Module 03: Traits

Introduction

If the C programming language had only one flaw, it would be its poorly reusable code. This language makes it very difficult to write code that may be used in multiple similar situations, especially when the only thing that changes is a bunch of types.

Rust fixes this issue by providing a way to encode what types have in common - their traits. In fact, the Rust type system wouldn't be very useful without traits. If you are comming from a more object-oriented language, you may already be familliar with them: those are basically interfaces on steroids.

This module will teach what traits are, how they work, and why they are so important.

General Rules

Any program you turn in should compile using the `cargo` package manager, either with `cargo run` if the subject requires a *program*, or with `cargo test` otherwise. Only dependencies specified in the `allowed dependencies` section are allowed.

Any program you turn in should compile *without warnings* using the `rustc` compiler available on the school's machines without additional options. You are allowed to use attributes to modify lint levels, but you must be able to explain why you did so. You are *not* allowed to use `unsafe` code anywere in your code.

Exercise 00: Every Type Has A Name

```
turn-in directory:
    ex00/

files to turn in:
    src/main.rs  Cargo.toml

allowed dependencies:
```

Creating a trait is *defining* a set of capabilities types may or may not exhibit. For example, one could create a `PrintMyself` trait, which requires any implementator to define a `print_myself` method.

You know what, let's do this. Create a `PrintMyself` trait with a `print_myself` method. That function should be prototyped like this:

```
fn print_myself(&self);
```

Implement this trait for a couple types (you can even create your own types and implement the trait). Once you are done deriving the trait, write a simple `main` function to test those implementations.

Example:

```
fn main() {
    125u32.print_myself();
    12i8.print_myself();
    "Hello!".print_myself();
}
```

```
>_ cargo run
125u32
12i8
"Hello!"
```

But you are free to experiment more with that!

Exercise 01: Infallible Addition

```
turn-in directory:
    ex01/

files to turn in:
    src/lib.rs  Cargo.toml

allowed dependencies:
```

Trait are not just mere interfaces. It is possible for them to define *associated types*.

Create the trait `InfallibleOps` with the following associated methods:

```
fn infallible_add(self, other: Self) -> /* ... */;
fn infallible_mul(self, other: Self) -> /* ... */;
```

The `/* ... */` comment must be replaced with the return-type of the functions. You will have to define an *associated type* for the trait and use it there. That type should be able to represent any possible result of the operation `self + other` (or `self * other`).

For example, the result of `250u8 + 10u8` cannot be represented by the `u8` type, but it fits in the `u16` type!

Implement the `InfallibleOps` trait for common types, such as `i8`, `u8`, or `u32`. Provide tests to show the functions are working as expected.

Exercise 02: Operator Overloading

```
turn-in directory:
    ex02/

files to turn in:
    src/lib.rs  Cargo.toml

allowed dependencies:
```

In Rust, common operators such as `+` or `/` are just fancy function calls. The function that is called when writing `a + b` is the `Add::add` function (the `add` associated function of the `Add` trait).

Create a `Vector2` type, with an `x` and `y` field and make it implement the `Add` trait. The following test should compile properly. You may have to add the `#[derive(Clone, Copy, Debug)]` attribute to your type, though.

```
#[test]
fn basic_vector2_add() {
    let a = Vector2 { x: 12, y: 25 };
    let b = Vector2 { x: 1, y: 2 };
    let c = Vector2 { x: 13, y: 27 };

    assert_eq!(c, a + b);
}
```

Now that you understand how to overload the `+` operator, do the same thing for the `+=`, `-`, `-=`, and `==` operators.

You must write tests for every function you write!

Tip: you may want to check the Standard Library's documentation out.

Exercise 03: Dry Boilerplate

```
turn-in directory:
    ex03/

files to turn in:
    src/main.rs  Cargo.toml

allowed dependencies:
```

Deriving manually some common traits can be a bit borring. For example, in most cases, the `PartialEq` trait will simply check for the equality of every field of a type. To gain time, Rust provides the `#[derive(...)]` attribute. This attribute can be added to any Rust type to derive automatically some common traits.

Create a Rust type. Anything. You simply have to name it `MyType`.

```
fn main() {
    let instance = MyType::default();

    println!("the default value of MyType is {instance:?}");
    assert_eq!(instance, MyType::default(), "the default value isn't always the same :/");
}
```

Copy the above `main` function and use the `#[derive(...)]` attribute on `MyType` to make it compile. You are not allowed to use the `impl` keyword!

Exercise 04: n-th Successor

```
turn-in directory:
    ex04/

files to turn in:
    src/lib.rs  Cargo.toml

allowed dependencies:
```

Sometimes, one of the methods of a trait can be expressed in term of another method. Rust allows you to provide a default implementation for the methods of a trait.

Create a trait named `Successor`. A type deriving this trait should be able to provide a `successor` method with the following prototype. It should inherit the `Sized` trait.

```
fn successor(self) -> Self;
```

Start by implenting this trait for basic types, like `u32`, or `i8`.

If someone wants to get the successor's successor of a value, they may try to call the function twice like that:

```
assert_eq!(12u32, 10u32.successor().successor());
```

And that would work! But wouldn't it be easier to simply provide a way to call the `successor` method multiple times with only one function call?

Add the `nth_successor` method to the `Successor` trait. That method should be automatically implemented for every type currently implementing that trait. In other words, the new method you just added must not break any of the existing implementations.

The method must be prototyped like this:

```
fn nth_successor(self, n: usize) -> Self;
```

Example:

```
assert_eq!(12u32, 10u32.nth_successor(2));
```

You. Must. Write. Tests.

