RUSTikales Rust for beginners

Plan for today

Plan for today

1. Recap

Plan for today

1. Recap
2. Traits

Plan for today

1. Recap
2. Traits
    a. Declaration and Usage

Plan for today

1. Recap
2. Traits
    a. Declaration and Usage
    b. Debug, Display

Plan for today

1. Recap
2. Traits
    a. Declaration and Usage
    b. Debug, Display
    c. Move, Copy, Clone

1. Recap

1. Recap

- Ownership-Model
- Borrowing
- Borrow Checker

1. Recap

- Ownership-Model
- Borrowing
- Borrow Checker
- Functions
  - Declared using keyword fn
  - Take in a list of parameters/arguments
  - Can return values

1. Recap

- Ownership-Model
- Borrowing
- Borrow Checker
- Functions
- Structs
    - Declared using the keyword struct
    - Made out of fields
        - Field names must be unique
    - Count as type definition

1. Recap

- Ownership-Model
- Borrowing
- Borrow Checker
- Functions
- Structs
    - Declared using the keyword struct
    - Made out of fields
        - Field names must be unique
    - Count as type definition
        - Can be used as parameter and variable types
        - Can be used as field type
        - etc

1. Recap

- Ownership-Model
- Borrowing
- Borrow Checker
- Functions
- Structs
- Associated functions
  - Declared using the keyword impl
  - Declared like normal functions
  - Used by calling <struct>::<fn_name>()

1. Recap

- Ownership-Model
- Borrowing
- Borrow Checker
- Functions
- Structs
- Associated functions
- Methods
  - Associated functions where the first parameter is either self, &self or &mut self
  - Can be called on instances of structs using <instance>.<fn_name>()

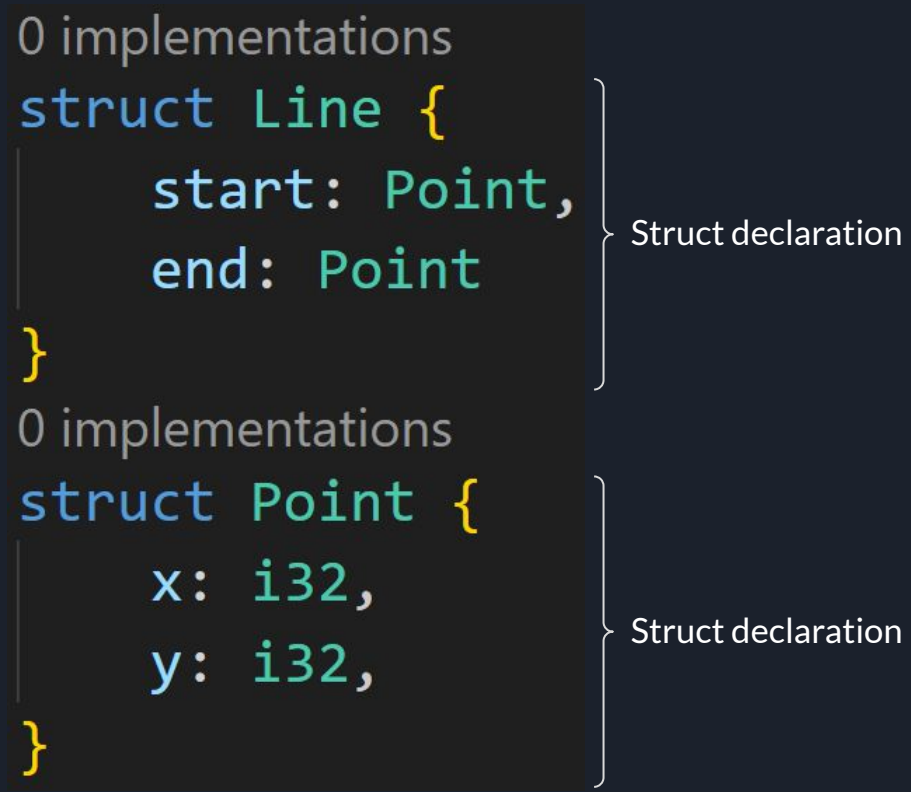
1. Recap

- Ownership-Model
- Borrowing
- Borrow Checker
- Functions
- Structs
- Associated functions
- Methods
- Order of struct and impl declarations does not matter

1. Recap

```
0 implementations
struct Line {
    start: Point,
    end: Point
}
0 implementations
struct Point {
    x: i32,
    y: i32,
}
```

1. Recap



1. Recap

```
0 implementations
struct Line {
    start: Point,
    end: Point
}
0 implementations
struct Point {
    x: i32,
    y: i32,
}
```

Struct defined here

1. Recap

```
0 implementations
struct Line {
    start: Point,
    end: Point
}
0 implementations
struct Point {
    x: i32,
    y: i32,
}
```

Can be used here

Struct defined here

1. Recap

```rust
impl Line {
    fn length(&self) -> f32 {
        let x: f32 = (self.end.x - self.start.x) as f32;
        let y: f32 = (self.end.y - self.start.y) as f32;
        f32::sqrt(self: x * x + y * y)
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 3, y: 4 };
    let p2: Point = Point { x: 5, y: 10 };
    let line: Line = Line { start: p1, end: p2 };
    println!("length = {}", line.length());
}
```

1. Recap

```
impl Line {
    fn length(&self) -> f32 {
        let x: f32 = (self.end.x - self.start.x) as f32;
        let y: f32 = (self.end.y - self.start.y) as f32;
        f32::sqrt(self: x * x + y * y)
    }
}
```

Method

```
▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 3, y: 4 };
    let p2: Point = Point { x: 5, y: 10 };
    let line: Line = Line { start: p1, end: p2 };
    println!("length = {}", line.length());
}
```

1. Recap

```
impl Line {
    fn length(&self) -> f32 {
        let x: f32 = (self.end.x - self.start.x) as f32;
        let y: f32 = (self.end.y - self.start.y) as f32;
        f32::sqrt(self: x * x + y * y)
    }
}
```

call to associated function

Method

```
▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 3, y: 4 };
    let p2: Point = Point { x: 5, y: 10 };
    let line: Line = Line { start: p1, end: p2 };
    println!("length = {}", line.length());
}
```

