with `loop` described next.

`while (true) ⋯`

Rust's `loop` expression simply loops forever. It's handier than you might think at first glance, and much more intuitive than `while (true)`.

```
let n = 0;

while (true) {
  n{pp};
  if (n > 3) break;
  else console.log(n);
}
```

```
let mut n = 0;
loop {
  n += 1;
  if n > 3 {
    break;
  }
}
println!("Finished. n={}", n);
```

```
Finished. n=4
```

18.4. Labels, `break`, `continue`

In Rust, labels work the same way as they do in JavaScript, with the only difference being Rust labels are prefixed with an apostrophe.

```
outer: while (true) {
  while (true) {
    break outer;
  }
}
```

```
'outer: loop {
  loop {
    break 'outer;
  }
}
```

18.5. `break` & `loop` expressions

`loop` blocks are expressions themselves and can return values. This is a better alternative than initializing a variable outside a loop just so you can update it internally.

```
let value = loop {
  if true {
    break "A";
  } else {
    break "B";
  }
};
println!("Loop value is: {}", value);
```

```
Loop value is: A
```

18.6. Intro to Rust Iterators

Iterators are how Rust deals with operations on a sequence. Iterators can be chained to produce more iterators. Unlike JavaScript's iteration methods, Rust iterators are lazy. They don't execute until you call a method that needs a value.

All iterators implement the Iterator trait which gives each a similar interface. This trait is different than some of the more basic Rust traits. It has an associated type named `Item` and is a placeholder for the type of the elements being iterated over. You don't need to worry about it much until you start trying to build your own iterators or return them from functions.

NOTE	Associated types in traits are similar to generics. They are a placeholder for a type that the implementer will define. To learn more about how they're different, read The Rust Book, ch 19.03: Advanced Traits

18.6.1. How to get and use iterators

Because a `Vec` isn't an iterator itself, we have to call a method to make it one. And because `Iterator`s are lazy, we have to call a method to get any value at all out of them. This means we have to add two method calls every time we want to iterate and return a value. It's a lot of noise:

```
let list = vec![1, 2, 3];
let doubled: Vec<_> = list
  .iter()
  .map(|num| num * 2)
  .collect();
println!("{:?}", doubled);
```

```
[2, 4, 6]
```

The `.iter()` method on many structures returns an `Iterator`, while the `Iterator` method `.collect()` consumes the rest of an iterator and returns a single value.

To get a single value out of an iterator, you'd use the `.next()` method.

NOTE	**error[E0282]: type annotations needed**

When you start using `.collect()` you will probably run into the error : `error[E0282]: type annotations needed` right away.

```
let list = vec![1, 2, 3];
let doubled = list.iter().map(|num| num * 2).collect();
```

```
error[E0282]: type annotations needed
  --> crates/day-17/iterators/src/main.rs:13:7
   |
13 |   let doubled = list.iter().map(|num| num * 2).collect();
   |       ^^^^^^^ consider giving `doubled` a type

For more information about this error, try `rustc --explain E0282`.
```

When I started, I couldn't figure out why I needed to annotate my types. Rust knew the types going into `map()` and knew the types coming out. Why do I need to annotate them like this?

```
let list = vec![1, 2, 3];
let doubled: Vec<i32>= list.iter().map(|num| num * 2).collect();
```

Well, you don't. It turns out that Rust does indeed know the type of its elements, but it has no knowledge of what its new wrapper should be. It's not the `i32` part of the type that Rust needs annotated, it's the `Vec<>` part. Just because we started with a `Vec` doesn't mean we will always want one when we're done.

When Rust knows one type but not another, you can omit it with an underscore (`_`), e.g `Vec<_>`,

```
let doubled: Vec<_>= list.iter().map(|num| num * 2).collect();
```

NOTE **error[E0596]: cannot borrow ⋯ as mutable, as it is behind a `&` reference**

```
let list = vec!["garbage".to_owned(), "data".to_owned()];
list.iter().for_each(|garbage| garbage.clear()); // .clear() mutates its self
```

```
error[E0596]: cannot borrow `*garbage` as mutable, as it is behind a `&` reference
  --> crates/day-17/iterators/src/main.rs:11:34
   |
11 |   list.iter().for_each(|garbage| garbage.clear());
   |                         -------  ^^^^^^^^^^^^^^^
   |                                  `garbage` is a `&` reference, so the data it
   |                                   refers to cannot be borrowed as mutable
   |                         |
   |                         help: consider changing this to be a mutable reference:
`&mut String`

For more information about this error, try `rustc --explain E0596`.
```

But *you* can't change `garbage` to `garbage: &mut String`, it causes a different compile error. This time Rust complains of a signature mismatch on the closure passed `for_each()`.

So what do you do? Instead of `.iter()` you use `.iter_mut()`.

`.iter()` immutably borrows elements, `.iter_mut()` mutably borrows them. When you are confronted with this error in an API you don't control, look for `*_mut()` methods that complement the ones you're already using.

18.7. Translating Array.prototype methods

18.7.1. `.filter()`

Iterator's `.filter()` method produces another iterator and has some tricky behavior explained in the note below.

```
let numbers = [1, 2, 3, 4, 5];
let even = numbers.filter((x) \=> x % 2 === 0);
console.log(even);
```

```
let numbers = [1, 2, 3, 4, 5];
let even: Vec<_> = numbers.iter().filter(|x| *x % 2 == 0).collect();
println!("{:?}", even);
```

```
[2, 4]
```

NOTE

Did you notice the asterisk (*) in front of the `x` in the filter body? That's because `.filter()` takes a reference and most iterators iterate over references so we have to dereference the double reference to get a reference to our integer. Yuck, but that's life. It's documented on Iterator but it's not an uncommon to find elsewhere.

18.7.2. `.find()`

`.find(predicate)` is essentially a `.filter(predicate).next()`. It consumes the iterator until your predicate returns true and returns that value.

```
let numbers = [1, 2, 3, 4, 5];
let firstEven = numbers.find((x) \=> x % 2 === 0);
console.log(firstEven);
```

```
let numbers = [1, 2, 3, 4, 5];
let first_even = numbers.iter().find(|x| *x % 2 == 0);
println!("{:?}", first_even.unwrap());
```

```
2
```

NOTE You can store the iterator and call `.find()` multiple times. You can't do *that* in JavaScript.

```
let numbers = [1, 2, 3, 4, 5];
let mut iter = numbers.iter(); // Note, our iter is mut
let first_even = iter.find(|x| *x % 2 == 0);
println!("{:?}", first_even.unwrap());
let second_even = iter.find(|x| *x % 2 == 0);
println!("{:?}", second_even.unwrap());
```

```
2
4
```

18.7.3. `.forEach()`

`.for_each()` consumes the iterator immediately. You'de use it at the end of an iterator chain to operate on each element. Using a plain loop is usually a more readable option.

```
let numbers = [1, 2, 3];
numbers.forEach((x) \=> console.log(x));
```

```
let numbers = [1, 2, 3];
numbers.iter().for_each(|x| println!("{}", x));
```

```
1
2
3
```

18.7.4. `.join()`

`.join()` works on arrays and `Vec`s without needing an iterator.

```
let names = ["Sam", "Janet", "Hunter"];
let csv = names.join(", ");
console.log(csv);
```

```
let names = ["Sam", "Janet", "Hunter"];
let csv = names.join(", ");
println!("{}", csv);
```

```
Sam, Janet, Hunter
```

18.7.5. `.map()`

`.map()` is another `Iterator` method that returns an `Iterator`.

```
let list = [1, 2, 3];
let doubled = list.map((x) \=> x * 2);
console.log(doubled);
```

```
let list = vec![1, 2, 3];
let doubled: Vec<_> = list.iter().map(|num| num * 2).collect();
println!("{:?}", doubled)
```

```
[2, 4, 6]
```

18.7.6. `.push()` and `.pop()`

While you can use `.iter()` on regular arrays, `.push()` and `.pop()` are only available on `Vec` types.

```
let list = [1, 2];
list.push(3);
console.log(list.pop());
console.log(list.pop());
console.log(list.pop());
console.log(list.pop());
```

```
3
2
1
undefined
```

```rust
let mut list = vec![1, 2];
list.push(3);
println!("", list.pop());
println!("", list.pop());
println!("", list.pop());
println!("", list.pop());
```

```
Some(3)
Some(2)
Some(1)
None
```

NOTE If you use a `VecDeque`, `.push()`/`.pop()` become `.push_back()` and `.pop_back()`

18.7.7. `.shift()` & `.unshift()`

You can't get the same behavior as `.shift()` and `.unshift()` with a `Vec`. `Vec`s only grow from the back. You need a `VecDeque` (Double Ended QUEue) to push/pop from the front of a list.

```javascript
let list = [1, 2];
list.unshift(0);
console.log(list.shift());
console.log(list.shift());
console.log(list.shift());
console.log(list.shift());
```

```
0
1
2
undefined
```

```rust
let mut list = VecDeque::from([1, 2]);
list.push_front(0);
println!("", list.pop_front());
println!("", list.pop_front());
println!("", list.pop_front());
println!("", list.pop_front());
```

```
Some(0)
Some(1)
Some(2)
None
```

18.7.8. How to return an Iterator

It's bad form to use `.collect()` to return a specific data structure when you could return the iterator itself. Returning an iterator keeps things flexible and retains the lazy evaluation Rust programmers expect. Since the basic `Iterator` is a trait, we can return it the same way we've returned closures and other values in previous guides.

