Using Traits and Generics

Zachary Bennett
Lead Software Developer

@z_bennett_ | zachbennettcodes.com

Traits define shared behavior

Example Trait

```
// Any type that implements this trait can calculate a total of some kind
pub trait Total {
    fn calc_total(&self) -> i64;
}

// Any type that implements this trait can calculate a description
pub trait Describe {
    fn gen_description(&self) -> String;
}
```

Using Traits

```
struct Snack {
  name: String;
  item_number: i32;
  cost: f64;
}

struct Coffee {
  name: String;
  cost: f64;
  size: String;
}

impl Describe for Snack {
  fn get_description(&self) -> String {
    format!(“{} / Item #: {} / Cost: {}”, self.name, self.item_number, self.cost)
  }
}

impl Describe for Coffee {
  fn get_description(&self) -> String {
    format!(“{} {} costing {}”, self.size, self.name, self.cost)
  }
}
```

Demo

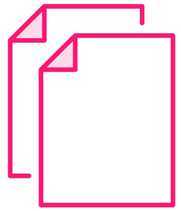
Traits

- Creating traits
- Implementing traits

Using Generics

Abstract types!

Why Generics?



Help reduce code duplication

More type safety - compile time guarantees

Flexible code but restrictive as necessary

Generic Structs

```
struct Coffee<‘a, T, U> {
  name: &’a str;
  cost: T;
  size: U;
}

let my_coffee = Coffee { name: “Drip”, cost: 4.99, size: 3 };

let my_other_coffee = Coffee { name: “Drip”, cost: 3.50, size: “Grande” };
```

Generic Enums

```
enum Result<T, E> {
  Ok(T),
  Err(E)
}
```

Generic Functions with Type Parameters

```
fn best<T>(items: Vec<T>) -> &T {
  let mut best = &items[0];

  for item in items {
    if item.calc_rating() > best.calc_rating() {
      best = item;
    }
  }

  best
}
```

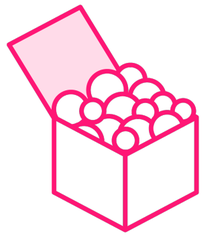
Enforncing Generic Type Behavior with Trait Bounds

```
pub trait Rate {
  fn calc_rating(&self) -> i32;
}

fn best<T: Rate>(items: Vec<T>) -> &T {
  let mut best = &items[0];

  for item in items {
    if item.calc_rating() > best.calc_rating {
      best = item;
    }
  }

  best
}
```

Object-oriented Programming in Rust

Key Aspects of an Object-oriented Language

Supports objects



Supports encapsulation

Supports inheritance

Rust Object-oriented Language Features

Supported	VS	Unsupported
Objects – structs/enums Encapsulation – pub/private Polymorphism – via traits		Inheritance – although traits/supertraits are a way around this

Rust Version of Objects – Data with Methods

```
struct Coffee {
  name: &str;
  cost: f64;
  size: &str;
}

impl Coffee {
  fn log(&self) -> () {
    println!("{} {} costing {}", self.size, self.name, self.cost);
  }
}

let my_coffee = Coffee { name: “Drip”, cost: 3.50, size: “Small” };

my_coffee.log();
```

Encapsulation

```
pub struct CoffeeInventory {
  coffees: Vec<Coffee>;
  pub total: f64;
}

impl CoffeeInventory {
  pub fn add_coffee(&mut self, coffee: &Coffee {
    self.coffees.push(coffee);
    self.update_total();
  }

  fn update_total(&mut self) {
    self.total = self.total + self.coffees[self.coffees.len() - 1].cost;
  }
}
```

Polymorphism

```
pub trait Mammal {
  fn describe_fur(&self);
}

impl Mammal for Monkey {
  fn describe_fur(&self) {
    println!(“A monkey’s fur commonly contains black, brown or grey tones.”);
  }
}

impl Mammal for Bear {
  fn describe_fur(&self) {
    println!(“A bear’s fur is commonly thick and shaggy.”);
  }
}

fn mammal_fur_is(mammal: Mammal) {
  mammal.describe_fur();
}
```

Pseudo-inheritance Via Supertraits

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

trait Mammal: Animal {
  fn describe_skin(&self) -> String;
}
```

You can use Rust in a functional way or in an object-oriented fashion – its up to you!

Demo

OOP using traits / generics

- Inheritance
- Encapsulation