1. Recap

```rust
impl Line {
    fn length(&self) -> f32 {
        let x: f32 = (self.end.x - self.start.x) as f32;
        let y: f32 = (self.end.y - self.start.y) as f32;
        f32::sqrt(self: x * x + y * y)
    }
}
```

Method

call to associated function

▶ Run | Debug

```rust
fn main() {
    let p1: Point = Point { x: 3, y: 4 };
    let p2: Point = Point { x: 5, y: 10 };
    let line: Line = Line { start: p1, end: p2 };
    println!("length = {}", line.length());
}
```

call to method on instance line

1. Recap

```
impl Line {
    fn length(&self) -> f32 {
        let x: f32 = (self.end.x - self.start.x) as f32;
        let y: f32 = (self.end.y - self.start.y) as f32;
        f32::sqrt(self: x * x + y * y)
    }
}
```

call to associated function

Method

```
▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 3, y: 4 };
    let p2: Point = Point { x: 5, y: 10 };
    let line: Line = Line { start: p1, end: p2 };
    println!("length = {}", line.length());
}
```

Ownership of Points behind p1 and p2 goes to line here!

call to method on instance line

2. Traits

2. Traits

- Structs and methods are very powerful, but also pretty limited right now

2. Traits

- Structs and methods are very powerful, but also pretty limited right now
  - We can’t print them to the console
  - Ownership problems, we can only move our structs

2. Traits

```
0 implementations
struct Foo {}

fn no_copy(foo: Foo) {}
▶ Run | Debug
fn main() {
    let foo: Foo = Foo {};
    println!("{}", foo);
    println!("{:?}", foo);
    no_copy(foo);
    println!("{}", foo);
}
```

2. Traits

```
0 implementations
struct Foo {}

fn no_copy(foo: Foo) {}
▶ Run | Debug
fn main() {
    let foo: Foo = Foo {};
    println!("{}", foo);
    println!("{:?}", foo);
    no_copy(foo);
    println!("{}", foo);
}
```

Can’t print normally

2. Traits

```
0 implementations
struct Foo {}

fn no_copy(foo: Foo) {}
▶ Run | Debug
fn main() {
    let foo: Foo = Foo {};
    println!("{}", foo);
    println!("{:?}", foo);
    no_copy(foo);
    println!("{}", foo);
}
```

Can’t print normally

Can’t debug print

2. Traits

```
0 implementations
struct Foo {}

fn no_copy(foo: Foo) {}
▶ Run | Debug
fn main() {
    let foo: Foo = Foo {};
    println!("{}", foo);
    println!("{:?}", foo);
    no_copy(foo);
    println!("{}", foo);
}
```

Can’t print normally

Can’t debug print

foo is moved, can’t use it below

2. Traits

- Structs and methods are very powerful, but also pretty limited right now
  - We can’t print them to the console
  - Ownership problems, we can’t move our structs
- To deal with this, Rust has the trait system

2. Traits

- Structs and methods are very powerful, but also pretty limited right now
  - We can't print them to the console
  - Ownership problems, we can't move our structs
- To deal with this, Rust has the trait system
- Traits are like a contract:
  - A trait consists of function declarations
  - If you want to use a trait for a struct, you need to define the functions for your struct

2. Traits

- Structs and methods are very powerful, but also pretty limited right now
  - We can't print them to the console
  - Ownership problems, we can't move our structs
- To deal with this, Rust has the trait system
- Traits are like a contract:
  - A trait consists of function declarations
  - If you want to use a trait for a struct, you need to define the functions for your struct
- Traits can be defined using the keyword trait

2. Traits

- Structs and methods are very powerful, but also pretty limited right now
  - We can't print them to the console
  - Ownership problems, we can't move our structs
- To deal with this, Rust has the trait system
- Traits are like a contract:
  - A trait consists of function declarations
  - If you want to use a trait for a struct, you need to define the functions for your struct
- Traits can be defined using the keyword trait
- Implementing a trait for a struct is similar to associated functions:
  impl <trait_name> for <struct_name> { … }

2. Traits

```rust
trait Geometry {
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
```

1 implementation

```rust
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

2. Traits

```
trait Geometry {
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
1 implementation
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

Define trait Geometry with two functions: area and perimeter

2. Traits

```
trait Geometry {
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
1 implementation
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

Normally, traits do not define a function body

2. Traits

```
trait Geometry {
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
1 implementation
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

Normally, traits do not define a function body

All we later care about is that those functions exist, we don't care about what they're doing

2. Traits

```
trait Geometry {
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
1 implementation
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

Normally, traits do not define a function body

All we later care about is that those functions exist, we don't care about what they're doing

In most cases, we couldn't even give a base implementation, like here: What's the base area for every possible object?

2. Traits

```rust
trait Geometry {
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
1 implementation
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

Implement trait for our struct

2. Traits

```rust
trait Geometry {
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
1 implementation
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

We need to implement every function of a trait, and we need to respect the signature

Implement trait for our struct

2. Traits

```
trait Geometry {
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
1 implementation
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

We need to implement every function of a trait, and we need to respect the signature:

- Parameters must be identical
  - Same count
  - Same type
- Return type must match
- Name must match

2. Traits

```rust
trait Geometry {
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
1 implementation
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

We need to implement every function of a trait, and we need to respect the signature:

- Parameters must be identical
  - Same count
  - Same type
- Return type must match
- Name must match

Here it means:
Implement two methods which return the area and perimeter of that instance

2. Traits

```
trait Geometry {
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
1 implementation
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

We need to implement every function of a trait, and we need to respect the signature:

- Parameters must be identical
  - Same count
  - Same type
- Return type must match
- Name must match

Here it means:
Implement two methods which return the area and perimeter of that instance

We do that here! Everything's fine

2. Traits

- Implementing a trait allows us to use the defined functions as we would use associated functions, or methods

```
▶ Run | Debug
fn main() {
    let rect: Rectangle = Rectangle { width: 10.0, height: 20.0 };
    println!("Area: {}", rect.area());
    println!("Perimeter: {}", rect.perimeter());
}
```

2. Traits

- Implementing a trait allows us to use the defined functions as we would use associated functions, or methods
- But isn’t that kinda redundant, what do we gain from doing that?

