Student Name: ______________________________________ Student ID: ____________________ TA: ______________________

Rust Lab 14 – Traits, Bounds, Associated Types & Iterators

8/10/2025

Lab 1: Basic Traits and Generics

Objective This lab focuses on the fundamentals: defining a new trait, implementing it for custom data structures, and using it to write generic functions. This covers the "The Need for Traits" and basic implementation syntax from your slides.

Problem Description

You are building a small geometry library. You need a way to handle different shapes in a uniform manner. Your task is to define a common interface for any shape that can calculate its area and perimeter.

Requirements

1. Define a Trait:
   - Create a trait named ShapeProperties.
   - This trait should declare two methods:
     - area(&self) -> f64 which calculates the area.
     - perimeter(&self) -> f64 which calculates the perimeter.
2. Create Structs:
   - Define a struct Rectangle with fields width: f64 and height: f64.
   - Define a struct Circle with a field radius: f64.
   - Use the #[derive(Debug)] attribute on both structs so they can be easily printed.
3. Implement the Trait:
   - Implement the ShapeProperties trait for the Rectangle struct.
     - Area: width * height
     - Perimeter: 2 * (width + height)
   - Implement the ShapeProperties trait for the Circle struct.
     - Area: π * radius² (You can use std::f64::consts::PI).
     - Perimeter (Circumference): 2 * π * radius
4. Create a Generic Function:
   - Write a generic function print_details<T: ShapeProperties>(shape: &T).
   - This function should take a reference to any object that implements ShapeProperties.
   - Inside the function, print the shape's details using the Debug format, then print its calculated area and perimeter.

Example main Function

```
fn main() {
    let rect = Rectangle { width: 10.0, height: 5.0 };
    let circle = Circle { radius: 7.5 };
    println!("--- Rectangle ---");
    print_details(&rect);
    println!("\n--- Circle ---");
    print_details(&circle);
}
```

Expected Output

```
--- Rectangle ---
Shape: Rectangle { width: 10.0, height: 5.0 }
Area: 50
Perimeter: 30
--- Circle ---
Shape: Circle { radius: 7.5 }
Area: 176.71458676442586
Perimeter: 47.12388980384689
```

TA Check: ____

Lab 2: Operator Overloading and Standard Traits

Objective

This lab explores implementing standard library traits to integrate your custom types with Rust's core language features, like the + operator and formatted printing ({}). This covers topics like std::ops::Add, associated types, and std::fmt::Display.

Problem Description

You need to create a Vector2D struct to represent a point or vector in 2D space. You want to be able to add two vectors together using the + operator and print them in a clean, user-friendly format.

Requirements

1. Define the Struct:
   - Create a struct Vector2D with fields x: f32 and y: f32.
   - Use #[derive(Debug, Copy, Clone)]. As the slides note, Copy and Clone make passing and using the struct more ergonomic, especially when implementing Add.
2. Implement Add Trait:
   - Implement the std::ops::Add trait for Vector2D.
   - The add method should take another Vector2D (rhs) and return a new Vector2D where the x and y components are the sum of the operands' components.
   - Remember to define the associated type type Output = Self; as shown in the slides.
3. Implement Display Trait:
   - Implement the std::fmt::Display trait for Vector2D.
   - The fmt method should format the vector as (x, y). For example, a vector with x=3.1 and y=-2.5 should be printed as (3.1, -2.5).

Example main Function

```
use std::ops::Add;
use std::fmt;
// Your struct and impl blocks go here
fn main() {
    let v1 = Vector2D { x: 5.0, y: 2.0 };
    let v2 = Vector2D { x: -1.0, y: 3.0 };
    let v3 = v1 + v2; // Uses your Add implementation
    println!("Vector 1: {}", v1);     // Uses your Display implementation
    println!("Vector 2: {}", v2);
    println!("v1 + v2 = {}", v3);
    println!("Debug format: {:?}", v3); // Uses the derived Debug implementation
}
```

Expected Output

```
Vector 1: (5, 2)
Vector 2: (-1, 3)
v1 + v2 = (4, 5)
Debug format: Vector2D { x: 4.0, y: 5.0 }
```

TA Check: ____

Lab 3: Dynamic Dispatch and Trait Objects

Objective

This lab contrasts static and dynamic dispatch. You will build a heterogeneous collection—a list containing different types that all share the same behavior—using trait objects (dyn Trait). This directly addresses the "Static vs. Dynamic Dispatch" section of your slides.

Problem Description

You are creating a simple UI framework. The framework needs to manage a list of different drawable components (like buttons and text labels). Each component has a different internal structure, but they all share the ability to be "rendered" as a string. Your task is to create a list of these different components and render them all in a single loop.

Requirements

1. Define a Trait:
   - Create a trait named Renderable.
   - It should have one method: render(&self) -> String.
2. Create Structs:
   - Define a Button struct with a label: String.
   - Define a Label struct with text: String.
   - Define a Container struct with a name: String and children: Vec<Box<dyn Renderable>>.
3. Implement the Trait:
   - For Button: The render method should return a string like "Button: [Submit]".
   - For Label: The render method should return a string like "Label: 'Welcome to my App!'".

   - For Container: The render method should return a string that shows its name and its children's rendered output, indented. For example: Container ('Login Form') {\n Label: 'Username'\n Button: [Submit]\n}.
4. Use Dynamic Dispatch:
   - In your main function, create a Vec<Box<dyn Renderable>>. This is your heterogeneous list.
   - Push instances of Button, Label, and Container onto the vector. Remember to wrap them in Box::new().
   - The Container should itself contain a Button and a Label.
   - Loop through the vector and call the render method on each trait object, printing the result.

Example main Function

```
// Your trait and struct/impl blocks go here
fn main() {
  // Create a container that holds other renderable items
  let mut inner_container = Container {
    name: "Login Form".to_string(),
    children: Vec::new(),
  };
  inner_container.children.push(Box::new(Label { text: "Username".to_string() }));
  inner_container.children.push(Box::new(Button { label: "Submit".to_string() }));
  // Create the main screen list
  let mut screen: Vec<Box<dyn Renderable>> = Vec::new();
  screen.push(Box::new(Label { text: "Welcome to my App!".to_string() }));
  screen.push(Box::new(inner_container));
  screen.push(Box::new(Button { label: "Sign Out".to_string() }));
  // Render everything on the screen
  println!("--- Rendering Screen ---");
  for component in screen {
     println!("{}", component.render());
  }
}
```

Expected Output

```
--- Rendering Screen ---
Label: 'Welcome to my App!'
Container ('Login Form') {
  Label: 'Username'
  Button: [Submit]
}
Button: [Sign Out]
```

TA Check: ____