The data structure below is part of the day-17-names(day-17-names example project. It holds its own `Vec` of names and provides a method to search the list.

Rather than returning a `Vec<&String>`, it returns an `Iterator` of borrowed `String`s.

```
struct Names {
  names: Vec<String>, }

impl Names {
  fn search<T: AsRef<str>>(&self, re: T) -> impl Iterator<Item = &String> {
     let regex = regex::Regex::new(re.as_ref()).unwrap();
     self.names.iter().filter(move |name| regex.is_match(name))
  }
}
```

NOTE Confused about `AsRef<str>`? Head back to Chapter 12: Strings, Part 2 to brush up.

18.7.9. Additional reading

- The Rust Book: ch 13.02 - Iterators
- Rust by Example: Flow control
- Rust by Example: Vectors
- Rust by Example: Iterators
- Rust docs: Vec
- Rust docs: VecDeque
- Rust docs: Iterator

18.8. Wrap-up

Porting over our mental model of how lists and iteration works is important. If you subscribe to the functional programming style in JavaScript, you're going to have a great time in Rust. While Rust is not a purely functional language, its default behavior for things like Iterators will net you greater rewards for little effort. Iterators and eventually streams give you more control over how much processing is done and when.

19. Async in Rust

19.1. Introduction

Rust's async story has its good parts and bad parts. Futures (Rust's promises) are a core part of Rust. Actually being able to use them is not. That's weird, let's break it down.

Rust's standard library defines what an asynchronous task needs to look like with the `Future` trait. But implementing `Future` isn't enough to be "async" on its own. You need something that will manage them. You need a futures bucket that checks which futures are done and notifies what's waiting on them. You need an executor and a reactor, kind of like node.js's event loop. You don't get that with Rust. The Rust team left it to the community to decide how best to flesh out the async ecosystem. It may seem nuts, but did you know there was a time where JavaScript didn't have promises? JavaScript had this problem in reverse. It had the executor and reactor but no way to represent a task. The community had to define what a Promise was. We were left polyfilling them until they landed in ES6.

In Rust, you have to polyfill async. Maybe it'll exist in Rust core someday, but that's not relevant. The important part is figuring out which polyfills exist now and how to get started ASAP.

19.1.1. Rust async libraries

- Tokio (repo) (crates.io) (docs.rs)
- Async-std (repo) (crates.io) (docs.rs)
- Smol (repo) (crates.io) (docs.rs)

There are more, but this is already enough. Every library has their audience and there's little point debating which one is "best." What's important right now is which will be the easiest to deal with. That's Tokio. There are libraries that depend on Tokio's behavior which means you can't (easily) use them without also using Tokio's executor. It's a bit of a hostage situation. Tokio's not bad though. It has documentation, loads of community contributions, and there's a lot of code to learn from. It's just a weird situation to be in when you're used to async JavaScript and node.js.

NOTE	Did you know that there used to be many other server-side JavaScript implementations before node.js? Some of them were even single-threaded and required you to deal with blocking logic by forking. Tokio is kind of like node.js. It's an async implementation with a core set of async methods you can rely on. Smol is kind of like deno. It's newer, has an answer for interoperability, but promises to be faster and better.

19.1.2. Quickstart

Add Tokio as a dependency in your `Cargo.toml` with the `full` feature flag.

```
[dependencies]
tokio = { version = "1", features = ["full"] }
```

NOTE	Feature flags expose conditional compilation to users of a library. All tags are arbitrary, "full" doesn't mean anything special. In some libraries feature flags are used to turn on or off platform-specific code. In others like Tokio, it's used to conditionally require what amounts to sub-crates. Tokio used to be split up into many small modules. Community feedback changed that course and now we use feature flags.

When you get more comfortable, you can read up on the feature flags and trim it down to what you need.

You must starting an executor before running your futures. Tokio gives you a handy macro that sets everything up behind the scenes. You add it to `main()` and *voila*, you get a fully async Rust.

```
#[tokio::main]
async fn main()  { // Notice we write async main() now
}
```

19.1.3. async/.await

Rust has an async/await style syntax like JavaScript. Adding `async` turns a function's return value from `T` into `impl Future<Output = T>`, e.g.

```
fn regular_fn() -> String {
  "I'm a regular function".to_owned()
}

async fn async_fn() -> String { // actually impl Future<Output = String>
  "I'm an async function".to_owned()
}
```

Unlike JavaScript, the `await` syntax must be appended to an actual future. It isn't prepended nor can it be used on arbitrary values. It looks like this.

```
#[tokio::main]
async fn main() {
    let msg = async_fn().await;
}
```

Also unlike JavaScript, ***your futures don't run until you await them***.

```
#[tokio::main]
async fn main() {
    println!("One");
    let future = prints_two();
    println!("Three");
    // Uncomment and move the following line around to see how the behavior changes.
    // future.await;
}

async fn prints_two() {
    println!("Two")
}
```

```
One
Three
```

Uncommenting the line above produces

```
One
Three
Two
```

19.1.4. async blocks

Asynchronous behavior and closures are part of every developer's toolbox. You will inevitable get to a point where you try to return a closure that's also `async` and you run into this error:

```
error[E0658]: async closures are unstable
 --> src/send-sync.rs:6:15
  |
6 |     let fut = async || {};
  |               ^^^^^
  |
  = note: see issue #62290 <https://github.com/rust-lang/rust/issues/62290> for more
information
  = help: to use an async block, remove the `||`: `async {`
```

Async closures are unstable but the helper text advises you to use async blocks. Async blocks?

That's right, any old block of code in Rust can be async all on its own. They implement `Future` and can be returned from functions just like any other value:

```
#[tokio::main]
async fn main() {
    let msg = "Hello world".to_owned();

 let async_block = || async {
     println!("{}", msg);
 };
 async_block().await; }
```

You can get all the parts of an async closure you need by making a closure that returns an async block!

```
#[tokio::main]
async fn main() {
  let msg = "Hello world".to_owned();

  let closure = || async {
      println!("{}", msg);
  };
  closure().await;
}
```

19.1.5. Send + Sync

Using threads and futures combined with traits will rapidly get you into a situation where you start seeing Rust complain about `Send` and `Sync`, frequently combined with the error `future cannot be sent between threads safely`.

The code below won't compile. It demonstrates a scenario that produces the error.

```
use std::fmt::Display;
use tokio::task::JoinHandle;

#[tokio::main]
async fn main() {
    let mark_twain = "Samuel Clemens".to_owned();

    async_print(mark_twain).await;
}

fn async_print<T: Display>(msg: T) -> JoinHandle<()> {
    tokio::task::spawn(async move {
        println!("{}", msg);
    })
}
```

```
error: future cannot be sent between threads safely
   --> src/send-sync.rs:12:5
    |
12  |     tokio::task::spawn(async move {
    |     ^^^^^^^^^^^^^^^^^^ future created by async block is not `Send`
    |
note: captured value is not `Send`
   --> src/send-sync.rs:13:24
    |
13  |         println!("{}", msg);
    |                        ^^^ has type `T` which is not `Send`
note: required by a bound in `tokio::spawn`
   --> /.../tokio-1.15.0/src/task/spawn.rs:127:21
    |
127 |         T: Future + Send + 'static,
    |                    ^^^^ required by this bound in `tokio::spawn`
help: consider further restricting this bound
    |
11  | fn async_print<T: Display + std::marker::Send>(msg: T) -> JoinHandle<()> {
    |                           ++++++++++++++++++
```

Send and Sync are core to how Rust can promise "fearless concurrency". They are automatic traits. That is, Rust automatically adds Send or Sync to a type if all of its constituent types are also Send or Sync. These traits indicate whether a type can be sent safely across threads or safely accessed by multiple threads. Without these constructs, you could fall into situations where separate threads clobber each other's data or any number of other problems stemming from multi-threaded programming.

Lucky for you though, many Rust types are already Sync *and* Send. You just need to know how to get rid of the error. It's as simple as adding `+ Send`, `+ Sync`, or `+ Sync + Send` to your trait:

```
fn async_print<T: Display + Send>(msg: T) -> JoinHandle<()> {
    tokio::task::spawn(async move {
        println!("{}", msg);
    })
}
```

But now we're presented with another error...

```
error[E0310]: the parameter type `T` may not live long enough
   --> src/send-sync.rs:12:5
    |
11  | fn async_print<T: Display + Send>(msg: T) -> JoinHandle<()> {
    |                -- help: consider adding an explicit lifetime bound...: `T:
'static +`
12  |     tokio::task::spawn(async move {
    |     ^^^^^^^^^^^^^^^^^^ ...so that the type `impl Future` will meet its required
lifetime bounds...
    |
```

We went over `'static` in Chapter 16: Lifetimes, references, and `'static` which is perfect. We now know that we shouldn't fear it. Rust knows that it doesn't know when our async code will run. It's telling us that our type (`parameter type `T``) *may* not live long enough. That wording is important. We just need to let Rust know that the type *can* last forever. Using `'static` here is not saying that it *will* last forever.

```
fn async_print<T: Display + Send + 'static>(msg: T) -> JoinHandle<()> {
    tokio::task::spawn(async move {
        println!("{}", msg);
    })
}
```

There's a lot more to Send & Sync but we'll deal with that another time.

