Day-3

Generic Types

Generics allow writing flexible and reusable code that works with different data types without sacrificing safety. You can define functions, structs, enums, and methods using generic types.

Syntax:

```rust
fn function_name<T>(param: T) { ... }
struct StructName<T> { field: T }
```

```rust
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("The largest number is {result}");

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("The largest char is {result}");
}


```

Example

```rust
fn largest<T: PartialOrd>(a: T, b: T) -> T {
    if a > b {
        a
    } else {
        b
    }
}

fn main() {
    println!("{}", largest(5, 10));       // Output: 10
    println!("{}", largest(3.4, 2.1));    // Output: 3.4
    println!("{}", largest('a', 'z'));
    println!("{}", largest(f64::NAN, 5.0));    // output?
}


```

- `T: PartialOrd` means "T must implement the `PartialOrd` trait" so that `a > b` can compile.
- The function compares `a` and `b` and returns the larger one.
- the `PartialOrd` trait is used to enable comparison operations (like `<`, `>`, `<=`, `>=`) on values of a type. It is part of Rust's standard library traits that deal with ordering.

Rust developers commonly use:

	Name	Meaning
1	`T`	Type (generic/default)
2	`U`	Another type
3	`E`	Error type
4	`K`	Key (for maps)
5	`V`	Value (for maps)
6	`R`	Return type / Reader
7	`Item`	Element of a collection
8	`Input`, `Output`	For clarity in custom traits

Example with T and U:

</> Plain Text

```
fn combine<T, U>(a: T, b: U) {
    println!("Values: {:?}, {:?}", a, b);
}

fn main() {
    combine(5, "hello");
    combine(3.14, true);
}


//T is the type of a (i32, f64, etc.)
//U is the type of b (&str, bool, etc.)



```

</> Plain Text

```
//Same logic, better naming:
fn combine<Payload: std::fmt::Debug, Metadata: std::fmt::Debug>(payload: Payload, metadata: Metadata) {
 println!("Values: {:?}, {:?}", payload, metadata);
}
```

```

fn main() {
 combine(5, "hello");
 combine(3.14, true);
}
```

Exercise:

- A generic API function that returns `Result<Response<Data>, Error>`
- Use a custom `Payload` struct as input
- Use custom names instead of `T` and `E`
- Simulate a mock API call that can succeed or fail

Traits

Traits define **shared behavior** across types, similar to interfaces in other languages.

Trait Types Overview:

	Trait Type	Description
1	**Marker Trait**	A trait with **no methods**, used to "mark" a type with a behavior or meaning.
2	**Auto Trait**	Automatically implemented by Rust for types unless explicitly opted out.
3	**Unsafe Trait**	Must be implemented with caution—signals that unsafe code might be involved.

Marker Trait

Description:

Used to tag a struct/enum with metadata or signal certain behavior.

Plain Text

```
trait Serializable {} // Marker trait

struct User {
    name: String,
}

impl Serializable for User {}
```

Even though `Serializable` has no methods, you can use it in bounds:

Plain Text

```
fn serialize<T: Serializable>(data: T) {
    println!("Data can be serialized");
}
```

Auto Trait

Description:

Traits that are automatically implemented by the compiler.

Common example: `Send` and `Sync`

```
fn is_send<T: Send>() {}

fn main() {
    is_send::<i32>(); // i32 is Send
}

//You can use !Trait (negative impl) to opt out:

struct NotSend;
unsafe impl !Send for NotSend {}
```

Unsafe Trait

Description:

Traits that may violate Rust’s safety guarantees and must be implemented manually with caution.

Example:

```
unsafe trait Dangerous {}

struct MyType;

unsafe impl Dangerous for MyType {}

//Only use this when you’re handling raw pointers, FFI, or unsafe system calls.
```

Compare with the other Language:

	Language	Concept
1	Java	`interface`
2	TypeScript	`interface`
3	C++	abstract class
4	Rust	`trait`

Define the Trait:

```
trait Animal {
    fn speak(&self) -> String;
}
```

Implement it for a Struct:

```
struct Dog;

impl Animal for Dog {
    fn speak(&self) -> String {
        String::from("Woof!")
    }
}

```

```
struct Cat;

impl Animal for Cat {
    fn speak(&self) -> String {
        String::from("Meow!")
    }
}
```

</> Plain Text

```
Trait (e.g., Animal)
    ↓          ↓
Implemented for:
  Dog      → says Woof!
  Cat      → says Meow!
```

Use it like an Interface:

</> Plain Text

```
fn make_animal_speak<T: Animal>(animal: T) {
    println!("{}", animal.speak());
}

fn main() {
    let dog = Dog;
    let cat = Cat;

    make_animal_speak(dog);
    make_animal_speak(cat);
}



```

The make_animal_speak function is saying that `T` must implement the methods and behavior specified in the `Animal` trait.

list of standard Rust traits

1. Debug

Allows formatting a value using the `{:?}` formatter for debugging.

2. Clone

Enables creating a deep copy of a value.

3. Copy

Indicates values can be duplicated simply by copying bits (used with simple types like integers).

4. PartialEq

Enables equality comparisons using `==` and `!=` .

5. Eq

Marker trait for types that have full equivalence (used with `PartialEq`).

6. PartialOrd

Allows partial ordering comparisons (`<` , `>` , `<=` , `>=`), returns `Option<Ordering>` .

7. Ord

Enables total ordering comparisons, returns `Ordering` .

8. Default

Provides a default value for a type using `::default()` .

9. Drop

Called automatically when a value goes out of scope (for cleanup logic).

10. From / Into

Used for value-to-value conversions between types.