3/3

2. Traits

- Implementing a trait allows us to use the defined functions as we would use associated functions, or methods
- But isn't that kinda redundant, what do we gain from doing that?
- We can generalize our code to take in *any struct* that implements a given trait

2. Traits

```rust
fn calculate_geometry(obj: &impl Geometry) {
    println!("Area: {}", obj.area());
    println!("Perimeter: {}", obj.perimeter());
}
▶ Run | Debug
fn main() {
    let rect: Rectangle = Rectangle { width: 10.0, height: 20.0 };
    calculate_geometry(obj: &rect);
    let vector: Vec<i32> = vec![1, 2, 3];
    calculate_geometry(obj: &vector);
}
```

2. Traits

We do not care about the type of the parameter, as long as it implements the Geometry trait it's fine! :)

```rust
fn calculate_geometry(obj: &impl Geometry) {
    println!("Area: {}", obj.area());
    println!("Perimeter: {}", obj.perimeter());
}
▶ Run | Debug
fn main() {
    let rect: Rectangle = Rectangle { width: 10.0, height: 20.0 };
    calculate_geometry(obj: &rect);
    let vector: Vec<i32> = vec![1, 2, 3];
    calculate_geometry(obj: &vector);
}
```

2. Traits

We do not care about the type of the parameter, as long as it implements the Geometry trait it's fine! :)

```rust
fn calculate_geometry(obj: &impl Geometry) {
    println!("Area: {}", obj.area());
    println!("Perimeter: {}", obj.perimeter());
}
▶ Run | Debug
fn main() {
    let rect: Rectangle = Rectangle { width: 10.0, height: 20.0 };
    calculate_geometry(obj: &rect);
    let vector: Vec<i32> = vec![1, 2, 3];
    calculate_geometry(obj: &vector);
}
```

Rectangle implements Geometry

2. Traits

We do not care about the type of the parameter, as long as it implements the Geometry trait it's fine! :)

```rust
fn calculate_geometry(obj: &impl Geometry) {
    println!("Area: {}", obj.area());
    println!("Perimeter: {}", obj.perimeter());
}
▶ Run | Debug
fn main() {
    let rect: Rectangle = Rectangle { width: 10.0, height: 20.0 };
    calculate_geometry(obj: &rect);
    let vector: Vec<i32> = vec![1, 2, 3];
    calculate_geometry(obj: &vector);
}
```

Rectangle implements Geometry

Vec<T> does not :^)

2. Traits

- With this mechanism, we can take a look at the compiler error for println!() earlier

2. Traits

- With this mechanism, we can take a look at the compiler error for println!() earlier

```
error[E0277]: `Foo` doesn't implement `std::fmt::Display`
 --> src\main.rs:7:20
  |
7 |     println!("{}", foo);
  |                    ^^^ `Foo` cannot be formatted with the default formatter
  |
  = help: the trait `std::fmt::Display` is not implemented for `Foo`
```

2. Traits

- With this mechanism, we can take a look at the compiler error for println!() earlier

```
error[E0277]: `Foo` doesn't implement `std::fmt::Display`
 --> src\main.rs:7:20
  |
7 |     println!("{}", foo);
  |                    ^^^ `Foo` cannot be formatted with the default formatter
  |
  = help: the trait `std::fmt::Display` is not implemented for `Foo`
```

Aha!

2. Traits

- With this mechanism, we can take a look at the compiler error for println!() earlier
- We can just follow the compiler errors until it works :)

2. Traits

- With this mechanism, we can take a look at the compiler error for println!() earlier
- We can just follow the compiler errors until it works :)
- It wanted std::fmt::Display for Foo, so let's do that:

```
impl std::fmt::Display for Foo {
}
```

2. Traits

- With this mechanism, we can take a look at the compiler error for println!() earlier
- We can just follow the compiler errors until it works :)
- It wanted std::fmt::Display for Foo, so let's do that:

```
impl std::fmt::Display for Foo {
}
```

Hm, missing some function definitions!

```
error[E0046]: not all trait items implemented, missing: `fmt`
 --> src\main.rs:3:1
  |
3 | impl std::fmt::Display for Foo {
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ missing `fmt` in implementation
  |
  = help: implement the missing item: `fn fmt(&self, _: &mut Formatter<'_>) -> Result<(), std::fmt::Error> { todo!() }`
```

2. Traits

- With this mechanism, we can take a look at the compiler error for println!() earlier
- We can just follow the compiler errors until it works :)
- It wanted std::fmt::Display for Foo, so let's do that:

```
struct Foo {}
impl std::fmt::Display for Foo {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "We have implemented the Display trait for Foo!")
    }
}
```

2. Traits

- With this mechanism, we can take a look at the compiler error for println!() earlier
- We can just follow the compiler errors until it works :)
- It wanted std::fmt::Display for Foo, so let's do that:

Scary, but you can ignore that most of the time :)

```
struct Foo {}
impl std::fmt::Display for Foo {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "We have implemented the Display trait for Foo!")
    }
}
```

2. Traits

- With this mechanism, we can take a look at the compiler error for println!() earlier
- We can just follow the compiler errors until it works :)
- It wanted std::fmt::Display for Foo, so let's do that:

```
struct Foo {}
impl std::fmt::Display for Foo {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "We have implemented the Display trait for Foo!")
    }
}
```

The `fmt` function it wanted!