19.1.6. Additional reading

- Rust docs: Future
- Rust docs: `async`
- Asynchronous Programming in Rust book
- Rustonomicon: Send & Sync

19.2. Wrap-up

Async Rust is *beautiful*. There's enough to write a whole book on it alone (and people have). Rust's memory guarantees mean you can write multi-threaded, asynchronous code and be confident it won't explode in your face. *This* is where you start to turn up the heat and leave JavaScript behind. Yes, you *can* use threads in node.js and there are web workers but it's a weird middle ground. It's a compromise. With Rust, we don't have to compromise.

20. Tests and Project Structure

20.1. Introduction

Over the last 18 days we got our environment set up with rustup, VS Code, and rust-analyzer. We pushed through the tough parts of being a newbie Rust developer and just started learning which crates we should start depending on.

Now it's time to set up a real project.

20.2. Creating your workspace

You don't need to use Cargo's workspaces, but I recommend it. Rust — like node.js — is much easier to manage when you cut your application logic into small modules. Workspaces makes that tolerable. We're going to start an executable project which is best set up as a library first with the CLI as a wrapper over the library. This structure is easier to test and easier for you and your users to extend.

First, create a new workspace by starting in an empty directory and making a `Cargo.toml` with the following contents

```
[workspace]
members = ["crates/*"]
```

The `members` entry lists all the crates in your workspace. We configured our workspace to include everything in the `crates` subdirectory (which doesn't exist yet).

20.3. Starting a library

Create a new library crate with `cargo new`:

```
$ cargo new --lib crates/my-lib
```

The difference between a binary crate and a library is minimal. By default, binary crates have a `main.rs`. Libraries use `lib.rs`. The `cargo new` template for libraries also adds `Cargo.lock` to the `.gitignore`.

NOTE The Cargo book advises that you check in your `Cargo.lock` for end-products (binaries, servers, microservices, etc) and omit it for libraries.

The default `lib.rs` template is pretty basic but gives us a new topic to talk about:

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

Unit tests! That's right, Rust has unit testing *built in*. No more configuring the test framework du jour when you start a new project. No more figuring out how to run tests in new projects. It's all the same.

NOTE	Yes, this means that many tests live in your source files. No, there's not really any other way. Rust does have integration tests which can live in a separate `tests` folder alongside `src`, but those only have access to your public APIs. If you want to test small chunks of private code, you have to do it like this.
NOTE	Really? Yes, really. There are crates that extend Rust's testing functionality, but most of them hinge around this same harness and structure.

20.3.1. Unit tests in Rust

The library template introduces two new attributes, `[cfg()]` and `[test]`.

`[cfg()]` is for conditional compilation. By specifying `[cfg(test)]` before an entire module like below, we tell Rust to skip compiling the module unless the `test` flag is on.

```
#[cfg(test)]
mod tests {
}
```

The `[#test]` attribute marks the annotated function as a unit test. Rust's test harness runs each of these separately and reports the results when you run `cargo test`, e.g.

```
$ cargo test
running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s

   Doc-tests my-lib

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s
```

Rust's included assertions are pretty basic. You can assert something is `true` with `assert!()`, equality with `assert_eq!()`, or inequality with `assert_ne!()`.

20.3.2. Writing unit tests

Writing your tests first is a good way to figure out what your API should look like. "Test first" is the core philosophy behind Test Driven Development. Strict TDD is a bit extreme, but writing tests that flex major API points before writing the API methods will force your brain to think about usage before implementation.

What *should* our API look like? Well I hear WebAssembly is pretty hot so let's build a wasm runner. Let's change our `lib.rs` to look like this:

```
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn loads_wasm_file() {
        let result = Module::from_path("./tests/test.wasm");
        assert!(result.is_ok());
    }
}
```

Adding `use super::*` to the `tests` module on line 3 makes it easier to use everything in the parent module without prefixes.

The `Module` struct doesn't exist yet but it seems like a reasonable name for the construct that will wrap a loaded WebAssembly module. I don't know all the methods it will need, but I bet we'll want a function that loads a module from a local file path. Finally, loading from a file path might fail so the return value should be a `Result`. I don't know exactly what'll be in the `Result` but I know I'll want it to be `Ok` if I'm pointing to a valid wasm file.

NOTE	Test-driven development may sound strange if you're not used to it. Strict TDD means going back and forth between tests and code repeatedly. Write a small test, then write the code that makes it pass. I find strict TDD cumbersome and excessive, but the time I spent committed to it taught me a lot about writing testable code.

You can probably recognize what running `cargo test` will do. It will give us a compilation error because we reference structures and functions that don't yet exist.

```
error[E0433]: failed to resolve: use of undeclared type `Module`
 --> crates/wasm-runner/src/lib.rs:6:22
  |
6 |         let result = Module::from_file("./tests/test.wasm");
  |                      ^^^^^^ use of undeclared type `Module`
For more information about this error, try `rustc --explain E0433`.
```

We need to add our `Module` struct, then our `from_file` function. We passed the function a `&str` in our test, but we probably want to be anything that can be represented as a `Path`. This sounds familiar to when we wanted to flexibly represent Strings in Chapter 12: Strings, Part 2 and — guess what? — we can do the same thing with `Path`s:

```
use std::path::Path;
struct Module {}

impl Module {
    fn from_file<T: AsRef<Path>>(path: T) -> Result<Self, ???> {
      Ok(Self{})
    }
}
```

But now we need to figure out what kind of error we're going to return. Since we're loading from a file system and those methods return an `io::Error` we can do that for now. If you don't need to wrap an error, don't. Let your user deal with it.

Now we have code that runs! It doesn't do anything useful but we're getting there. This is our `lib.rs` now:

```
use std::path::Path;
struct Module {}

impl Module {
  fn from_file<T: AsRef<Path>>(path: T) \-> Result<Self, std::io::Error> {
    Ok(Self {})
  }
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn executes_wasm_file() {
        let result = Module::from_file("./tests/test.wasm");
        assert!(result.is_ok());
    }
}
```

20.4. Creating a CLI that uses your library

Run `cargo new crates/[your cli name]` in your workspace. Naming is hard. It's best to leave important names 'til the very end. This is a good place to put a codename if you're creative, or use `cli` if you're not.

```
$ cargo new crates/cli
```

Add the library we just created as a dependency in our `Cargo.toml`.

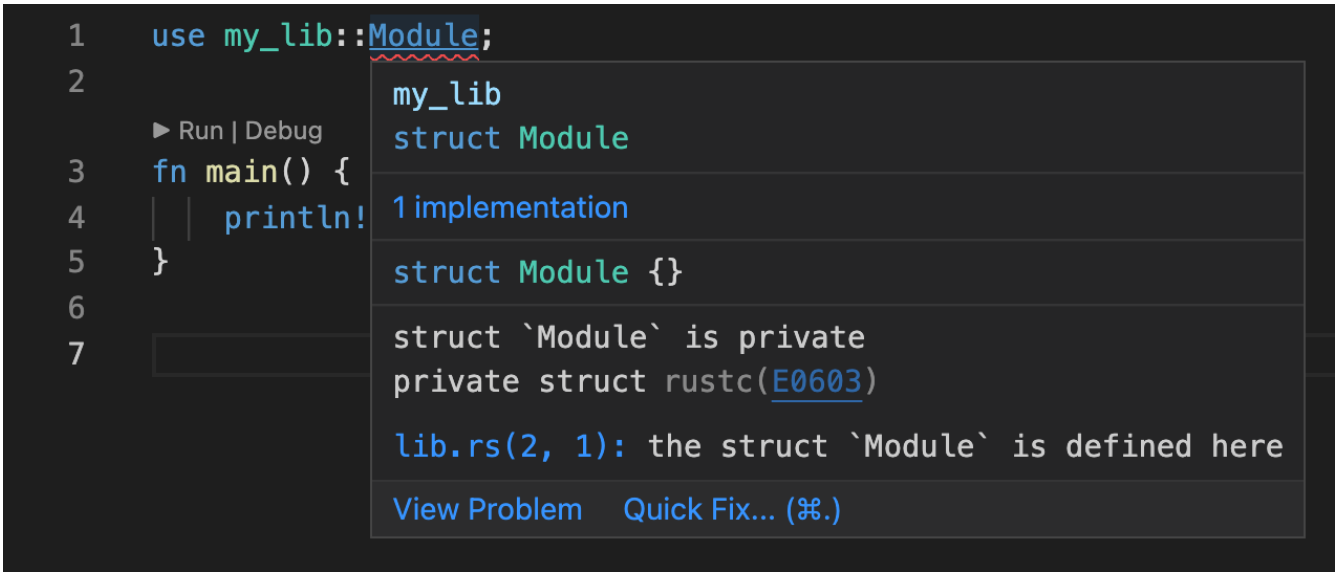
```
[dependencies]
my-lib = { path = "../my-lib" }
```

Now we can use our library by importing from the `my_lib` namespace.

IMPORTANT	Rust has the unfortunate policy of allowing hyphens in crate names but disallowing them as Rust identifiers. If you have a crate with a hyphen, Rust requires that you reference it with the hyphens replaced with underscores.

```
use my_lib::Module;
```

When you add this you'll already see VS Code complaining.



