COLM ONEACRE

GENERICS

WHY GENERICS

- Flexible and reusable code
- Define functions, structs, enums, and traits with placeholders

STRUCTS

- Define generics name in angle brackets <> after the struct's name
- Generally start with 'T' and follow the alphabet
- Several different generics can be used
- Allows for different instances to be created using different value type

```
struct Point<T> {
    x: T,
    y: T,
}
```

```
struct Point<T, U> {
    x: T,
    y: U,
}
```

```
struct Point {
    x: i32,
    y: i32,
}
fn main() {
    let origin = Point {x: 0, y:0};
}
```

Will give a 'mismatched type' error if anything but i32 values are used

IMPLEMENTATION BLOCKS

- Mention generics after *impl keyword and after the name of struct*
- In function we must clarify what type each variable will be and what we will return
- Generic implementation allows for other specific implementations that Rust can differentiate

```
impl<T, U> Point<T, U> {}
```

```
impl<T, U> Point<T, U> {
    fn new(x: y, y: U) -> Point<T, U> {
        Point {x, y}
    }
}
fn main() {
    let origin = Point::new(0, 0);
    let p1 = Point::new(1.0, 4.0);
    let p2 = Point::new(5, 5.0);
}
```

SPECIALIZATION

- Using generics and concrete implementations allows for specialization of different types
- Group methods for specific types to optimize and customize
- Specialized implementations are less flexible but paired with generics can be very powerful

```
impl Point<i32, i32> {
    fn printing(&self) {
        println!("The values of the coordinates are {}, {}", self.x, self.y);
    }
}
```

DUPLICATE DEFINITIONS

Functions of the same name in the same implementation blocks are not allowed

This is allowed in different concrete implementation blocks

```
impl Point<i32, i32> {
    ...
    fn new(x: i32, y: i32) -> Point<i32, i32> { // Error
        Point {x, y}
    }
}
```

```
impl Point<i32, i32> {
    ...
    fn new_1(x: i32, y: i32) -> Point<i32, i32> {
        Point {x, y}
    }
}
```

```
impl Point<i32, i32> {
    fn printing(&self) {
        println!("The values of the coordinates are {}, {}", self.x, self.y);
    }
    ...
}
impl Point<f64, f64> {
    fn printing(&self) {
        println!("The values of the coordinates are {}, {}", self.x, self.y);
    }
}
...
```

FREE FUNCTIONS

- Not tied to any structs, enums, or traits
- Generics are named three times
    - What generics to be used
    - What type the variables will be
    - What type will be returned
- Customer retention
- Operational efficiency

```rust
fn add_points<T, U>(p1: &Point<T, U>, p2: &Point<T, U>) -> Point<T, U> {
    unimplemented!();
}
```

MONOMORPHIZATION

- Generics do not cost much at runtime
- Rust creates specialized concrete implementations of each set of generic types used (Static dispatch)
- Occurs at compile time
- One problem this presents is 'code bloat', the generation of multiple copies of a function

```rust
fn main() {
    let origin = Point::new(0, 0);
    let p1 = Point::new(1.0, 4.0);

    add_points(&origin, &origin);
    add_points(&p1, &p1);
}
```

```rust
fn add_points_i32(p1: &Point<i32, i32>, p2: &Point<i32, i32>) -> Point<i32, i32> {
    unimplemented!();
}
fn add_points_f64(p1: &Point<f64, f64>, p2: &Point<f64, f64>) -> Point<f64, f64> {
    unimplemented!();
}
```

Thank you