2. Traits

- With this mechanism, we can take a look at the compiler error for println!() earlier
- We can just follow the compiler errors until it works :)
- It wanted std::fmt::Display for Foo, so let's do that:

```
struct Foo {}
impl std::fmt::Display for Foo {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "We have implemented the Display trait for Foo!")
    }
}
```

We have now implemented the Display trait, and can use println!("{}") for Foo :)

2. Traits

```rust
struct Foo {}
impl std::fmt::Display for Foo {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "We have implemented the Display trait for Foo!")
    }
}
▶ Run | Debug
fn main() {
    let foo: Foo = Foo {};
    println!("{}", foo);
```

```
Command Prompt

C:\Users\pfhau\GithubProjects\progkurs\rust-beginner\09 - Traits\no_trait>cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target\debug\no_trait.exe`
We have implemented the Display trait for Foo!
```

2. Traits

```rust
struct Foo {}
impl std::fmt::Display for Foo {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "We have implemented the Display trait for Foo!")
    }
}
▶ Run | Debug
fn main() {
    let foo: Foo = Foo {};
    println!("{}", foo);
}
```

The "{}" formatter calls to Display::fmt() in the background, which is why we need to implement the trait

```
Command Prompt

C:\Users\pfhau\GithubProjects\progkurs\rust-beginner\09 - Traits\no_trait>cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target\debug\no_trait.exe`
We have implemented the Display trait for Foo!
```

Intermission - Strings

- write!, format!, print! - A lot of macros make use of Strings, but what is that?

Intermission - Strings

- write!, format!, print! - A lot of macros make use of Strings, but what is that?
- We've come a long way without ever using Strings, it is now time to introduce them

Intermission - Strings

- write!, format!, print! - A lot of macros make use of Strings, but what is that?
- We've come a long way without ever using Strings, it is now time to introduce them
- Rust has two String types: &str and String

Intermission - Strings

- write!, format!, print! - A lot of macros make use of Strings, but what is that?
- We've come a long way without ever using Strings, it is now time to introduce them
- Rust has two String types: &str and String
- Both essentially do the same thing: They are collections of characters

Intermission - Strings

- write!, format!, print! - A lot of macros make use of Strings, but what is that?
- We've come a long way without ever using Strings, it is now time to introduce them
- Rust has two String types: &str and String
- Both essentially do the same thing: They are collections of characters
- You can think of them as Arrays and Vectors:
  - &str is basically an Array: On the data section of your executable, not resizable
  - String is basically a Vector: On the heap, resizable

Intermission - Strings

- write!, format!, print! - A lot of macros make use of Strings, but what is that?
- We've come a long way without ever using Strings, it is now time to introduce them
- Rust has two String types: &str and String
- Both essentially do the same thing: They are collections of characters
- You can think of them as Arrays and Vectors:
  - &str is basically an Array: On the data section of your executable, not resizable
  - String is basically a Vector: On the heap, resizable

```
#[derive(PartialEq, PartialOrd, Eq, Ord)]
#[stable(feature = "rust1", since = "1.0.0")]
#[cfg_attr(not(test), lang = "String")]
63 implementations
pub struct String {
    vec: Vec<u8>,   ⟵ String is literally a Vector! :^)
}
```

Intermission - Strings

- write!, format!, print! - A lot of macros make use of Strings, but what is that?
- We've come a long way without ever using Strings, it is now time to introduce them
- Rust has two String types: &str and String
- Both essentially do the same thing: They are collections of characters
- You can think of them as Arrays and Vectors:
  - &str is basically an Array: On the data section of your executable, not resizable
  - String is basically a Vector: On the heap, resizable
- Strings in Rust are UTF-8 encoded, which means you can have arabic, cyrillic or japanese characters in your String literals, and even emojis

Intermission - Strings

```rust
fn main() {
    let str: &str = "Hello, world!";
    let string: String = String::from("Hello, world!");
    let emote_char: char = '👋';
    let emote: &str = "👋";
    let emote_string: String = String::from("👋");
    let hello_in_greek: String = String::from("Γεια σου κόσμε!");
    let hello_in_japanese: String = String::from("こんにちは世界!");
    println!("str: {}", str);
    println!("string: {}", string);
    println!("emote_char: {}", emote_char);
    println!("emote: {}", emote);
    println!("emote_string: {}", emote_string);
    println!("hello_in_greek: {}", hello_in_greek);
    println!("hello_in_japanese: {}", hello_in_japanese);
}
```

Intermission - Strings

```rust
fn main() {
    let str: &str = "Hello, world!";
    let string: String = String::from("Hello, world!");
    let emote_char: char = '👋';
    let emote: &str = "👋";
    let emote_string: String = String::from("👋");
    let hello_in_greek: String = String::from("Γεια σου κόσμε!");
    let hello_in_japanese: String = String::from("こんにちは世界!");
    println!("str: {}", str);
    println!("string: {}", string);
    println!("emote_char: {}", emote_char);
    println!("emote: {}", emote);
    println!("emote_string: {}", emote_string);
    println!("hello_in_greek: {}", hello_in_greek);
    println!("hello_in_japanese: {}", hello_in_japanese);
}
```

String literals are always of type &str

Intermission - Strings

```rust
fn main() {
    let str: &str = "Hello, world!";
    let string: String = String::from("Hello, world!");
    let emote_char: char = '👋';
    let emote: &str = "👋";
    let emote_string: String = String::from("👋");
    let hello_in_greek: String = String::from("Γεια σου κόσμε!");
    let hello_in_japanese: String = String::from("こんにちは世界!");
    println!("str: {}", str);
    println!("string: {}", string);
    println!("emote_char: {}", emote_char);
    println!("emote: {}", emote);
    println!("emote_string: {}", emote_string);
    println!("hello_in_greek: {}", hello_in_greek);
    println!("hello_in_japanese: {}", hello_in_japanese);
}
```

associated function takes a &str and converts it to a String

Intermission - Strings

```rust
fn main() {
    let str: &str = "Hello, world!";
    let string: String = String::from("Hello, world!");
    let emote_char: char = '👋';
    let emote: &str = "👋";
    let emote_string: String = String::from("👋");
    let hello_in_greek: String = String::from("Γεια σου κόσμε!");
    let hello_in_japanese: String = String::from("こんにちは世界!");
    println!("str: {}", str);
    println!("string: {}", string);
    println!("emote_char: {}", emote_char);
    println!("emote: {}", emote);
    println!("emote_string: {}", emote_string);
    println!("hello_in_greek: {}", hello_in_greek);
    println!("hello_in_japanese: {}", hello_in_japanese);
}
```