Our `Module` was not explicitly made public so we can't import it. This is one of the many reasons why it's a good idea to set up your projects this way. You get a first-hand view of what it's like to actually use your library. Add `pub` to `struct Module` and `fn from_file` in the `impl` as well. We know we'll need it right away.

```
pub struct Module {}

impl Module {
    pub fn from_file<T: AsRef<Path>>(path: T) \-> Result<Self, std::io::Error> { Ok
(Self {}) } }
```

Now we can import `Module` and use `Module::from_file` in our CLI.

```
use my_lib::Module;

fn main() {
  match Module::from_file("./module.wasm") {
    Ok(_) => {
      println!("Module loaded");
    }
    Err(e) => {
      println!("Module failed to load: {}", e);
    }
  }
}
```

We'll get to the implementations soon, but we're putting together a solid structure for any Rust project right now.

20.5. Running your CLI from your workspace

You can run your CLI from the `./crates/cli` directory with `cargo run`, but cargo can also run

commands in any sub-crate with the `-p` flag. In your project's root, run `cargo run -p cli` to run the default binary in the `cli` crate.

```
$ cargo run -p cli
Module loaded
```

Perfect! We have much more to do, but we have a foundation to build off of now.

20.6. Additional reading

- Rust by Example: Unit testing
- Rust Book, 11.01: How to Write Tests
- Rust Book, 14.03: Cargo Workspaces
- How to Structure Unit Tests in Rust

20.7. Wrap-up

Setting up a solid foundation is important. You'll frequently look at Rust and think "Really? This is the way I'm supposed to do this?" It can shake your confidence and that's what we're here for. When you come across those moments, I'd love to hear them! We've all gone through it, but it's hard to remember how alien everything felt at first now that Rust is a part of our daily lives.

21. CLI Arguments and Logging

21.1. Introduction

Today marks the second day of our real Rust project, a command line utility that runs WebAssembly. So far we've hard-coded our file path and we are using `println!()` as basic logging. It's pretty rigid. We need to add flexibility to our foundation before we add more logic.

Rust has you covered with great solutions for CLI args and basic logging. `structopt` makes CLI arguments even easier to manage than any package I've used from npm. `log` + `env_logger` will give you flexible logging across executables *and* libraries.

21.2. Adding debug logs

Many node.js libraries depend on the the `debug` npm package for per-library debug logging controlled by an environment variable. Rust has a similar, richer solution. Many Rust crates use the `log` crate for logging and the `env_logger` crate for quick STDOUT output. Having the logging and the output decoupled means that you can freely log from your libraries without caring about the output. The output will only be handled by the end user or product.

This is *huge*. It gives library developers the confidence to log whatever they want without worrying if it meshes with other output.

21.2.1. The `log` crate

Let's add the log crate as a dependency to both of our workspace crates:

```
[dependencies]
log = "0.4"
```

Our CLI project should already have our library already listed as dependency and now looks like this:

```
[dependencies]
my-lib = { path = "../my-lib" }
log = "0.4"
```

The log crate gives us the `trace!()`,`debug!()`,`warn!()`,`info!()`, and `error!()` macros. You can use each exactly like you use `println!()`, i.e. you use a string with formatting syntax as the first argument and values that implement `Display` or `Debug` et al. Each of the macros logs your message with the relevant log level. This gives your choice of logger better control over what to log where or what to output.

Let's add some log messages to see how this works. In the `lib.rs` for our `my-lib` crate, add a `debug!()` line right at the start of `from_file()`.

```
pub fn from_file<T: AsRef<Path>>(path: T) \-> Result<Self, std::io::Error> { debug!
("Loading wasm file from {:?}", path.as_ref()); Ok(Self {}) }
```

Now in `main.rs` of our `cli`, change our `println!()` methods to something more appropriate.

```
match Module::from_file("./module.wasm") {
  Ok(_) => {
      info!("Module loaded");
  }
  Err(e) => {
      error!("Module failed to load: {}", e);
  }
}
```

Now when we run our cli we see… nothing at all.

```
$ cargo run -p cli
$
```

Which is exactly right! We can freely add log messages anywhere and not worry about clobbering output!

21.2.2. Printing our logs with `env_logger`

The `env_logger` crate is a simple way to turn those log commands into useful output.

Add `env_logger` to you `cli` project *only*. You don't want the library printing anything. Log output is strictly for the end product.

```
[dependencies]
my-lib = { path = "../my-lib" }
log = "0.4"
env_logger = "0.9"
```

Now we need to initialize our logger as the first thing we do in our `main()`. I added a `debug!()` log following the initialization to make sure we see something:

```
fn main() {
  env_logger::init();
  debug!("Initialized logger");

// ...
}
```

Now when we run our cli we see… nothing at all.

```
$ cargo run -p cli
$
```

Which is exactly right, again! `env_logger` doesn't output anything by default. It needs to be enabled. You can do this programmatically or via an environment variable named `RUST_LOG` by default.

```
$ RUST_LOG=debug cargo run -p cli
[2021-12-21T02:33:24Z DEBUG cli] Initialized logger
[2021-12-21T02:33:24Z DEBUG my_lib] Loading wasm file from "./module.wasm"
[2021-12-21T02:33:24Z INFO  cli] Module loaded
```

Notice how we see logs from both our CLI and our library crates. You can control the level per-module or globally. If we specify `RUST_LOG=info` we'll only see the info messages.

```
$ RUST_LOG=info cargo run -p cli
[2021-12-21T02:33:24Z INFO  cli] Module loaded
```

We can use `[package]=[level]` syntax to filter the level per-module, e.g.

```
» RUST_LOG=cli=debug cargo run -p cli
[2021-12-21T02:35:30Z DEBUG cli] Initialized logger
[2021-12-21T02:35:30Z INFO  cli] Module loaded
```

NOTE Check out env_logger for more documentation on how to control output via the `RUST_LOG` variable.

21.3. Adding CLI Arguments

Now that we can see what's going on in our app, we need it to start pulling in configuration from the command line. `clap` is an amazing library for configuring CLI arguments and `structopt` makes using clap trivial. `structopt` has *many* options. All of it feels like magic.

Add `structopt` to your CLI's dependencies:

```
[dependencies]
my-lib = { path = "../my-lib" }
log = "0.4"
env_logger = "0.9"
structopt = "0.3"
```

Using `structopt` revolves around creating a `struct` that derives the `StructOpt` trait:

```
use structopt::StructOpt;

#[derive(StructOpt)]
struct CliOptions {
}
```

Configuring `StructOpt` happens at two levels, globally and per-argument. Global configuration happens with the `structopt` attribute on the struct itself. The code below gives our application a name, a description, and uses clap's `AppSettings` to give our tool's help fancy colors.

```
use structopt::{clap::AppSettings, StructOpt};

#[derive(StructOpt)]
#[structopt(
    name = "wasm-runner",
    about = "Sample project",
    global_settings(&[
      AppSettings::ColoredHelp
    ]),
)]
struct CliOptions {}
```

Give it a try!

```
cargo run -p cli -- --help
wasm-runner 0.1.0
Sample project

USAGE:
    cli

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information
```

Adding CLI arguments is as easy as adding fields to our struct. Any rustdoc comments (comments starting with three slashes (`///`)) turn into descriptions in your help. The `#[structopt]` attribute takes arguments that control the default value, how its parsed, its environment variable fallback, the short and long form, and much more. If you don't specify a `short` or `long` configuration, then your field is considered a required positional argument.

This code adds one required argument named `file_path`. I could have used a `String` type and used it as a file path, but `structopt` can also preprocess argument values into a more appropriate type by using `parse()` like below:

```
struct CliOptions {
    /// The WebAssembly file to load.
    #[structopt(parse(from_os_str))]
    pub(crate) file_path: PathBuf,
}
```

Generating this structure from actual command line options is a one-line chore. The `StructOpt` traits adds a `from_args` function to your struct and you get a fully-hydrated struct as simple as this:

```
let options = CliOptions::from_args();
```

After adding the above line (line 5) and changing our hard-coded path to use the new `file_path` field in our `CliOptions` struct (line 7), our full `main()` now looks like the code below.

```
fn main() {
  env_logger::init();
  debug!("Initialized logger");

  let options = CliOptions::from_args();

  match Module::from_file(&options.file_path) {
    Ok(_) => {
        info!("Module loaded");
    }
    Err(e) => {
        error!("Module failed to load: {}", e);
    }
  }
}
```

Our CLI behavior and help output update with no effort:

```
$ cargo run -p cli -- --help
wasm-runner 0.1.0
Sample project

USAGE:
    cli <file-path>

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

ARGS:
    <file-path>    The WebAssembly file to load
```

I've been using the `cargo run` syntax which requires that we pass our binary's flags after `--`. If you run the binary directly then you pass them without the separator, e.g.

```
./target/debug/cli --help
```

21.4. Putting it all together.

Because of our debug logging, we can see our command line argument propagate through our simple app by setting `RUST_LOG`, e.g.

```
RUST_LOG=debug ./target/debug/cli ./test_file.wasm
[2021-12-21T03:08:09Z DEBUG cli] Initialized logger
[2021-12-21T03:08:09Z DEBUG my_lib] Loading wasm file from "./test_file.wasm"
[2021-12-21T03:08:09Z INFO  cli] Module loaded
```

NOTE	`./test_file.wasm` doesn't exist, it's just an arbitrary path. Try omitting it and see what happens.

