# Traits

Last week we looked at *sum types*. We introduced the *enum* keyword, and looked at a couple of common enums in the standard library used to handle two cases - optional values (Option) & fallibility (Result). With Result, we started talked about Rust's error handling story. As a reminder, the Result type is defined in terms of two generic parameters, like so:

```
enum Result <Success,Error> {
    Ok(Success),
    Err(Error)
}
```

But we still have to discuss errors in a bit more detail. Because the Result type doesn't constrain the Error type in Result. That is pretty broad. You can put anything in there. In order to provide a consistent error handling experience, Rust's standard library defines a companion trait called Error, which looks like this:

```
pub trait Error: Debug + Display {
    // rust provides a default impl (returns None)
    fn source(&self) -> Option<&(dyn Error + 'static)> {...};

    // only available in nightly. Defaults to None
    fn backtrace(&self) -> Option<&Backtrace> {...};

    // deprecated. use Display impl. Rust provides a default impl so you
dont have to implement it
    fn description(&self) -> &str {...};

    // deprecated. use Display impl. Rust provides a default impl so you
dont have to implement it
    fn cause(&self) -> Option<&dyn Error> {...} ;
}
```

Since we have only touched briefly on traits, an explanation is in order. On the first line, that bit with the colon

```
pub trait Error: Debug + Display
```

means that Error is a supertrait of the Debug and the Display traits. In other words, it means that in order to implement Error, one must also implement Debug and Display. In fact, as an aside, it is perfectly valid for a trait to do nothing more than this.

```
pub trait Social: Debug + Display {}
```

The second thing to note is that unlike pure interfaces, a Rust trait may provide a default implementation for a trait function. This provides a powerful way of inheriting behaviors. So while Rust does not provide classical OO inheritance, it does provide another means of being lazy.

And getting back to Error, all of the trait's defined functions have default implementations. Meaning that as long as you define Debug and Display for a type, you can trivially implement Error.

Which brings us to another point. How do you implement a trait? The syntax is only a bit different than implementing a struct or enum.

```
impl <TRAIT> for <STRUCT|ENUM>
```

So if we have a trait called Info, and a struct called Person, we would implement the trait like so:

```
impl Info for Person {
    //... whatever functions we need to implement as defined by the trait
go here
}
```

One more bit of syntax you will have to know while we are at it. If the struct or enum you are defining a trait for has a generic or lifetime parameter (or parameters), you have to declare them on the impl block.

```
struct StructB<B> {
    marker: B
}

trait Trait<T: MyMarker> {
    fn foo(&self, a: StructA<T>) -> Self;
}

impl<T: MyMarker> Trait<T> for StructB<T> {
    fn foo(&self, _: StructA<T>) -> StructB<T> {
        unimplemented!()
    }
}
```

Getting back to the Error trait, and custom errors, typically, library authors will implement at least one error enum for their crate. And, as we have seen, Error is a supertrait of Display and Debug. So lets take a quick look at them.

But first, a quick note about these two traits. For those of you coming from python (everyone), you can think of Display as being akin to `__str__` and Debug as akin to `__repr__`... kind of. Display's audience is made up of the end users. Debug is intended to aid in debugging your library or program and should provide information in that light.

## Display Trait

```
pub trait Display {
    fn fmt(&self, f: &mut Formatter) -> Result<(), Error>;
}
```

So what is this Formatter type?

A Formatter represents various options related to formatting. Users do not construct Formatters directly; a mutable reference to one is passed to the fmt method of all formatting traits, like Debug and Display.

Here is an implementation:

```
use std::fmt;

struct Point {
    x: i32,
    y: i32,
}

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

Your implementation of Display is going to normally follow the example above, passing the formatter to the `write!` macro.

# Debug Trait

The Debug trait is a bit more complex (just kidding). Debug should format the output in a programmer-facing, debugging context.

Here is how the Debug trait is defined:

```
pub trait Debug {
    fn fmt(&self, f: &mut Formatter) -> Result<(), Error>;
}
```

And here is an example implementation

```
use std::fmt;

struct Point {
    x: i32,
```

```
    y: i32,
}

impl fmt::Debug for Point {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.debug_struct("Point")
         .field("x", &self.x)
         .field("y", &self.y)
         .finish()
    }
}
```

There are a number of methods on the Formatter struct to help you with manual implementations, such as debug_struct.

Debug implementations using either derive or the debug builder API on Formatter support pretty-printing using the alternate flag: {:#?}.

In reality, the above code should almost always be handled by deriving Debug:

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}
```

Debug is implemented for all of your default types for you already.

