Traits

Defining Traits

- Traits help define shared behaviour or methods between different structs
- This is how rust handles its "object oriented" behaviour of inheritance and parents/children
- Let's start by defining two structs that can be simiilar in nature
- We will be making everything here public for other crates to possibly use this in the future which is one main usecase for traits
- Example:

```
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}
```

    - Here we have two structs, `NewsArticle` and `Tweet`

- We can imagine that we would want to summarize a News Article and a Tweet
- So we may want to create a method for both
- Example:

```
impl NewNewsArticle{
    fn summarize(){}
}

impl Twitter{
    fn summarize(){}
}
```

    - Without any logic we can see that there is a similarity used here which is both can call summarize

    - In other words they have a common **trait** but different implementations

- Let's define a trait to make this easier
- We define a trait using the `trait` keyword
- Example:

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

    - Within the `trait Summary` block, we can define a set of shared methods
    - In this case it is only the `summarize` method
    - Also notice it only has a method signature and no implementation
    - This gives the user flexibility to treat a `trait` as either a `parent class` or a `interface`

- Now that we have a trait, we can implement the trait
- Example:

```
impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}
impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

Default Implementations

- We can add method body to the any trait methods
- This gives a default implementation
- However, if a Struct implements the method then the default implementation would be overriden by the Struct
- Also, you cannot do much with the `self` parameter of a trait unless you constrict it with another trait. So that it can only be implemented by those with the same restrictions

    - This is because it doesn't know what type will be using it and how it will be used

- Example:

```rust
pub trait Summary: std::fmt::Debug {
    fn summarize(&self) -> String{
        format!("Read More... {:?}", self)
    }
}


impl Summary for NewsArticle {

}
impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

- Here we add on hte `Debug` so we can format the Struct that inherits summarize
- We also need to add the `debug` annotation to the Struct
- Note that you still need the `impl` block, these block are how the `traits` get passed on to the Structs
- We will initialize the struct and we can print to see how it would work

    ```rust
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    let article = NewsArticle{
        author: String::from("John Smith"),
        headline: String::from("The Sky is falling"),
        content: String::from("The sky is not actually falling"),
        location: String::from("New York")

    };

    println!("1 new tweet: {}", tweet.summarize());
    println!("1 new article: {}", article.summarize());
    ```

        - You will see the article summary is just a debug summary but twitter overrides the method and puts in its own implementation

- A trait can have a deault implementation and a method signature in the same block.
- This means that a default implementation can call another method within a trait
- Let's look at a quick example:

```
pub trait Summary: std::fmt::Debug {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}

impl Summary for NewsArticle {
    fn summarize_author(&self) -> String {
        format!("Author: {}", self.author)
    }
}
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }

    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

    - Here we made a method signature for authors
    - Since it is only a signature within the trait, the Structs that inherit from the trait must implement it
    - However, we see that even though `summarize` uses the `summarize_author` method we do not have to implement it since it has a default implementation

Trait Parameters and Bounds

- Let's say we want to take in anything that implements or is a "child" of a trait as a parameter
- We may not care about the exact type but the implementation of the trait needs to be there
- We can include by using the `&impl <Trait>` type
- This allows us to take a reference to any type that implements `<Trait>`
- Example:

```
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

    - Here the type of parameter we take in is anything that implements `Summary`
    - We know that if it implements `Summary` then it must have the `summarize` method either as defualt or by overriding
- Now this is actually sugarcoating the real way rust reads this
- Logically this seems more similar to Java way of using generics with wild cards
    - We are basically saying that we want anytype that inherits upto `Summary` trait
- And in reality this can be rewritten so that it mimics Java's implementation using Generics
- The true syntax for the example above is:

```
pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

    - Here we can see that we actually take in a reference to a Generic that is **bounded** by the trait of `Summary`
    - In other words, we can take any generic type that is **upper bounded** by the `Summary` trait
- Continuing with this Java idea, in Java when pulling in children classes using Parent paramets, the object gets converted to the Parent type implicitly
    - This is very important as the object loses its access to any specific child class field
    - **However** the child object maintains its own implmentation of the parent's methods
    - This is because traits are only methods that are shared, not field, so the child gets to keep the methods but loses the fields.
- Either syntax is perfectly valid but the second one is more clear on what's happening and better to use as the code gets more complex
- The first one is a little easier to read for shorthand coding
- To make this point more clear, consider a simple example:

    - Imagine if you wanted to take two parameters that inherit from `Summary` and have the exact same type

    - With the `&impl` syntax there is a little ambiguity since the first parameter and second parameter doesn't have to be the same type

    - But we can specify this with the long hand `Generic` syntax instead ```rust pub fn notify(item: &impl Summary, item2: &impl Summary) { println!("Breaking news! {} and {}", item.summarize(), item2.summarize()); } pub fn notify2<T: Summary>(item: &T, item2: &T){ println!("Breaking news! {} and {}", item.summarize(), item2.summarize()); }

