Structs, Polymorphism, Dynamic Dispatch

Dr. Nathaniel Lahn: ITEC 320

Questions

- What is a class in Java?
- What is an interface in Java?
- What is *polymorphism?*
- How do we support similar functionality in Rust?
- How is polymorphism supported by the ownership model?

What is a class in Java?

What is a class in Java?

- A class in Java is trying to accomplish two things simultaneously:
  - A class is a *data type*, which includes:
    - A set of possible values (data fields)
    - A set of possible operations (methods)
  - A class is also a *module*
    - You can import a class
    - public / private visibility
    - namespacing
- In Rust, we split these ideas up into *structs* and *modules*

Structs in Rust

```
struct Rectangle {
    width: u32,
    height: u32,
}
```

The primary purpose of a struct in Rust is to describe a *data type that involves multiple fields*.

Field Visibility

- You can access struct fields using "." notation, just like in Java
  - But only if it is accessible…
    - It's like Java : There are rules about *visibility modifiers*
    - *By default, struct fields are **private***
- You can make fields public using the "pub" keyword below:

```
struct Rectangle {
    pub width: u32,
    pub height: u32,
}
```

Implications of private

- In Java, what does “private” mean?

Implications of private

- In Java, what does “private” mean?
  - It’s only directly accessible *inside the class. Anyone who wants to access the data outside the class must do so* **indirectly.**
- In Rust, what does “private” mean?
  - It’s only accessible within the current *module*. **Remember, structs are not modules in Rust**
  - We’ll talk about how to create modules later, but for now, realize that every program we’ve looked at has exactly one (default) module

Struct Functions Can Be Separate

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn area(r : &Rectangle) -> u32 {  // Note that the & is necessary for borrowing
    return r.width * r.height;
}
```

Method Syntax Vs. Function Syntax

- Imagine that we have a rectangle variable named “rectangle”
- Java allows us to call methods like this:
  - double area = rectangle.area();

- However, to call our area function in Rust, we would use:
  - let area : f64 = area(&rectangle);

- Either way, “area” is a sort of “function”, but…
  - Java passes the rectangle as an *implicit parameter*
  - Rust passes the rectangle as an *explicit* parameter
- In reality, they are doing the same sort of thing – calling a function.

Methods in Rust

- In Rust, a method is a special type of function that is associated with a *struct implementation block*
- To write a Rust method, write an impl {} block like so:

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

Methods in Rust

- In Rust, a method is a special type of function that is associated with a *struct implementation block*
- To write a Rust method, write an impl {} block like so:

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

- An implementation block is a way to list *functions* that are *associated* with a *struct type* **and** have an instance of that type as the *first parameter*

Method Vs. Functions in Rust

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

- What does &self mean?
- We can replace it with this:
  - fn area(self : &Self) -> u32
- What does Self mean? We can replace it with:
  - fn area(self : &Rectangle) -> u32

- self (lowercase) means “the name of the first parameter”
- Self (uppercase) means “the type of the struct we are implementing”
- self has type Self

Polymorphism

- Rust is not technically an object oriented programming language
  - (Because it doesn’t allow for inheritance)
- However, Rust does allow for *polymorphism*
  - What exactly is polymorphism?
  - How does Rust polymorphism compare to Java polymorphism?

- Let’s look at Java first….
  - See Polymorphism.java
  - Then, see the corresponding Rust example on D2L.

Polymorphism in Rust

- Requirements:
  - Two or more types that share a similar *interface*, but have different *implementations* for those interfaces
  - A single *trait* that describes the *shared interface*
  - Multiple implementation blocks (impl) that describe the precise *implementations of the interface* (one per type that implements the trait)
  - Whenever the Trait is know, but *not the underlying type*, then you must use "Box<dyn trait_name>" as the data type
    - Ex: Vec<Box<dyn Shape>>
- Why all the hoopla?
  - Because memory diagramming → See the example diagram on D2L.

Structs vs. Classes

- Java classes are like structs except:
  - Java classes have methods *inside them*
    - *It's like methods are a **part** of the class*
  - Rust structs have methods *in a separate implementation block*
    - *It's like methods are **associated** with a type, but they can be seen as "syntactic sugar" for what are really standalone functions.*

- Also, as mentioned before, Java classes are also modules, while Rust separates out the idea of a module and a struct.
  - More on Modules in Rust later…

Interfaces vs. Traits

- Rust traits look a lot like Interfaces in Java
- Similar because: They both describe “interfaces”, in the general sense of the word
  - An interface is simply a set of function signatures that describe how two parts of a program interact
  - Interfaces do not need to be “object oriented”
- Different because : Java Interfaces are *also* types, while Rust traits are *just* interfaces
- Similar to how classes in Java are *also* modules, while Rust structs are *just types*
- *Yes, it’s confusing --- blame Java*