# Error - Debug + Display

As I mentioned before, to implement Error you really only need to implement Debug and Display. All of the other methods have default implementations.

```
use std::error::Error;
use std::fmt::Display;
use std::fmt::Debug;

#[derive(Debug, PartialEq, Eq)]
pub enum MyError {
    NoInputError,
    IoError { problem: String },
    RuntimeError(String),
}

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

```
            MyError::IoError { problem } => write!(f, "IoError - cause:
{}", problem.as_str()),
            MyError::RuntimeError(inner) => write!(f, "RuntimeError({})",
inner),
        }
    }
}
impl Error for MyError {}
```

## Handling errors

You might be wondering about now just how much of a pain it is going to be handling these errors. Especially given what we know about the match statement and its requirement that you handle all cases. If you have a bunch of fallible function calls, that sounds like a lot of boilerplate.

And you would be right, except...

### Introducing the '?' Operator

Rust has a magic little operator - the question mark. The question mark does a couple of things when operating on a Result.

First, if the result is Ok of something, it unwraps it.

Second, if the result is Err of something then it raises it. (actually it does a bit more but you will have to wait a bit to find out what else).

So, in reality this is what an function might look like:

```
fn get_uri() -> Result<Uri, MyError> {
    let scheme = get_scheme()?;
    let authority = get_authority()?;
    let resource = get_resource()?;
    Uri(scheme, authority, resource)
}
```

Much better. However, you might be wondering what to do about heterogeneous errors. In the above example, the function is calling other functions which all seem to return a MyError instance. However, it is reasonable to assume that in any project, one will include dependencies that themselves define their own errors. How do we deal with that?

The first way is to in fact do what we did. Define an error enum and wrap foreign errors. This is a common approach used by library authors. Of course, this still wont magically convert between a foreign error and your error. So Result defines a function called map_err, that lets you map over the result value if the variant is an Error.

```
    let foo = frobulator().map_err(|e| MyError::FrobulatorError{cause:
format!("{:?}", e)})?;
```

However, there is an even better way to deal with this. Simply define an implementation of the From trait, mapping "from" some foreign error type "to" your error type.

### The From Trait

The from trait is defined simple like so:

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

And one would use it like so:

```
impl From<Io::Error> for MyError {
    fn from(e: Io::Error) -> Self {
        Self{issue: e.to_string() }
    }
}
```

If you do this, then the `?` operator will invoke from on your behalf. (that is the other bit I was talking about).

So, you don't have to use the `map_err` method, as long as you have a `From` impl.

That is certainly much better, but it still sounds like some work. It isn't always necessary either. Some times, you don't really care about the error type, because you are simply going to present it to the user. You aren't going to do anything else fancy. In this case, you may simply return an error trait object.

If you recall from a last week, when talking about generics, I mentioned that there were two ways of dealing with generics. The first an most prevalent is the standard, monomorphised generic. Eg

```
pub enum foo<T> {
    Foo(T)
    Bar
}
```

In the above case, when you fill in the generic variable with a type, the compiler follows suite and creates a distinct type. So `Foo<String> is distinct from Foo<u32>` and when you define a Vec if you want to store a vector of Foos, you must declare what type they are. Eg `Vec<Foo<String>>`. That vector will only hold Foo of Strings.

But there is another case - the trait object. A trait object is dynamic. You can have a container contain different underlying types which all implement a particular trait. When operating on said object, you only have access to the trait and supertrait methods fyi. And they are dispatched virtually at runtime. (like c++ virtual methods).

You declare a trait object in return position using the *dyn* keyword. So, if you want to return a function that can handle any error which implements dyn Error, you do this:

```
fn foo() -> Result<(), Box<dyn Error>> {
    let bla = frobulate()?;
    let blab = corpusculate(&bla)?;
    Ok(())
}
```

As an aside, if you don't recall, with trait objects, we cannot deal with them directly because the compiler needs to know how big they are at compile time. But they are dynamic. By boxing them (putting then on the heap), the compiler knows their size.

If you are really paying attention, you might be wondering how Rust handles the Boxing. Like how do we go from a specific error to a box of a dyn error? The stdlib helps in boxing our errors by having Box implement conversion from any type that implements the Error trait into the trait object Box, via From.

https://doc.rust-lang.org/beta/std/boxed/struct.Box.html