한 번에 끝내는
블록체인 개발 A to Z

Chapter 2

Rust Advanced

Chapter 2

Rust Advanced

Traits: Defining Shared Behavior

Traits Definition

A **trait** defines functionality a particular type has and can share with other types. We can use traits to define shared behavior in an abstract way. We can use trait bounds to specify that a generic type can be any type that has certain behavior.

Note: **Traits** are similar to a feature often called **interfaces** in other languages, although with some differences.

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

Implementing a Trait

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

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

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

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

```rust
use aggregator::{Summary, Tweet};

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

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

Default Implementation

Sometimes it's useful to have default behavior for some or all of the methods in a trait instead of requiring implementations for all methods on every type. Then, as we implement the trait on a particular type, we can keep or override each method's default behavior.

Default implementations can call other methods in the same trait, even if those other methods don't have a default implementation.

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

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

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

Traits as Parameters

Instead of a concrete type for the item parameter, we specify the **impl** keyword and the **trait** name. This parameter accepts any type that implements the specified trait. In the body of notify, we can call any methods on item that come from the Summary **trait**, such as summarize.

The **impl Trait** syntax works for straightforward cases but is actually syntax sugar for a longer form known as a **trait bound.**

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

```
pub fn notify(item1: &impl Summary, item2: &impl Summary) {
```

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

```
pub fn notify<T: Summary>(item1: &T, item2: &T) {
```

Specifying Multiple Trait Bounds

We can also specify more than one trait bound. Say we wanted notify to use display formatting as well as summarize on **item:** we specify in the notify definition that item must implement both **Display** and **Summary**. We can do so using the + syntax.

```
pub fn notify(item: &(impl Summary + Display)) {
```

```
pub fn notify<T: Summary + Display>(item: &T) {
```

Clearing Trait Bounds with `where` Clauses

Using too many trait bounds has its downsides. Each generic has its own trait bounds, so functions with multiple generic type parameters can contain lots of trait bound information between the function's name and its parameter list, making the function signature hard to read. For this reason, Rust has alternate syntax for specifying trait bounds inside a **where** clause after the function signature.

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
```

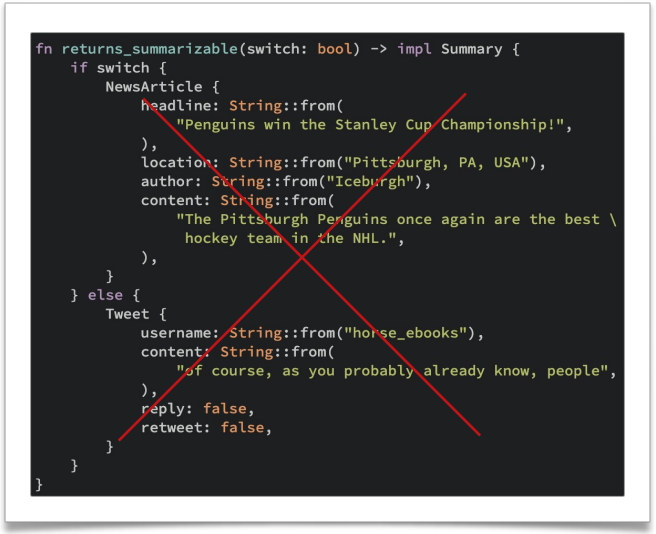
```
fn some_function<T, U>(t: &T, u: &U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
```

Returning Types that Implement Traits

We can also use the **impl Trait** syntax in the return position to return a value of some type that implements a trait.

Returning either a NewsArticle or a Tweet isn't allowed due to restrictions around how the **impl** Trait syntax is implemented in the compiler.

```
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,
    }
}
```



Conditionally Implement Methods

We can also conditionally implement a trait for any type that implements another trait. Implementations of a trait on any type that satisfies the trait bounds are called **blanket implementations** and are extensively used in the Rust standard library.

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

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

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);
        }
    }
}
```