Exercise 05: Return Of The DiamondTrap

```
turn-in directory:
    ex05/

files to turn in:
    src/main.rs  Cargo.toml

allowed dependencies:
```

What? I thought Rust wasn't an object oriented language? You liar! Was I just learning C++ again? - is what you would think if I

wasn't there to calm you down. I know C++ can be quite the... traumatic experience. Don't worry: Rust isn't like that (it's worse).

To begin, let's create a `ClapTrap` trait. ClapTraps are weird creatures only found in some underground C++ learning materials (also, a game, but let's not talk about that). A `ClapTrap` must implement the following methods:

```
/* Required Methods */
fn name(&self) -> &'static str;
fn health(&self) -> u32;
fn health_mut(&mut self) -> &mut u32;
fn energy(&self) -> u32;
fn energy_mut(&mut self) -> &mut u32;

/* Provided Methods */
fn attack(&mut self, target: &str);
fn take_damage(&mut self, amount: u32);
fn be_repaired(&mut self, amount: u32);
```

The *provided methods* should be implemented by default and use the required getters. The messages they print must include the name of the ClapTrap, and shouldn't work when the ClapTrap is either dead, or exhausted.

- The `attack` method must print a message indicating that the ClapTrap has dealt a certain amount of damages to a target.
- The `take_damage` method must print a message indicating that the ClapTrap has taken a certain amount of damages.
- The `be_repaired` method must indicate that the ClapTrap has gained a certain amount of health.

Now that you have your trait, we can create a type that implements it. Create a `BasicClapTrap` type that implenents the `ClapTrap` trait.

Now that we know how to create ClapTraps, let's describe what `ScavTrap`s and `FragTrap`s are. Both traits must inherit from the `ClapTrap` trait, and require the following methods to be implemented:

```
/* ScavTrap must have the following method: */
fn gate_keeper_mode(&self) -> bool;
fn gate_keeper_mode_mut(&mut self) -> &mut bool;

fn guard_gate(&mut self); // This one must be implemented by default.

/* FragTrap must have the following method: */
fn high_five_guys(&mut self); // It must be implemented by default.
```

When the `guard_gate` method is called, a message indicates that the ScavTrap enters Gate Keeper Mode. The message must include the name of the ScavTrap.

When the `high_five_guys` method is called, a message indicates that the FragTrap wants to high-five its friends.

Both actions consume one energy point. The behaviour is the same as with `ClapTrap`, once the energy reaches 0, the ScavTrap/FragTrap cannot do anything more.

Let's now create two additional types : a `BasicScavTrap` and a `BasicFragTrap`. The `BasicScavTrap` type must derive the `ScavTrap` trait, and the `BasicFragTrap` must derive the `FragTrap` trait.

Fiouh! That wasn't as funny as I thought... And it's not over ~

Let's now create a `DiamondTrap` trait. That trait must inherit from both `ScavTrap` and `FragTrap`. It must have one additional required method.

```
fn name(&self);

fn who_am_i(&self);
```

The `who_am_i` method must print a name different from the one used for the other `ClapTrap`, `FragTrap` and `ScavTrap` traits.

Let's now create a `BasicDiamondTrap` type, deriving the `DiamondTrap` trait.

Write a `main` function to showcase how those types and traits can be used.

Let's be clear about one thing: this kind of "diamond" inheritance pattern was already pretty hard to get right in C++. In Rust, it cannot really go wrong, but it is still a huge pain to write. In fact, any kind of Object Oriented-like inheritance is a pain to reproduce in Rust. This exercise should be the one and only time you write something like that. The language *requires* you to think about your code differently. It's not a flaw, it's a design choice. As we'll see in later modules, Rust gives you *a lot* of alternatives to that boring inheritance.

Exercise 06: A Type That You Can `print!`

```
turn-in directory:
    ex06/

files to turn in:
    src/main.rs  Cargo.toml

allowed dependencies:
```

As you might've understood in exercise 4, the `Debug` trait is used to print types with the `?` formatting option.

```
let a: i32 = 12;
println!("{a:?}");    // 12
println!("{:?}", a);  // 12
```

Implement the right trait for the following struct...

```
struct Greet;
```

... so that this `main` function compiles to display the text "Hey! How are you?".

```
fn main() {
    let greet = Greet;

    println!("{greet}");
}
```

```
>_ cargo run
Hey! How are you?
```

Exercise 07: Dynamic Dispatch

```
turn-in directory:
    ex07/

files to turn in:
    src/main.rs  Cargo.toml

allowed dependencies:
    ftkit
```

In Rust, everything has to be explicit. If you're not going to use concrete types, you'll have to write it explicitly.

Here is a Rust trait:

```
trait AllowValue {
    fn allow_value(&self, value: i32) -> bool;
}
```

The `allow_value` method either returns `true` or `false` depending on whether a given `value` is allowed or not.

Create four types, each of those types should implement the `AllowValue` trait. Their `allow_value` method should behave in the following ways:

- `Even` - Returns whether `value` is even.
- `Odd` - Returns whether `value` is odd.
- `Positive` - Returns whether `value` is positive or null.
- `Negative` - Returns whether `value` is negative or null.

Now, create a function named `find_original` that is able to take any value whose type implements the `AllowValue` trait. This function must try to call the `allow_value` method with multiple inputs to determine which of the four original types (`Even`, `Odd`, `Positive` or `Negative`) was passed. The result of the function must be displayed on the standard output. You are not allowed to use *generics* for this function. You have to use *trait objects* instead.

Create a `main` function that generates a random number, creates an instance of one of the created types, and gives it to the `find_original` function.

What you just used is called "dynamic dispatch". One of the next module's subject is "static dispatch" - a process that's both more interesting, and way more powerful.