**Understanding OOP in RUST**

[[](https://adarsharegmi121.medium.com/?source=post_page-----7f70bb336fe7--------------------------------)](https://adarsharegmi121.medium.com/?source=post_page-----7f70bb336fe7--------------------------------)

[Adarsha Regmi](https://adarsharegmi121.medium.com/?source=post_page-----7f70bb336fe7--------------------------------)

·

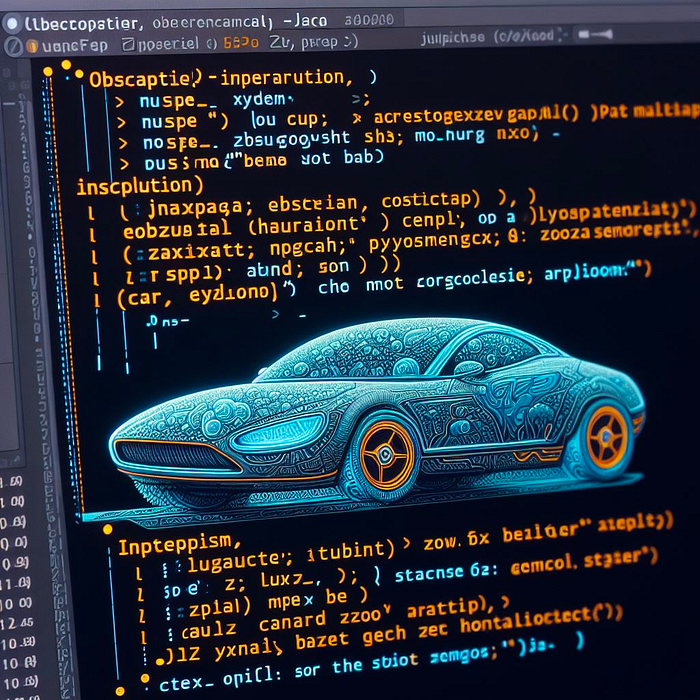
Follow

3 min read

·

10 hours ago

Overview:: Rust is gaining popularity over the years because of its safety. And for those interested to switch career to RUST and learn I think OOP is where we look for. I have created this article as I went through same process to grasp familiarity to RUST.



OOps in RUST Lang

**1. Traits and Structs**

**With compared old fashioned languages like C++, Java RUST leverages approach of using traits and struct for achieving such functionalities.**

>> Traits :: similar to interface where we define the method names.

>> Structs :: create custom data types by grouping together variables (fields) under a single name.

struct Person {  
 name: String,  
 age: u32,  
}  
  
let my\_person = Person {  
 name: String::from("Alice"),  
 age: 30,  
};

>> impl :: implement functionality or behavior for a particular struct, trait, or enum

struct Circle {  
 radius: f64,  
}  
  
impl Circle {  
 // Method to calculate area for Circle  
 fn calculate\_area(&self) -> f64 {  
 std::f64::consts::PI \* self.radius \* self.radius  
 }  
}  
  
let my\_circle = Circle { radius: 3.0 };  
let area = my\_circle.calculate\_area();

Whole example:: >>

// Define a trait [equivalent to interface]  
trait Sound {  
 fn make\_sound(&self);  
}  
  
// Implementing the trait for a struct  
struct Dog;  
  
impl Sound for Dog {  
 fn make\_sound(&self) {  
 println!("Woof!");  
 }  
}  
  
// Using the struct and trait  
let my\_dog = Dog;  
my\_dog.make\_sound(); // Output: Woof!

**2. Encapsulation:**

*use of ‘pub’ keyword allows the visibility being equivalent to public keyword in other languages.*

struct Person {  
 pub name: String,  
 age: u8,  
}  
  
impl Person {  
 pub fn new(name: String, age: u8) -> Self {  
 Person { name, age }  
 }  
  
 pub fn greet(&self) {  
 println!("Hello, my name is {}.", self.name);  
 }  
}

**3. Inheritance-Like Behavior with Traits:**

*While Rust doesn’t have built-in class-based inheritance, it achieves similar behaviors using traits. Traits can inherit other traits, enabling a form of code reuse and allowing different structs to share behavior.*

trait Animal {  
 fn eat(&self);  
}  
  
trait Sound: Animal {  
 fn make\_sound(&self);  
}  
  
struct Dog;  
  
impl Animal for Dog {  
 fn eat(&self) {  
 println!("Dog is eating.");  
 }  
}  
  
impl Sound for Dog {  
 fn make\_sound(&self) {  
 println!("Woof!");  
 }  
}

**4. Polymorphism and Dynamic Dispatch:**

*Rust also supports polymorphism and dynamic dispatch through trait objects, allowing for flexibility in working with different types.*

// Define a trait  
trait Shape {  
 fn area(&self) -> f64;  
}  
  
// Implement trait for multiple structs  
struct Circle {  
 radius: f64,  
}  
  
impl Shape for Circle {  
 fn area(&self) -> f64 {  
 std::f64::consts::PI \* self.radius \* self.radius  
 }  
}  
  
struct Square {  
 side: f64,  
}  
  
impl Shape for Square {  
 fn area(&self) -> f64 {  
 self.side \* self.side  
 }  
}  
  
// Create a function using trait objects  
fn print\_area(shape: &dyn Shape) {  
 println!("Area: {}", shape.area());  
}  
  
// Usage  
let circle = Circle { radius: 3.0 };  
let square = Square { side: 4.0 };  
  
print\_area(&circle); // Output: Area: 28.274333882308138  
print\_area(&square); // Output: Area: 16.0

**Conclusion**

Rust’s take on OOP might seem different from what you’re used to in languages like Java or Python, but its focus on safety, concurrency, and performance makes it a compelling choice. Using traits, structs, and the concept of ownership, Rust enables developers to build robust, scalable, and safe systems while embracing modern programming paradigms.