Now we're up and running! Next up we need to figure out this whole WebAssembly thing...

21.5. Additional reading

- `env_logger`
- `log`
- `structopt`
- `clap`

21.6. Wrap-up

This is a solid foundation for many small to medium sized Rust projects. Simple debug logging will last you a while. Eventually you may want to log output to rotated files or pipe them to log aggregators and you can scale up to that.

The next topic we tackle will be WebAssembly. More specifically, how to run it and build with it.

22. Building and Running WebAssembly

22.1. Introduction

WebAssembly is the most exciting technology I've come across since, well, node.js. Server-side JavaScript had been around for ages, but node.js made it *attractive*. It was a real development platform, not just a scripting host. A lot of people — myself included — thought JavaScript could be *the* universal compilation target. Write once, run anywhere. But for real, this time. We even had terms like "Univeral JavaScript" and "Isomorphic JavaScript." We had UMD, the universal module definition to share modules across any platform.

We got close, but we didn't get all the way. Part of the problem is that a lot of important applications depend on serious CPU work. JavaScript just isn't cut out for that no matter how hard we try. We can't always rely on extending the web platform for every use case.

WebAssembly was built for the web but has since gained traction as a universal bytecode format to run code everywhere from the cloud to the blockchain to the internet of things. WebAssembly offers no standard library and can only compute. WebAssembly modules can't even access the filesystem on their own. Many see these as shortcomings, but more are starting to consider them features. Since you *have* to compile foreign standard libraries in your WebAssembly modules, it means every module reliably runs on its own. They're like docker containers for code. Add WASI, the WebAssembly System Interface and the you get granular permissions for expanded capabilities like filesystem or network access.

Oh yeah, you can use WebAssembly in web browsers. That's cool, too.

NOTE	This project builds off the previous two days. It's not critical that you have the foundation to make use of the code here, but it helps.

22.2. Building a WebAssembly module

Compiling down into WebAssembly is easy. What's hard is making it do anything useful. WebAssembly can *still* only talk in numbers (mostly). You can only pass numbers into WebAssembly and WebAssembly can only return numbers. That makes WebAssembly sound like its only good at math, but you could say the same thing about computers. We're skilled at making automatic math machines (a.k.a. "computers") good at *everything*.

WebAssembly will eventually get to a point where it's easier to pass arbitrary data types. Until we get those standards, we have to define our own.

22.2.1. Building applications with waPC

The waPC project, or WebAssembly Procedure Calls, defines a protocol for communicating in and out of WebAssembly. It's like a plugin framework with WebAssembly as the plugin format. In waPC terms, the implementer is a "host" and a WebAssembly module is a "guest."

You can build waPC guests in Rust, TinyGo, and AssemblyScript and you can build waPC hosts in

Rust, JavaScript/node.js, Go, Zig, and Swift.

22.2.2. waPC redux

We've written three guides on how to get started on waPC. The second helps you build WebAssembly modules in Rust and the third shows you how to run them in node.js.

- Building WebAssembly platforms with waPC
- Getting Started with waPC & WebAssembly
- Building a waPC Host in Node.js

This guide will go through how to make a waPC host in Rust to run the same modules.

22.2.3. Our test module

We've pre-built a test module as part of this book. It has one exposed operation, `hello` that takes a string and returns a string.

22.2.4. Building a waPC host

We started off building our project with tests in `my_lib`. Our one test runs a stub function `Module::from_file`. Now we need to finish things up.

Reading files in Rust

You can perform basic reads with `std::fs::read` and `std::fs::read_to_string`. Since we're loading binary data we'll need to use `fs::read` to get a list of bytes, a `Vec<u8>`.

```
pub fn from_file<T: AsRef<Path>>(path: T) \-> Result<Self, std::io::Error> { debug!
("Loading wasm file from {:?}", path.as_ref()); let bytes = fs::read(path.as_ref())?;
// \... }
```

Now that we have bytes, what are we going to do with them? It makes sense that our `Module` might have a constructor function, a `new()`, that takes bytes. Let's finish this function up like so:

```
pub fn from_file<T: AsRef<Path>>(path: T) -> Result<Self, std::io::Error> {
    debug!("Loading wasm file from {:?}", path.as_ref());
    let bytes = fs::read(path.as_ref())?;
    Self::new(&bytes)
}
pub fn new(bytes: &[u8]) -> Result<Self, ???> {
  // ...
}
```

But now we have a problem. It's the same type of problem we had in Chapter 14: Managing Errors. I know that `new()` will need to return a `Result` because loading any old list of bytes as WebAssembly may fail a thousand ways. None of those ways will be an `io::Error` like the one `from_file` returns.

We need a general error that can capture multiple kinds.

Time for a custom error.

Add `thiserror` as a dependency in `my-lib`'s `Cargo.toml`:

```
[dependencies]
log = "0.4"
thiserror = "1.0"
```

Make a new file named `error.rs` and start our `Error` enum. We only know that we'll need an error to manage a load failure but that's a fine start.

```
use std::path::PathBuf;

#[derive(thiserror::Error, Debug)]
pub enum Error {
    #[error("Could not read file {0}: {1}")]
    FileNotReadable(PathBuf, String),
}
```

Rather than wrap the `io::Error` and be done, we added a custom message and multiple arguments so we can customize the error message.

To use our `Error`, first declare the `error` module in `lib.rs` and import our `Error` with:

```
pub mod error;
use error::Error;
```

Now we can change the `Result` type to return our error and use `.map_err()` on the `read()` call to map the `io::Error` to our `Error`:

```
pub fn from_file<T: AsRef<Path>>(path: T) \-> Result<Self, Error> { debug!("Loading
wasm file from {:?}", path.as_ref()); let bytes = fs::read(path.as_ref()) .map_err(|e|
Error::FileNotReadable(path.as_ref().to_path_buf(), e.to_string()))?; Self::new(&
bytes) }
```

`.map_err()` is a common way of converting error types, especially when combined with the question mark (`?`) operator. We didn't add an implementation of `From<io::Error>` because we wanted to have more control over our error message. We have to manually map it ourselves before using `?`.

22.2.5. Using the `wapc` and `wasmtime-provider` crates

The `wapc` crate is home to the `WapcHost` struct and the `WebAssemblyEngineProvider` trait. `WebAssemblyEngineProvider`s allow us to swap multiple WebAssembly engines in and out easily. We're going to use with the wasmtime engine but you can just as easily use wasm3 or implement a

`new` `WebAssemblyEngineProvider` for any new engine on the scene.

Add `wapc` and `wasmtime-provider` to your `Cargo.toml`.

```
[dependencies]
log = "0.4"
thiserror = "1.0"
wapc = "0.10.1"
wasmtime-provider = "0.0.7"
```

Before we make a `WapcHost`, we need to initialize the engine.

```
let engine = wasmtime_provider::WasmtimeEngineProvider::new(bytes, None);
```

The second parameter is our WASI configuration, which we're omitting for now.

The `WapcHost` constructor takes two parameters. One is a `Box`ed engine. The second is the function that runs when a WebAssembly guest calls back into our host. We don't have any interesting implementations for host calls yet, so we're just going to log it and return an error for now.

```
let host = WapcHost::new(Box::new(engine), |_id, binding, ns, operation, payload| {
    trace!(
        "Guest called: binding={}, namespace={}, operation={}, payload={:?}",
        binding,
        ns,
        operation,
        payload
    );
    Err("Not implemented".into())
})
```

The constructor returns a `Result` with a new error so we need to add another error kind to our `Error` enum.

```
#[derive(thiserror::Error, Debug)]
pub enum Error {
    #[error(transparent)]
    WapcError(#[from] wapc::errors::Error),
    #[error("Could not read file \{0}: \{1}")]
    FileNotReadable(PathBuf, String),
}
```

We also need to store the host as part of our `Module` so we can run it later.

```
pub struct Module {
    host: WapcHost,
}
```

The final code looks like this:

```
impl Module {
  pub fn from_file<T: AsRef<Path>>(path: T) -> Result<Self, Error> {
      debug!("Loading wasm file from {:?}", path.as_ref());
      let bytes = fs::read(path.as_ref())
          .map_err(|e| Error::FileNotReadable(path.as_ref().to_path_buf(), e
.to_string()))?;
      Self::new(&bytes)
  }
  pub fn new(bytes: &[u8]) -> Result<Self, Error> {
    let engine = wasmtime_provider::WasmtimeEngineProvider::new(bytes, None);

    let host = WapcHost::new(Box::new(engine), |_id, binding, ns, operation, payload|
{
        trace!(
            "Guest called: binding={}, namespace={}, operation={}, payload={:?}",
            binding,
            ns,
            operation,
            payload
        );
        Err("Not implemented".into())
    })?;
    Ok(Module { host })
  }
}
```

We now have a `from_file` that reads a file and instantiates a new `Module`. If we run `cargo test`, it should pass.

```
$ cargo test
[snipped]
     Running unittests (target/debug/deps/my_lib-afb9e0792e0763e4)

running 1 test
test tests::loads_wasm_file ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.84s
[snipped]
```

Now we need to actually run our wasm!