Editor warning, char could be confused

Intermission - Strings

```rust
fn main() {
    let str: &str = "Hello, world!";
    let string: String = String::from("Hello, world!");
    let emote_char: char = '👋';
    let emote: &str = "👋";
    let emote_string: String = String::from("👋");
    println!("str: {}", str);
    println!("string: {}", string);
    println!("emote_char: {}", emote_char);
    println!("emote: {}", emote);
    println!("emote_string: {}", emote_string);
    println!("hello_in_greek: {}", hello_in_greek);
    println!("hello_in_japanese: {}", hello_in_japanese);
}
```

The character `U+03bf` " `o` " could be confused with the ASCII character `U+006f` " `o` ", which is more common in source code. Adjust settings

Intermission - Strings

```rust
fn main() {
    let str: &str = "Hello, world!";
    let string: String = String::from("Hello, world!");
    let emote_char: char = '👋';
    let emote: &str = "👋";
    let emote_string: String = String::from("👋");
    let hello_in_greek: String = String::from("Γεια σου κόσμε!");
    let hello_in_japanese: String = String::from("こんにちは世界!");
    println!("str: {}", str);
    println!("string: {}", string);
    println!("emote_char: {}", emote_char);
    println!("emote: {}", emote);
    println!("emote_string: {}", emote_string);
    println!("hello_in_greek: {}", hello_in_greek);
    println!("hello_in_japanese: {}", hello_in_japanese);
}
```

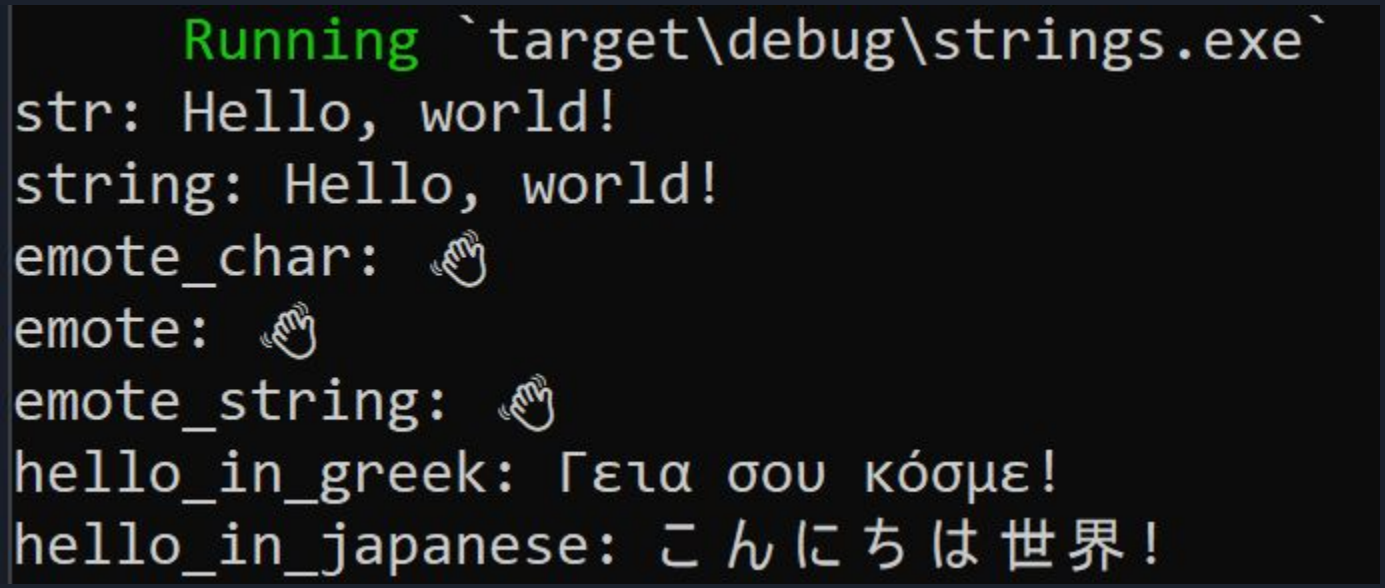
Everything can be encoded in UTF-8 :)

Intermission - Strings

```rust
fn main() {
    let str: &str = "Hello, world!";
    let string: String = String::from("Hello, world!");
    let emote_char: char = '👋';
    let emote: &str = "👋";
    let emote_string: String = String::from("👋");
    let hello_in_greek: String = String::from("Γεια σου κόσμε!");
    let hello_in_japanese: String = String::from("こんにちは世界!");
    println!("str: {}", str);
    println!("string: {}", string);
    println!("emote_char: {}", emote_char);
    println!("emote: {}", emote);
    println!("emote_string: {}", emote_string);
    println!("hello_in_greek: {}", hello_in_greek);
    println!("hello_in_japanese: {}", hello_in_japanese);
}
```

Also UTF-8 :)

Intermission - Strings



Intermission - format!()

- format!() is an important macro to turn any value into a String, for example to print it

Intermission - format!()

- format!() is an important macro to turn any value into a String, for example to print it
- The first argument is the format string, in it you build your string by providing placeholders for the values you want to format

Intermission - format!()

- format!() is an important macro to turn any value into a String, for example to print it
- The first argument is the format string, in it you build your string by providing placeholders for the values you want to format
- Commonly used placeholders in Rust:
  - {} → std::fmt::Display
  - {:?} → std::fmt::Debug
  - {:#?} → Debug, but pretty printed

Intermission - format!()

- format!() is an important macro to turn any value into a String, for example to print it
- The first argument is the format string, in it you build your string by providing placeholders for the values you want to format
- Commonly used placeholders in Rust:
  - {} → std::fmt::Display
  - {:?} → std::fmt::Debug
  - {:#?} → Debug, but pretty printed
- Other placeholders exist, but are (usually) only used for numbers:
  - {:o} → std::fmt::Octal
  - {:x} → std::fmt::LowerHex
  - {:b} → std::fmt::Binary

Intermission - format!()

- format!() is an important macro to turn any value into a String, for example to print it
- The first argument is the format string, in it you build your string by providing placeholders for the values you want to format
- For each placeholder you specify, you need to pass an additional argument to format!():
  - The value that goes into that place

Intermission - format!()

