One of the things I've wanted to learn about, in preparation for my college classes, is lower-level programming. Sure, TypeScript is cool on websites and node.js development makes it easy to get projects started, but building a physical product and controlling more parts of computers would probably be fun.

Rust is a systems design language - by design lower-level than JavaScript/Typescript and as such, it's more useful for embedded programming and other smaller things. This means it has a bunch of caveats, and many parts of the language and libraries are still changing.

Similarly to C, it has a main function that is launched when the program starts. `fn main() {}` is its syntax. I won't be talking too much on the parts that are similar to JavaScript, Java, and other languages.

Benefits

- Low-level
- Sidesteps a lot of pitfalls by clearly defining how to return an error or other bits of data

The Rust Ecosystem

Cargo

This is the home for a lot of Rust's ecosystem. A new project is created with `cargo new`. The formatter runs with `cargo fmt`. Additional code hints are provided by `cargo clippy`.

fmt

Basic formatting.

Clippy

Wanting to sidestep a lot of the issues that come from loosely structured programming languages like C, Clippy offers tips on what exactly is wrong with your code. There's a metric ton of hints, and they can be customized - info on that is in the GitHub.

crates.io

This is the documentation for Rust packages.

The more quirky parts

Mutability & Immutability

Because this is a system programming language, it has to deal with memory in different ways. A program may contain data that doesn't change (which can be especially optimized), or data that does change.

`trait`s, `impl`s, and `struct`s

Rust is not exactly object-oriented. It has `trait`s which define behaviors that are required for different `impl`s (implementations). It also allows you to derive traits from multiple sources, which means that you sometimes you need to import specific traits to get functionality. Thankfully, a lot of this is handled by the compiler so you get alerts early in the process.

Threading support

One of the important parts for distributed systems is support for threading and various utilities. Threads are easy enough to implement, like this:

```
use std::thread;

fn main() {
        let threads = thread::spawn(|| {
                println!("Hello from some other thread!");

        })
}
```

However, things like mutexes, read/write locks, and other portions are required for us to actually do important stuff.

Borrowing & Ownership

Its system of borrowing and ownership is *the* unique aspect of the programming language. Once something's passed to a different scope (function), it is given to that function, and the parent doesn't have it anymore.

```
fn main() {
let field = "Name";
ask_for_field(field);
println!("{}", field); // THIS WILL RETURN AN ERROR!
}
```

The field variable is passed to `ask_for_field` and `main` is not the owner anymore. Hence, if we want `ask_for_field` to only have it for a while, we have to do things like this:

```
fn main() {
let field = "Name";
ask_for_field(field);
println!("{}", field); // THIS WILL RETURN AN ERROR!
}
```

This is also the hardest part to understand, given that sometimes the automatic lifetime checking doesn't exist. Which means that sometimes you have to add `'static` to variables, and it gets even more complicated with UI controls and other parts.
There was a lot of useful documentation on the Rust website, and I also used a book to help explain some concepts.