22.2.6. Calling an operation in WebAssembly

Transferring data in and out of WebAssembly means serializing and deserializing. While waPC does not require any particular serialization format, it's default code generators use MessagePack. That's what we're going to use too but you're free to change it up later.

Add `rmp-serde` to a new `dev-dependencies` section in our `Cargo.toml` so we can use it in our tests.

```
[dev-dependencies]
rmp-serde = "0.15"
```

Writing our test

Like before, we're going to write a test before our implementation. Our test module contains one exposed operation named `hello` that returns a string like `"Hello, World."` when passed a name like `"World"`.

The test will run that operation and assert the output is what we expect.

```
#[cfg(test)]
mod tests {
  // ...snipped
  #[test]
  fn runs_operation() -> Result<(), Error> {
    let module = Module::from_file("./tests/test.wasm")?;

    let bytes = rmp_serde::to_vec("World").unwrap();
    let payload = module.run("hello", &bytes)?;
    let unpacked: String = rmp_serde::decode::from_read_ref(&payload).unwrap();
    assert_eq!(unpacked, "Hello, World.");
    Ok(())
  }
}
```

Line-by-line, the test above:

- loads our test module on line 6
- encodes `"World"` into MessagePack bytes on line 8
- calls an as-of-yet-unimplemented function `.run()` on line 9 with an operation named `hello` and the MessagePacked bytes as the payload.
- decodes the resulting payload as a `String` on line 10
- asserts the string is equal to `"Hello, World."`

When you run the test it won't get passed compilation. We don't have a `.run()` method and we need to implement it first.

Our `.run()` function needs to use the `WapcHost` we created to call into WebAssembly and return the

result.

You call a guest function from a `WapcHost` by using the `.call()` function with the operation (function) name and the payload as parameters.

```
pub fn run(&self, operation: &str, payload: &[u8]) -> Result<Vec<u8>, Error> {
    debug!("Invoking {}", operation);
    let result = self.host.call(operation, payload)?;
    Ok(result)
}
```

If we run our tests again, we'll see our new test passes too!

```
$ cargo test
[snipped]
     Running unittests (target/debug/deps/my_lib-afb9e0792e0763e4)

running 2 tests
test tests::runs_operation ... ok
test tests::loads_wasm_file ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.84s
[snipped]
```

22.2.7. Improving our CLI

Now that our library loads and runs WebAssembly, we need to expose that with our command line utility.

We have two new arguments that need a place in our `CliOptions`, the operation name and the data to pass.

```
struct CliOptions {
  /// The WebAssembly file to load.
  #[structopt(parse(from_os_str))]
  pub(crate) file_path: PathBuf,

  /// The operation to invoke in the WASM file.
  #[structopt()]
  pub(crate) operation: String,

  /// The data to pass to the operation
  #[structopt()]
  pub(crate) data: String,
}
```

We started by putting everything into our `main()` function but that leaves us little room to manage errors nicely. If we bail from `main()` we get a pretty ugly error message and it looks unprofessional.

The setup below extracts business logic to a `run()` function that produces a `Result` that we can test in `main()`. If we get an error, we print it and then exit the process with a non-zero error code to represent failure.

```
fn main() {
    env_logger::init();
    debug!("Initialized logger");

....
let options = CliOptions::from_args();

match run(options) {
    Ok(output) => {
        println!("{}", output);
        info!("Done");
    }
    Err(e) => {
        error!("Module failed to load: {}", e);
        std::process::exit(1);
    }
}; }
....

fn run(options: CliOptions) \-> anyhow::Result<String>{ //\... }
```

Notice how we're also using `anyhow` here. If you remember from [Chapter 14: Managing Errors](./chapter-14-managing-errors.adoc), `anyhow` is a great crate for when you are the end user. It generalizes over most errors and gives you the ability to get things done faster.

Along with `anyhow`, we'll also need to add `rmp-serde` to our production dependencies because we'll be serializing our argument data before sending it to WebAssembly.

```
[dependencies]
my-lib = { path = "../my-lib" }
log = "0.4"
env_logger = "0.9"
structopt = "0.3"
rmp-serde = "0.15"
anyhow = "1.0"
```

Our `.run()` looks very similar to our test, except the data comes from our `CliOptions` vs hard coded strings.

```
fn run(options: CliOptions) -> anyhow::Result<String> {
    let module = Module::from_file(&options.file_path)?;
    info!("Module loaded");

    let bytes = rmp_serde::to_vec(&options.data)?;
    let result = module.run(&options.operation, &bytes)?;
    let unpacked: String = rmp_serde::from_read_ref(&result)?;

    Ok(unpacked)
}
```

Now we can use our CLI utility to run our test wasm itself!

```
» cargo run -p cli -- crates/my-lib/tests/test.wasm hello "Potter"
[snipped]
Hello, Potter.
```

This is a great start to a flexible WebAssembly platform! Congrats! Unfortunately it's limited to only passing and returning strings for now.

We can do much better...

22.3. Additional reading

- `wapc.io`
- `wapc crate`
- `wasmtime`
- `anyhow`

22.4. Wrap-up

This chapter was a big one, I hope you were able to follow along! Developers are using WebAssembly in many different ways. waPC is only one of them. Wasm-bindgen is also popular and more tailored to browser usage of WebAssembly. No matter what route you take, you'll inevitably need to forge your own path through hazardous terrain. WebAssembly is very capable, but it hasn't hit mainstream development yet. As such, best practices are hard to come by.

Next up we'll make our CLI more flexible by allowing it to take and receive arbitrary JSON so you can run any sort of (waPC-compliant) WASM module.

23. Handling JSON

23.1. Introduction

JavaScript without JSON is unthinkable. JSON is the famous, loosely structured data format that — at its core — is just a JavaScript object. It's easy to create, serialize to, and deserialize from. It's so simple that JavaScript developers (myself included) frequently don't even bother associating JSON with a formal structure. We test for undefined values or nonexistent keys like it's normal.

Well for Rust and other typed languages, it's not normal. They need structure. You can represent JSON as the types represented in the spec, but that turns JSON into the worst of every world. It lacks meaningful types that play well in typed languages, and it's neither simple nor satisfying to use.

What we need is a way to translate JSON into something like a JavaScript object, except in Rust. We need to translate JSON to a struct.

23.2. Enter `serde`

`serde` (short for **Ser**ialization/**De**serialization) is a magical crate. With a single line of code, you can serialize your data structures to and from dozens of formats. Serde itself provides the `Serialize` and `Deserialize` traits that let you define how a data structure should be serialized. It doesn't actually serialize anything. That's what other crates are for. We've already used one such crate, `rmp-serde`, to encode data into the MessagePack format. We didn't need to know anything about `serde` because we were serializing common structures. If we want to make anything new, then we must implement the traits.

Luckily, `serde` makes this easy with its `derive` feature. You can get away with serializing most things automatically by deriving `Serialize` and/or `Deserialize` like this:

```
use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize)]
struct Author {
    first: String,
    last: String,
}
```

Once you've implemented one or both traits, you get automagic support for any format in the `serde` ecosystem.

The code below derives `serde`'s `Deserialize` and `Serialize` traits and uses `serde_json` & `rmp-serde` to transform a structure to and from JSON and MessagePack.

NOTE	Notice how we explicitly specify the types we deserialize into on lines 20 & 22. This is the only way the deserializer will know what to output.

```
use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Debug)]
struct Author {
    first: String,
    last: String,
}

fn main() {
    let mark_twain = Author {
        first: "Samuel".to_owned(),
        last: "Clemens".to_owned(),
    };


    let serialized_json = serde_json::to_string(&mark_twain).unwrap();
    println!("Serialized as JSON: {}", serialized_json);
    let serialized_mp = rmp_serde::to_vec(&mark_twain).unwrap();
    println!("Serialized as MessagePack: {:?}", serialized_mp);

    let deserialized_json: Author = serde_json::from_str(&serialized_json).unwrap();
    println!("Deserialized from JSON: {:?}", deserialized_json);
    let deserialized_mp: Author = rmp_serde::from_read_ref(&serialized_mp).unwrap();
    println!("Deserialized from MessagePack: {:?}", deserialized_mp);
}
```

```
$ cargo run -p day-22-serde
[snipped]
Serialized as JSON: {"first":"Samuel","last":"Clemens"}
Serialized as MessagePack: [146, 166, 83, 97, 109, 117, 101, 108, 167, 67, 108, 101,
109, 101, 110, 115]
Deserialized from JSON: Author { first: "Samuel", last: "Clemens" }
Deserialized from MessagePack: Author { first: "Samuel", last: "Clemens" }
```

23.3. Extending our CLI

NOTE	This project builds off the previous three days. It's not critical that you have the foundation to make use of the code here, but it helps.

The last chapter's CLI executed waPC WebAssembly modules that had a very strict signature. Today we're going to extend it to accept arbitrary input and output values represented as JSON.