- format!() is an important macro to turn any value into a String, for example to print it
- The first argument is the format string, in it you build your string by providing placeholders for the values you want to format
- For each placeholder you specify, you need to pass an additional argument to format!():
  - The value that goes into that place
- Rust then does the correct calls to the correct trait in the background, failing when it can't find an implementation

Intermission - format!()

- format!() is an important macro to turn any value into a String, for example to print it
- The first argument is the format string, in it you build your string by providing placeholders for the values you want to format
- For each placeholder you specify, you need to pass an additional argument to format!():
  - The value that goes into that place
- Rust then does the correct calls to the correct trait in the background, failing when it can't find an implementation
- format!() allows named arguments

Intermission - format!()

```rust
#[derive(Debug)]
2 implementations
struct Person {
    name: String,
    age: u8,
    height: u8,
}
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    println!("{}", person);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

Intermission - format!()

```rust
#[derive(Debug)]
2 implementations
struct Person {
    name: String,
    age: u8,
    height: u8,
}
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    println!("{}", person);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

Usage of println! identical to format!: First the format string, then the values

Intermission - format!()

```rust
#[derive(Debug)]
2 implementations
struct Person {
    name: String,
    age: u8,
    height: u8,
}
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    println!("{}", person);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

Debug is a derivable macro, and usually always implemented that way

Both call to that Debug implementation

Intermission - format!()

```rust
#[derive(Debug)]
2 implementations
struct Person {
    name: String,
    age: u8,
    height: u8,
}
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    println!("{}", person);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

Debug output:

```
Person { name: "John", age: 32, height: 180 }
Person {
    name: "John",
    age: 32,
    height: 180,
}
```

Intermission - format!()

```rust
#[derive(Debug)]
2 implementations
struct Person {
    name: String,
    age: u8,
    height: u8,
}
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    println!("{}", person);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

Debug is derivable, because it follows a simple formula:

For every field in your struct:
→ Collect debug format output of that field
→ field_name: field_value
Print that

```
Person { name: "John", age: 32, height: 180 }
Person {
    name: "John",
    age: 32,
    height: 180,
}
```

Intermission - format!()

```rust
#[derive(Debug)]
2 implementations
struct Person {
    name: String,
    age: u8,
    height: u8,
}
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    println!("{}", person);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

format field name

```
Person { name: "John", age: 32, height: 180 }
Person {
    name: "John",
    age: 32,
    height: 180,
}
```

Intermission - format!()

```rust
#[derive(Debug)]
2 implementations
struct Person {
    name: String,
    age: u8,
    height: u8,
}
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    println!("{}", person);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

format field age

```
Person { name: "John", age: 32, height: 180 }
Person {
    name: "John",
    age: 32,
    height: 180,
}
```

Intermission - format!()

```rust
#[derive(Debug)]
2 implementations
struct Person {
    name: String,
    age: u8,
    height: u8,
}
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    println!("{}", person);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

format field height

```
Person { name: "John", age: 32, height: 180 }
Person {
    name: "John",
    age: 32,
    height: 180,
}
```

Intermission - format!()

```rust
#[derive(Debug)]
2 implementations
struct Person {
    name: String,
    age: u8,
    height: u8,
}
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    println!("{}", person);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

```
Person { name: "John", age: 32, height: 180 }
Person {
    name: "John",
    age: 32,
    height: 180,
}
```

{:#?} just adds newlines :)

Intermission - format!()

```rust
#[derive(Debug)]
2 implementations
struct Person {
    name: String,
    age: u8,
    height: u8,
}
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    println!("{}", person);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

This calls to the implementation of Display, which is given above

Intermission - format!()

```rust
#[derive(Debug)]
2 implementations
struct Person {
    name: String,
    age: u8,
    height: u8,
}
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    println!("{}", person);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

Essentially also uses format!() internally
Difference: write!() requires a formatter as first argument, which is kindly given as a parameter and just passed along :)

This calls to the implementation of Display, which is given above

Intermission - format!()

```rust
#[derive(Debug)]
2 implementations
struct Person {
    name: String,
    age: u8,
    height: u8,
}
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    println!("{}", person);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

Note that write!() returns a value, which we also return in `fmt`!
It returns the result of writing, which might fail. More on Results later :^)

no semicolon!

This calls to the implementation of Display, which is given above

Intermission - format!()

```rust
#[derive(Debug)]
2 implementations
struct Person {
    name: String,
    age: u8,
    height: u8,
}
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    println!("{}", person);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

Output:

```
Name: John, Age: 32, Height: 180
```

Intermission - format!()

```rust
#[derive(Debug)]
2 implementations
struct Person {
    name: String,
    age: u8,
    height: u8,
}
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    println!("{}", person);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

- Named arguments must be given last
- For named Debug etc, use {p:?}

Intermission - format!()

```rust
#[derive(Debug)]
2 implementations
struct Person {
    name: String,
    age: u8,
    height: u8,
}
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    println!("{}", person);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

```
    Running `target\debug\format.exe`
Name: John, Age: 32, Height: 180
Person { name: "John", age: 32, height: 180 }
Person {
    name: "John",
    age: 32,
    height: 180,
}
Name: John, Age: 32, Height: 180
```

2. Traits

- Many things in Rust are only syntactic sugar for calls to traits
  - format!() and println!() call to Display and Debug, for example

2. Traits

- Many things in Rust are only syntactic sugar for calls to traits
  - format!() and println!() call to Display and Debug, for example
- Additionally, the following things are traits:
  - Arithmetic operations such as Addition and Subtraction
  - Comparisons such as Equal or LessThan
  - for-loops require the Iterator-Trait

2. Traits

- Many things in Rust are only syntactic sugar for calls to traits
  - format!() and println!() call to Display and Debug, for example
- Additionally, the following things are traits:
  - Arithmetic operations such as Addition and Subtraction
  - Comparisons such as Equal or LessThan
  - for-loops require the Iterator-Trait
- This means that we could theoretically tell Rust how to add two persons together

2. Traits

