**Rust Trait by Example**

[[Xiaodong Yan](https://yanxd.medium.com/?source=post_page-----2d37d05f03f0--------------------------------)](https://yanxd.medium.com/?source=post_page-----2d37d05f03f0--------------------------------)

[Xiaodong Yan](https://yanxd.medium.com/?source=post_page-----2d37d05f03f0--------------------------------)

·

Follow

3 min read

·

5 days ago

6

1

I’m excited to share a couple of topics on effectively utilizing traits in Rust.



**Supertrait**

In Rust, a ***supertrait*** is a trait that imposes additional requirements beyond those specified by another trait. A supertrait extends the functionality of a trait by adding more methods or functionality that the implementing types must fulfill. It establishes a relationship where any type implementing the ***subtrait*** must also implement its supertrait.

For example:

trait Area {  
 fn calculate\_area(&self) -> f64;  
}  
  
// Area is a supertrait of Shape, Shape is a subtrait of Area  
trait Shape: Area {  
 fn describe(&self);  
}  
  
struct Rectangle {  
 length: f64,  
 width: f64,  
}  
  
impl Area for Rectangle {  
 fn calculate\_area(&self) -> f64 {  
 self.length \* self.width  
 }  
}  
// To impl Shape we must impl its supertrait Area like above  
impl Shape for Rectangle {  
 fn describe(&self) {  
 println!("I am a rectangle with length {} and width {}", self.length, self.width);  
 // We can access supertrait's method using self  
 println!("Area: {}", self.calculate\_area());  
 }  
}  
  
fn print\_shape\_info<T: Shape>(shape: &T) {  
 shape.describe();  
}  
  
fn main() {  
 let rectangle = Rectangle { length: 4.0, width: 3.0 };  
  
 print\_shape\_info(&rectangle);  
 // Output:  
 // I am a rectangle with length 4 and width 3  
 // Area: 12  
}

**Trait Extension**

Trait extension in Rust allows you to add new methods or behaviour to existing types by implementing traits for those types. It’s a powerful concept that enables you to extend the functionality of types without modifying their original definitions.

Here’s an example to illustrate trait extension:

// Define a trait with a method  
trait Hello {  
 fn say\_hello(&self);  
}  
  
// Implement the trait for the type i32  
impl Hello for i32 {  
 fn say\_hello(&self) {  
 println!("Hello, I'm {}!", self);  
 }  
}  
  
fn main() {  
 let number = 42;  
 number.say\_hello(); // Calling the method from the trait extension  
}

**Blanket Implementation**

Blanket Implementation is a generic extension that allows you to conditionally implement a trait for any type that implements another trait:

use std::fmt::Debug;  
  
trait Displayable {  
 fn display(&self);  
}  
  
impl<T: Debug> Displayable for T {  
 fn display(&self) {  
 println!("Hello {:?}", self);  
 }  
}  
  
#[derive(Debug)]  
struct MyStruct {  
 value: i32,  
}  
  
fn main() {  
 let instance = MyStruct { value: 42 };  
  
 instance.display(); // This will print: Hello MyStruct { value: 42 }  
}

**Method Disambiguation**

Rust compiler prevents ambiguity in method resolution when a type implements multiple traits with conflicting methods.

Here is an example that introduce this scenario and solutions:

1. Explicit casting to access a specific trait’s methods
2. Using a trait object

trait A {  
 fn greet(&self) -> &'static str {  
 "Hello from A"  
 }  
}  
  
trait B {  
 fn greet(&self) -> &'static str {  
 "Hello from B"  
 }  
}  
  
struct C;  
  
impl A for C {}  
impl B for C {}  
  
fn main() {  
 let c = C;  
 // println!("{}", c.greet()); // Compiling error: multiple `greet` found  
   
 // Option 1: Explicit casting to access a specific trait's methods  
 println!("{}", A::greet(&c)); // Output: "Hello from A"  
 println!("{}", B::greet(&c)); // Output: "Hello from B"  
  
 // Option 2: Convert c to a trait object of type A  
 let a: &dyn A = &c;  
 println!("{}", a.greet()); // Output: "Hello from A"  
}

**Reference**

**[The Rust Programming Language](https://doc.rust-lang.org/book/ch19-03-advanced-traits.html?source=post_page-----2d37d05f03f0--------------------------------" \t "_blank)**

[We first covered traits in the "Traits: Defining Shared Behavior" section of Chapter 10, but we didn't discuss the more…](https://doc.rust-lang.org/book/ch19-03-advanced-traits.html?source=post_page-----2d37d05f03f0--------------------------------" \t "_blank)

[doc.rust-lang.org](https://doc.rust-lang.org/book/ch19-03-advanced-traits.html?source=post_page-----2d37d05f03f0--------------------------------" \t "_blank)