Add `serde_json` to our CLI's `Cargo.toml`. We don't need `serde` here. I'll go over why below.

```
[dependencies]
my-lib = { path = "../my-lib" }
log = "0.4"
env_logger = "0.9"
structopt = "0.3"
rmp-serde = "0.15"
anyhow = "1.0"
serde_json = "1.0"
```

23.4. Representing arbitrary JSON

Using custom structs is fine when we know what we're representing, but sometimes we don't. Sometimes we need to pass along or translate data structures as an intermediary broker. In that case we need more generic representations. `serde_json's internal representation of JSON is captured in the `serde_json::Value` enum. Rather than create a new struct that derives `Serialize` and `Deserialize` and represents JSON's circular type structure, we can use `serde_json::Value`. This keeps the structure of the JSON in a generic, intermediary format that we can pass along or translate to other formats.

Before we do that though, let's change our CLI argument from passed JSON data to a file path where we can find the JSON.

```
struct CliOptions {
    /// The WebAssembly file to load.
    #[structopt(parse(from_os_str))]
    pub(crate) file_path: PathBuf,

    /// The operation to invoke in the WASM file.
    #[structopt()]
    pub(crate) operation: String,

    /// The path to the JSON data to use as input.
    #[structopt(parse(from_os_str))]
    pub(crate) json_path: PathBuf,
}
```

Now that we have a file, we need to read it. We used `fs::read` to read our WASM file in as bytes, we can use `fs::read_to_string` to read a file in as a `String`.

```
fn run(options: CliOptions) -> anyhow::Result<String>{
  // snipped

  let json = fs::read_to_string(options.json_path)?;

  // snipped
}
```

We use `serde_json::from_str` to parse the JSON into a `serde_json::Value`:

```
fn run(options: CliOptions) -> anyhow::Result<String> {
  // snipped

  let json = fs::read_to_string(options.json_path)?;
  let data: serde_json::Value = serde_json::from_str(&json)?;
  debug!("Data: {:?}", data);

  // snipped
}
```

Lastly, we change our return type and the deserialization type to `serde_json::Value` so we can represent the output as JSON in turn.

```
fn run(options: CliOptions) -> anyhow::Result<serde_json::Value> {
    let module = Module::from_file(&options.file_path)?;
    info!("Module loaded");

    let json = fs::read_to_string(options.json_path)?;
    let data: serde_json::Value = serde_json::from_str(&json)?;
    debug!("Data: {:?}", data);

    let bytes = rmp_serde::to_vec(&data)?;

    debug!("Running  {} with payload: {:?}", options.operation, bytes);
    let result = module.run(&options.operation, &bytes)?;
    let unpacked: serde_json::Value = rmp_serde::from_read_ref(&result)?;

    Ok(unpacked)
}
```

And we're done! We can run our test file from the last chapter after putting the input into a JSON file:

```
cargo run -p cli -- crates/my-lib/tests/test.wasm hello hello.json
[snipped]
"Hello, Potter."
```

But now you can run arbitrary, waPC-compliant WebAssembly modules and parse the output as JSON. Today's project includes a module that produces HTML output from a handlebars template and a `Blog`-style type that includes a title, author, and body.

```
$ cargo run -p cli -- ./blog.wasm render ./blog.json
[snipped]
"<html><head><title>The Adventures of Tom Sawyer</title></head><body><h1>The
Adventures of Tom Sawyer</h1><h2>By Mark Twain</h2><p>TOM!\n\nNo
answer.\n\nTOM!\n\nNo answer.\n\nWhats gone with that boy,  I wonder? You
TOM!\n\nNo answer.</p></body></html>"
```

Our CLI is getting useful. It's about time we name it something better than `cli`. The binary takes on the name of the crate unless overridden. Change it to something appropriate like `wapc-runner` in `Cargo.toml`.

```
[package]
name = "wapc-runner"
```

We've also been running our debug builds up to now. Try building the binary in release mode to see what your end product looks like.

WARNING	Building in release mode may take a *lot* longer, depending on the machine you are building on.

```
$ cargo build --release
[snipped]
    Finished release [optimized] target(s) in 6m 08s
$ cp ./target/release/wapc-runner .
$ ./wapc-runner ./blog.wasm render ./blog.json
"<html><head><title>The Adventures of Tom Sawyer</title></head><body><h1>The
Adventures of Tom Sawyer</h1><h2>By Mark Twain</h2><p>TOM!\n\nNo answer.\n\n
TOM!\n\nNo answer.\n\nWhats gone with that boy,  I wonder? You TOM!\n\nNo
answer.</p></body></html>"
```

NOTE	Note, `wasmtime` performance is great with already-loaded modules, but the startup time is noticeable. You can reduce this substantially by using its `cache` feature which caches an intermediary representation for speedier startup.

And now we have a portable WebAssembly executor that runs waPC modules on the command line. That's pretty awesome.

If you're looking for ideas on where to go next:

1. Take JSON data from STDIN when the file argument is missing so you can `cat` JSON to your binary. (Hint, the `atty` crate will help you determine if your process is being piped to or is interactive)
2. Decouple the template from the JSON for the blog module and take an optional template from a file. A `.hbs` file is included in the project repo. (Hint: An optional file path argument should probably be `Option<PathBuf>`)

23.5. Additional reading

- `serde`
- `serde_json`
- `rmp-serde`
- `handlebars`
- `The Adventures of Tom Sawyer`

23.6. Wrap-up

We just built a pretty heavy CLI application in surprisingly little code. Well, I hope you're surprised. Once you get passed some of the early hurdles and find ways to mitigate the verbosity of Rust's quirks, you can deliver a big impact just as easy as if you were writing JavaScript. `StructOpt` and `serde` are only a few of the amazing crates you can find on `crates.io`. There are many others and opportunities for many more. All Rust needs are some motivated new developers who come from a rich ecosystem of small modules. *Hint hint…*

24. Cheating The Borrow Checker

24.1. Introduction

Over the last twenty-two days you've gotten comfortable with the basics of Rust. You know that Rust values can only have one owner, you kind of get lifetimes, and you've come to terms with mutability. It's all a bit strange but you accept it. People are making big things with Rust and you trust the momentum. You're ready.

Then you start a project. It goes well at first. You get into a groove and then — all of a sudden — you're stopped dead in your tracks. You can't figure out how to satisfy Rust. You've already bent over backwards. You're drowning in references and lifetime annotations. Rust is yelling about `Sync`, `Send`, lifetimes, moved values, what have you. You just want more than one owner or multiple mutable borrows. You're about to give up.

That's where `Rc`, `Arc`, `Mutex`, and `RwLock` come in.

NOTE	Yeah, it's not "cheating the borrow checker," but it feels like it.

24.1.1. Reference counting in Rust with `Rc` & `Arc`

`Rc` and `Arc` are Rust's reference counted types. That's right. After all the talk about how garbage collectors manage memory by counting references and how Rust uses lifetimes and doesn't need a garbage collector; you can manage memory by counting references.

NOTE	I don't know about you, but my journey through Rust was an emotional tug-of-war. The heavy focus on Rust's lifetimes and borrow checker made it seem like *I* was wrong when I ran up against it. I felt like I was trying to cheat and kept getting caught. I would refactor and refactor but always end up in the same place.

Turns out I *was* wrong, but not in the way I thought. My interpretation of Rust docs and articles gave me tunnel vision. I had myself convinced that reference counting was bad so I never looked for it. When I saw it, I assumed it was in a negative context and I skimmed passed. This was only reinforced by seeing recurring references to how you should avoid things like `Rc<RefCell>`. I lost a lot of time. I hope you don't.

Consider a situation where you have an value that multiple other structs need to point to. Maybe you're building a game about space pirates that have high-tech treasure maps that always point to a treasure's actual location. Our maps will need a reference to the Treasure at all times. Creating these objects might look like this.

```
let booty = Treasure { dubloons: 1000 };

let my_map = TreasureMap::new(&booty);
let your_map = my_map.clone();
```

Our `Treasure` struct is straightforward:

```
#[derive(Debug)]
struct Treasure {
    dubloons: u32,
}
```

But our `TreasureMap` struct holds a reference and Rust starts yelling about lifetime parameters. A budding Rust developer might accept what they're told to get things compiling. After all: if it compiles, it works. Right?

```
#[derive(Clone, Debug)]
struct TreasureMap<'a> {
    treasure: &'a Treasure,
}

impl<'a> TreasureMap<'a> {
    fn new(treasure: &'a Treasure) -> Self {
        TreasureMap { treasure }
    }
}
```

It works! Our code grew a bit and for questionable value, but it works.

```
fn main() {
    let booty = Treasure { dubloons: 1000 };

    let my_map = TreasureMap::new(&booty);
    let your_map = my_map.clone();
    println!("{:?}", my_map);
    println!("{:?}", your_map);
}

#[derive(Debug)]
struct Treasure {
    dubloons: u32,
}

#[derive(Clone, Debug)]
struct TreasureMap<'a> {
    treasure: &'a Treasure,
}

impl<'a> TreasureMap<'a> {
    fn new(treasure: &'a Treasure) -> Self {
        TreasureMap { treasure }
    }
}
```

```
$ cargo run -p day-23-rc-arc --bin references
[snipped]
TreasureMap { treasure: Treasure { dubloons: 1000 } }
TreasureMap { treasure: Treasure { dubloons: 1000 } }
```

But now everything that uses a `TreasureMap` needs to deal with lifetime parameters...

```
struct SpacePirate<'a> {
    treasure_maps: Vec<TreasureMap<'a>>,
}
impl<'a> SpacePirate<'a> {}

struct SpaceGuild<'a> {
    pirates: Vec<SpacePirate<'a>>,
}
impl<'a> SpaceGuild<'a> {}
```

If you keep going, you're rewarded with more pain and no perceivable gain.