```rust
impl std::ops::Add for Person {
    type Output = Person;
    fn add(self, other: Person) -> Person {
        Person {
            name: format!("{}-{}", self.name, other.name),
            age: self.age + other.age,
            height: self.height + other.height,
        }
    }
}
▶ Run | Debug
fn main() {
    let person1: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let person2: Person = Person {
        name: String::from("Jane"),
        age: 28,
        height: 160,
    };
    let person3: Person = person1 + person2;
    println!("{}", person3);
}
```

2. Traits

```rust
impl std::ops::Add for Person {
    type Output = Person;
    fn add(self, other: Person) -> Person {
        Person {
            name: format!("{}-{}", self.name, other.name),
            age: self.age + other.age,
            height: self.height + other.height,
        }
    }
}
▶ Run | Debug
fn main() {
    let person1: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let person2: Person = Person {
        name: String::from("Jane"),
        age: 28,
        height: 160,
    };
    let person3: Person = person1 + person2;
    println!("{}", person3);
}
```

This is a call to Add::add(), which we implemented above :)
Note that this consumes both persons

2. Traits

- Many things in Rust are only syntactic sugar for calls to traits
  - format!() and println!() call to Display and Debug, for example
- Additionally, the following things are traits:
  - Arithmetic operations such as Addition and Subtraction
  - Comparisons such as Equal or LessThan
  - for-loops require the Iterator-Trait
- This means that we could theoretically tell Rust how to add two persons together
- But more importantly, Ownership is also handled with traits :)

2. Traits

- Many things in Rust are only syntactic sugar for calls to traits
  - format!() and println!() call to Display and Debug, for example
- Additionally, the following things are traits:
  - Arithmetic operations such as Addition and Subtraction
  - Comparisons such as Equal or LessThan
  - for-loops require the Iterator-Trait
- This means that we could theoretically tell Rust how to add two persons together
- But more importantly, Ownership is also handled with traits :)
  - Structs in Rust are always moved, unless the Copy-trait is implemented
  - Copy is a derived trait, and only works if all fields can be copied
    - For our Person that doesn't work, because String can't be copied :^)

2. Traits

```
#[derive(Clone, Copy)]
2 implementations
struct Point {
    x: i32,
    y: i32,
}

▶ Run | Debug
fn main() {
    let p: Point = Point { x: 1, y: 2 };
    let p1: Point = p;
    println!("p.x = {}", p.x);
    println!("p1.x = {}", p1.x);
    println!("Hello, world!");
}
```

2. Traits

```rust
#[derive(Clone, Copy)]
2 implementations
struct Point {
    x: i32,
    y: i32,
}

▶ Run | Debug
fn main() {
    let p: Point = Point { x: 1, y: 2 };
    let p1: Point = p;
    println!("p.x = {}", p.x);
    println!("p1.x = {}", p1.x);
    println!("Hello, world!");
}
```

Clone is supertrait of Copy, so we need to derive this too

2. Traits

```rust
#[derive(Clone, Copy)]
2 implementations
struct Point {
    x: i32,
    y: i32,
}

▶ Run | Debug
fn main() {
    let p: Point = Point { x: 1, y: 2 };
    let p1: Point = p;
    println!("p.x = {}", p.x);
    println!("p1.x = {}", p1.x);
    println!("Hello, world!");
}
```

Because we derived Copy, Rust no longer moves `p`! :)

Intermission - Exercise

- Time for exercises!

```rust
trait Stats {
    fn get_weight(&self) -> f32;
    fn get_length(&self) -> f32;
}
1 implementation
struct Fish {
    kind: String,
    weight: f32,
    length: f32,
}
impl Stats for Fish {
    fn get_weight(&self) -> f32 {
        self.weight
    }
    fn get_length(&self) -> f32 {
        self.length
    }
}
pub fn main() {
    let fish: Fish = Fish {
        kind: String::from("Salmon"),
        weight: 10.0,
        length: 20.0,
    };
    println!("The {} is {}cm long and weighs {}kg",
             fish.kind, fish.get_length(), fish.get_weight());
}
```

```rust
trait Stats {
    fn get_weight(&self) -> f32;
    fn get_length(&self) -> f32;
}
1 implementation
struct Fish {
    kind: String,
    weight: f32,
    length: f32,
}
impl Stats for Fish {
    fn get_weight(&self) -> f32 {
        self.weight
    }
    fn get_length(&self) -> f32 {
        self.length
    }
}
pub fn main() {
    let fish: Fish = Fish {
        kind: String::from("Salmon"),
        weight: 10.0,
        length: 20.0,
    };
    println!("The {} is {}cm long and weighs {}kg",
             fish.kind, fish.get_length(), fish.get_weight());
}
```

Does this example compile?
If yes, what does it print?

```rust
trait Stats {
    fn get_weight(&self) -> f32;
    fn get_length(&self) -> f32;
}
1 implementation
struct Fish {
    kind: String,
    weight: f32,
    length: f32,
}
impl Stats for Fish {
    fn get_weight(&self) -> f32 {
        self.weight
    }
    fn get_length(&self) -> f32 {
        self.length
    }
}
pub fn main() {
    let fish: Fish = Fish {
        kind: String::from("Salmon"),
        weight: 10.0,
        length: 20.0,
    };
    println!("The {} is {}cm long and weighs {}kg",
             fish.kind, fish.get_length(), fish.get_weight());
}
```

Does this example compile?
If yes, what does it print?

Yes, it does compile!

```rust
trait Stats {
    fn get_weight(&self) -> f32;
    fn get_length(&self) -> f32;
}
1 implementation
struct Fish {
    kind: String,
    weight: f32,
    length: f32,
}
impl Stats for Fish {
    fn get_weight(&self) -> f32 {
        self.weight
    }
    fn get_length(&self) -> f32 {
        self.length
    }
}
pub fn main() {
    let fish: Fish = Fish {
        kind: String::from("Salmon"),
        weight: 10.0,
        length: 20.0,
    };
    println!("The {} is {}cm long and weighs {}kg",
             fish.kind, fish.get_length(), fish.get_weight());
}
```

Trait declaration

Trait Implementation

Does this example compile?
If yes, what does it print?

Yes, it does compile!

