Shared Functionality | Generic Functions

What Are Generic Functions?

- A way to write a function that can have a single parameter with multiple data types
- Trait is used as function parameter instead of data type
  - Function depends on existence of functions declared by trait
- Less code to write
  - Automatically works when new data types are introduced

Quick Review: Traits

```rust
trait Move {
    fn move_to(&self, x: i32, y: i32);
}

struct Snake;
impl Move for Snake {
    fn move_to(&self, x: i32, y: i32) {
        println!("slither to ({},{})", x, y);
    }
}

struct Grasshopper;
impl Move for Grasshopper {
    fn move_to(&self, x: i32, y: i32) {
        println!("hop to ({},{})", x, y);
    }
}
```

Quick Review: Traits

```rust
trait Move {
    fn move_to(&self, x: i32, y: i32);
}

fn make_move(thing: impl Move, x: i32, y: i32) {
    thing.move_to(x, y);
}

let python = Snake {};
make_move(python, 1, 1);
// Output:
// slither to (1,1)
```

Generic Syntax

```rust
fn function<T: Trait1, U: Trait2>(param1: T, param2: U) {
    /* body */
}
```

Generic Syntax

```rust
fn function<T: Trait1, U: Trait2>(param1: T, param2: U) {
    /* body */
}

fn function<T, U>(param1: T, param2: U)
where
    T: Trait1 + Trait2,
    U: Trait1 + Trait2 + Trait3,
{
    /* body */
}
```

Generic Example

```
fn make_move(thing: impl Move, x: i32, y: i32) {
    thing.move_to(x, y);
}

fn make_move<T: Move>(thing: T, x: i32, y: i32) {
    thing.move_to(x, y);
}
```

Generic Example

```rust
fn make_move(thing: impl Move, x: i32, y: i32) {
    thing.move_to(x, y);
}

fn make_move<T>(thing: T, x: i32, y: i32)
where
    T: Move,
{
    thing.move_to(x, y);
}
```

Which syntax to choose?

```rust
fn function(param1: impl Trait1, param2: impl Trait2) {
    /* body */
}


impl Move for Grasshopper {
    fn move_to(&self, x: i32, y: i32) {
        println!("hop to ({},{})", x, y);
    }
}
```

Which syntax to choose?

```rust
fn function(param1: impl Trait1, param2: impl Trait2) {
    /* body */
}

fn function<T: Trait1, U: Trait2>(param1: T, param2: U) {
    /* body */
}
```

Which syntax to choose?

```rust
fn function(param1: impl Trait1, param2: impl Trait2) {
    /* body */
}

fn function<T: Trait1, U: Trait2>(param1: T, param2: U) {
    /* body */
}

fn function<T, U>(param1: T, param2: U)
where
    T: Trait1 + Trait2,
    U: Trait1 + Trait2 + Trait3,
{
    /* body */
}
```

Details - Monomorphization

```rust
trait Move {
    fn move_to(&self, x: i32, y: i32);
}
fn make_move<T: Move>(thing: T, x: i32, y: i32) {
    thing.move_to(x, y);
}
make_move(Snake {}, 1, 1);
make_move(Grasshopper {}, 3, 3);
```

Details – Monomorphization

```rust
trait Move {
    fn move_to(&self, x: i32, y: i32);
}
fn make_move<T: Move>(thing: T, x: i32, y: i32) {
    thing.move_to(x, y);
}
make_move(Snake {}, 1, 1);
make_move(Grasshopper {}, 3, 3);
```

```rust
fn make_move(thing: Snake, x: i32, y: i32) {
    thing.move_to(x, y);
}
fn make_move(thing: Grasshopper, x: i32, y: i32) {
    thing.move_to(x, y);
}
```

Recap

- Generics let you write one function to work with multiple types of data
- Generic functions are “bound” or “constrained” by the traits
  - Only able to work with data that implements the trait
- Three syntaxes available:

```rust
fn func(param: impl Trait) {}
fn func<T: Trait>(param: T) {}
fn func<T>(param: T) where T: Trait {}
```