It's time for `Rc`.

Remember how `Box` essentially gives you an owned reference (see: Chapter 14 : What's a box?)? `Rc`

is like a shared Box. You can .clone() them all day long with every version pointing to the same underlying value. Rust's borrow checker cleans up the memory when the last reference's lifetime ends. It's like a mini garbage collector.

You use Rc just as you would Box, e.g.

```
use std::rc::Rc;

fn main() {
    let booty = Rc::new(Treasure { dubloons: 1000 });

 let my_map = TreasureMap::new(booty);
 let your_map = my_map.clone();
 println!("{:?}", my_map);
 println!("{:?}", your_map); }

#[derive(Debug)]
struct Treasure {
    dubloons: u32,
}

#[derive(Clone, Debug)]
struct TreasureMap {
    treasure: Rc<Treasure>, }

impl TreasureMap {
    fn new(treasure: Rc<Treasure>) \-> Self { TreasureMap { treasure } } }
```

Rc does not work across threads. That is, Rc is !Send (See: [Chapter 18: Send + Sync](/blog/node-to-rust-day-18-async/#send-sync)). If we try to run code that sends a TreasureMap to another thread, Rust will yell at us.

```
fn main() {
    let booty = Rc::new(Treasure { dubloons: 1000 });

    let my_map = TreasureMap::new(booty);

    let your_map = my_map.clone();
    let sender = std::thread::spawn(move || {
        println!("Map in thread {:?}", your_map);
    });
    println!("{:?}", my_map);

    sender.join();
}
```

```
[snipped]
error[E0277]: `Rc<Treasure>` cannot be sent between threads safely
   --> crates/day-23/rc-arc/./src/rc.rs:9:18
    |
9   |       let sender = std::thread::spawn(move || {
    |  ______________________^^^^^^^^^^^^^^^^^^_-
    | |                      |
    | |                      `Rc<Treasure>` cannot be sent between threads safely
10  | |         println!("Map in thread {:?}", your_map);
11  | |     });
    | |_____- within this `[closure@crates/day-23/rc-arc/./src/rc.rs:9:37: 11:6]`
[snipped]
```

`Arc` is the `Send` version of `Rc`. It stands for "Atomically Reference Counted" and all you need to know is that it handles what `Rc` can't, with slightly greater overhead.

> **NOTE** When Rust documentation refers to "additional overhead" for a feature you need, just take it. You come from JavaScript, don't fret about "overhead."

`Arc` is a drop-in replacement for `Rc` in read-only situations. If you need to alter the held value, that's a different story. Mutating values across threads requires a lock. Attempting to mutate an `Arc` will give you errors like:

```
error[E0596]: cannot borrow data in an `Arc` as mutable

or

error[E0594]: cannot assign to data in an `Arc`
```

24.2. `Mutex` & `RwLock`

If `Arc` is the answer to you needing `Send`. `Mutex` and `RwLock` are your answers to needing `Sync`.

`Mutex` (Mutual Exclusion) provides a lock on an object that guarantees *only one* access to read *or* write at a time. `RwLock` allows for *many reads* but *at most one write* at a time. `Mutex`es are cheaper than `RwLock`s, but are more restrictive.

With an `Arc<Mutex>` or `Arc<RwLock>`, you can mutate data safely across threads. Before we start going into `Mutex` and `RwLock` usage, it's worth talking about `parking_lot`.

24.2.1. `parking_lot`

The `parking_lot` crate offers several replacements for Rust's own sync types. It promises faster performance and smaller size but the most important feature in my opinion is they doesn't require managing a `Result`. Rust's `Mutex` and `RwLock` return a `Result` which is unwelcome noise if we can avoid it.

24.2.2. Locks and guards

When you lock a `Mutex` or `RwLock` you take ownership of a guard value. You treat the guard like your inner type, and when you drop the guard you drop the lock. You can drop a guard explictly via `drop(guard)` or let it drop naturally when it goes out of scope. When dealing with locks you should get into the practice of dropping them ASAP, lest you run into a deadlock situation where two threads hold locks the other is waiting on.

Rust's blocks make it easy to limit a guard's scope and have them drop automatically when you are done with them. The code sample below uses a block to scope the guard from `treasure.write()` so that it automatically drops at the end of the block (line 8)

```
fn main() {
  let treasure = RwLock::new(Treasure { dubloons: 1000 });

{
      let mut lock = treasure.write();
      lock.dubloons = 0;
      println!("Treasure emptied!");
  }

println!("Treasure: {:?}", treasure);
}
```

24.3. Async

Async Rust and futures add another wrench into the lock and guard problem. It's easy to write code that holds a guard across an async boundary, e.g.

```
#[tokio::main]
async fn main() {
    let treasure = RwLock::new(Treasure { dubloons: 100 });
    tokio::task::spawn(empty_treasure_and_party(&treasure)).await;
}

async fn empty_treasure_and_party(treasure: &RwLock<Treasure>) {
  let mut lock = treasure.write();
  lock.dubloons = 0;

  // Await an async function
  pirate_party().await;

} // lock goes out of scope here

async fn pirate_party() {}
```

The best solution is to not do this. Drop your lock before you await. If you can't avoid this situation,

`tokio` does have its own sync types. Use them as a last resort. It's not about performance (though there is that "overhead" again), it's a matter of adding complexity and cycles for a situation you can (or should) get out of.

24.4. Additional reading

- `std::rc`
- `std::rc::Rc`
- `std::sync::Arc`
- `std::sync::Mutex`
- `std::sync::RwLock`
- `parking_lot`
- `Tokio Sync`

24.5. Wrap-up

`RwLock` and `Mutex` (along with types like `RefCell`) give you the flexibility to mutate inner fields of a immutable struct safely, even across threads. `Arc` and `Rc` were major keys to me understanding how to use real Rust. I overused each of these types when I started using them. I don't regret it one bit. If you take away only one thing from this book, let it be that you need to do what works for you. If you keep trying you can always improve, but you won't get anywhere if you're so frustrated you give up.

25. Crates & Valuable Tools

25.1. Introduction

Hopefully you've become comfortable enough with Rust that you want to continue your journey. Rust is hard to get into but it's worth it. I've found that I can build *massive* projects with a fraction of the unit tests I'm used to writing. Rust handles so many error cases at the compilation stage. Writing Rust is like pair programming with a grumpy old developer who catches *everything*.

Now you need to start building up your suite of core dependencies, the crates you rely on repeatedly and become part of your personal toolbox. These are a few of the crates in mine.

25.2. Crates

25.2.1. Data

- serde - Data serialization ecosystem.
- serde_json - JSON parsing and stringification.
- parking_lot - Better Rust `Mutex` and `RwLock`.
- once_cell - When you think you want global variables.

25.2.2. Error handling

- thiserror - Easy custom errors.
- anyhow - Easy generic errors.

25.2.3. Logging

- log - Logging facade that abstracts logging usage away from the logger implementation.
- env_logger - Easy console logging controlled by an environment variable.
- test-log - Get `env_logger`-style output in your tests without worrying about initializing a logger.
- pretty-env-logger - `env_logger`, but *prettier*.
- tracing - a drop-in replacement for `log` and `env_logger` with expanded capabilities and first-class async support.

25.2.4. CLI

- structopt - CLI arguments from configuration.
- clap - CLI arguments from code.

25.2.5. Async/concurrency

- tokio - Async runtime.

- tokio-stream - Stream utilities for Tokio.
- async-trait - For when you try to make traits with async methods.
- crossbeam - bidirectional communication channels and more.

25.2.6. Web

- rocket - An HTTP server with a great developer experience.
- reqwest - an easy-to-use HTTP client.
- hyper - a fast and correct HTTP implementation.

25.2.7. Functionality you expected in the standard library

- rand - random number generator and related tools.
- regex - Regular expressions.
- base64 - Base64 implementation.
- http - A general purpose library of common HTTP types.

25.2.8. Misc & tools

- uuid - UUID implementation.
- itertools - Additional iterator functions and macros.
- maplit - `hashmap!{}` macro like `vec![]`
- cfg-if - Macros to simplify mutually exclusive conditional compilation `#[cfg]` attributes.
- `just` - a better `make`.

25.3. Additional reading

- Rust Language Cheatsheet
- Awesome Rust list
- Rust Design Patterns
- Rust Quiz

25.4. Wrap-up

During this book I learned that Rust has a curse. Once you learn "The Rust Way" of doing things, you get amnesia. You forget how to program any other way. This curse leads to documentation and examples that naturally skirt around problems 99% of new developers will experience in practice. I found myself unable to replicate several errors that were major roadblocks in my early Rust journey.

The Rust curse is why it's critical that ***you*** are part of the Rust community. ***You*** are the only one that can help. You have a fresh perspective and run into real new-user problems. When you get stuck,

reduce your code down to a minimally reproducible example and save it. When you eventually fix it, write about what you changed! If you don't like writing, can't figure something out, or are unsure about your fix, send it to me. I would be honored to take your experience and write about it.