```
notify(&article, &tweet);
notify2(&article, &article2);
notify2(&tweet, &tweet2);
```
* In this quick example, we can see that in the first implementation we can pass in tweets and news articles equally
  * The first way doesn't restrict the generic type of the second

```
parameter to be the same as the first
* But the second implementation we had to pass in 2 articles or two
tweets because it required that both parameters were of the same
Generic type
  * This isn't a big deal when running the code but it can help clean
up the code if you know whether you want to have the exact same type
for all your paramters
```

- What if you want to restrict the **upper bound** more and you want to make sure it implements more than one trait
- You can use the `+` operator to mean `and` and say that you want to implement `trait1 + trait2`
- Example:

```rust
pub fn some_fun(item: &(impl Summary + Display)) {}
pub fn some_fun2<T: Summary + Display>(item: &T) {}
```

  - Notice that there are two ways of writing this, either the shorthand or the Generic signature
  - Also we can see that we require now that the parameter implements both `Summary` and `Display`
  - This can get a little messy with more complex parameters
- Imagine if you wanted to have two Generic parameters of different types that inherit from different traits
  - In this case the function signature can get a little complex and hard to read
  - Example:

```rust
fn some_function(t: &(impl Display + Clone), u: &(impl Clone +
Summary)) -> i32 {0}
fn some_function2<T: Display + Clone, U: Clone + Summary>(t: &T, u: &U)
-> i32 {0}
```

    - We can see that either way we define this function, it becomes very complicated and hard to read
- **Where** operator comes in handy here to help with readability of the code
  - The `where` operator is special and is writted between the return type and the `{}`
  - It helps organize the code a little better
  - Example: `rust fn some_function3<T, U>(t: &T, u: &U) -> i32 where T: Display + Clone, U: Clone + Summary { 0 }`
    * We are basically moving the bounds to after the return type is given for the function * This makes the code more readable and cleaner

Return Type with Traits

- If we want to return a type that is upper bounded by a specific trait we can use `-> impl <Trait>` return type
- This is very useful and gives more flexibility as to what can be returned from a function

- Let's say we want to return a object that can be summarized
- Example:

```rust
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    }
}
```

  - We can see that this returns a `Tweet` object but the return type is vague
  - Similar to pulling parameters, we lose access to all fields and methods not shared between the trait and its children.
  - However we maintain the Tweet implementation of `summarize()` even when returned
- Important to note that even though we can return any `impl Summary`, it cannot be of different types
  - We cannot return two different generic of `Summary` traits, rust has not implementation for this feature for now
  - Example: `rust fn returns_summarizable(switch: bool) -> impl Summary { if switch { NewsArticle { headline: String::from("Penguins win the Stanley Cup Championship!",), location: String::from("Pittsburgh, PA, USA"), author: String::from("Iceburgh"), content: String::from("The Pittsburgh Penguins once again are the best \ hockey team in the NHL.",), } } else { Tweet { username: String::from("horse_ebooks"), content: String::from("of course, as you probably already know, people",), reply: false, retweet: false, } } }` * This returns either a `Tweet` or a `NewsArticle` which isn't allowed with the `impl Trait` syntax * It has to return either a `Tweet` **or** a `NewsArticle`

Conditional Implement Trait

- Let's say our struct has a generic and depending of the type that is passed into the struct we want to implement different methods
- Example:

```rust
struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}
```

```
impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

  - So here we have a struct `Pair` that takes in a generic `<T>`
  - This means that the type `T` it takes in could be anything
  - This struct also has two `impl` blocks
  - In one of those `impl` blocks for any type we want to implement the `new` method
  - The other `impl` block is specifically for types that inherit from `Display + ParitalOrd`
    - This means that if we get a type that inheirts these traits it will have access to the function `cmp_display`
    - All other types cannot access this function
- Conditionally Implementing is another layer of flexibility to Structs for the user to help dictact what happens when a specific type is used in the Generic

Blanket Implementation

- Blanket Implementation is when you want to implement a trait on **any** type
- This uses generics when defining the implementation block rather than any specific Struct
- For example, in the Rust standard library this is done extensively
- The `to_string` method is one example of this:

```
impl<T: Display> ToString for T {
    // --snip--
}
```

  - Reading this is a little confusing but basically
  - For any type `<T>` that implements the trait `Display`, it can now also implment the trait `ToString` and use the defualt method `to_string()`