```rust
trait Stats {
    fn get_weight(&self) -> f32;
    fn get_length(&self) -> f32;
}
1 implementation
struct Fish {
    kind: String,
    weight: f32,
    length: f32,
}
impl Stats for Fish {
    fn get_weight(&self) -> f32 {
        self.weight
    }
    fn get_length(&self) -> f32 {
        self.length
    }
}
pub fn main() {
    let fish: Fish = Fish {
        kind: String::from("Salmon"),
        weight: 10.0,
        length: 20.0,
    };
    println!("The {} is {}cm long and weighs {}kg",
             fish.kind, fish.get_length(), fish.get_weight());
}
```

Trait declaration

Trait Implementation

Does this example compile?
If yes, what does it print?

Yes, it does compile!

Output:

```
     Running `target\debug\exercises.exe`
The Salmon is 20cm long and weighs 10kg
```

```rust
use std::fmt::{Debug, Display};
1 implementation
trait Printable: Debug + Display {
    fn print_normal(&self) {
        println!("{}", self);
    }
    fn print_debug(&self) {
        println!("{:?}", self);
    }
}
1 implementation
struct Point {
    x: i32,
    y: i32,
}
impl Printable for Point {}

pub fn main() {
    let p: Point = Point { x: 10, y: 20 };
    p.print_normal();
    p.print_debug();
}
```

```rust
use std::fmt::{Debug, Display};
1 implementation
trait Printable: Debug + Display {
    fn print_normal(&self) {
        println!("{}", self);
    }
    fn print_debug(&self) {
        println!("{:?}", self);
    }
}
1 implementation
struct Point {
    x: i32,
    y: i32,
}
impl Printable for Point {}

pub fn main() {
    let p: Point = Point { x: 10, y: 20 };
    p.print_normal();
    p.print_debug();
}
```

Does this example compile?
If yes, what does it print?

```rust
use std::fmt::{Debug, Display};
1 implementation
trait Printable: Debug + Display {
    fn print_normal(&self) {
        println!("{}", self);
    }
    fn print_debug(&self) {
        println!("{:?}", self);
    }
}
1 implementation
struct Point {
    x: i32,
    y: i32,
}
impl Printable for Point {}

pub fn main() {
    let p: Point = Point { x: 10, y: 20 };
    p.print_normal();
    p.print_debug();
}
```

Does this example compile?
If yes, what does it print?

Supertraits:

Anything that implements Printable must also implement Debug and Display

```rust
use std::fmt::{Debug, Display};
1 implementation
trait Printable: Debug + Display {
    fn print_normal(&self) {
        println!("{}", self);
    }
    fn print_debug(&self) {
        println!("{:?}", self);
    }
}
1 implementation
struct Point {
    x: i32,
    y: i32,
}
impl Printable for Point {}

pub fn main() {
    let p: Point = Point { x: 10, y: 20 };
    p.print_normal();
    p.print_debug();
}
```

Does this example compile?
If yes, what does it print?

Supertraits:

Anything that implements Printable must also implement Debug and Display

This is okay, we actually provide function definitions in the trait (which is allowed in Rust)

```rust
use std::fmt::{Debug, Display};
1 implementation
trait Printable: Debug + Display {
    fn print_normal(&self) {
        println!("{}", self);
    }
    fn print_debug(&self) {
        println!("{:?}", self);
    }
}
1 implementation
struct Point {
    x: i32,
    y: i32,
}
impl Printable for Point {}

pub fn main() {
    let p: Point = Point { x: 10, y: 20 };
    p.print_normal();
    p.print_debug();
}
```

Does this example compile?
If yes, what does it print?

Supertraits:

Anything that implements Printable must also implement Debug and Display

Point does not implement Debug and Display :(

This is okay, we actually provide function definitions in the trait (which is allowed in Rust)

```rust
use std::fmt::{Debug, Display};
1 implementation
trait Printable: Debug + Display {
    fn print_normal(&self) {
        println!("{}", self);
    }
    fn print_debug(&self) {
        println!("{:?}", self);
    }
}
1 implementation
struct Point {
    x: i32,
    y: i32,
}
impl Printable for Point {}

pub fn main() {
    let p: Point = Point { x: 10, y: 20 };
    p.print_normal();
    p.print_debug();
}
```

Does this example compile?
If yes, what does it print?

Supertraits:

Anything that implements Printable must also implement Debug and Display

Point does not implement Debug and Display :(

→ Does not compile!

This is okay, we actually provide function definitions in the trait (which is allowed in Rust)

```rust
trait Animal {
    fn get_name(&self) -> &String;
    fn make_sound(&self);
}
1 implementation
struct Cat { name: String }
1 implementation
struct Dog { name: String }
impl Animal for Cat {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Meow!"); }
}
impl Animal for Dog {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Woof!"); }
}
pub fn main() {
    let cat: Cat = Cat { name: String::from("Misty") };
    cat.make_sound();
    let dog: Dog = Dog { name: String::from("Rusty") };
    dog.make_sound();
}
```

```rust
trait Animal {
    fn get_name(&self) -> &String;
    fn make_sound(&self);
}
1 implementation
struct Cat { name: String }
1 implementation
struct Dog { name: String }
impl Animal for Cat {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Meow!"); }
}
impl Animal for Dog {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Woof!"); }
}
pub fn main() {
    let cat: Cat = Cat { name: String::from("Misty") };
    cat.make_sound();
    let dog: Dog = Dog { name: String::from("Rusty") };
    dog.make_sound();
}
```

Does this example compile?
If yes, what does it print?

```rust
trait Animal {
    fn get_name(&self) -> &String;
    fn make_sound(&self);
}
```

Trait declaration

```rust
1 implementation
struct Cat { name: String }
1 implementation
struct Dog { name: String }
impl Animal for Cat {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Meow!"); }
}
impl Animal for Dog {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Woof!"); }
}
pub fn main() {
    let cat: Cat = Cat { name: String::from("Misty") };
    cat.make_sound();
    let dog: Dog = Dog { name: String::from("Rusty") };
    dog.make_sound();
}
```

Trait implementations

Does this example compile?
If yes, what does it print?

```rust
trait Animal {
    fn get_name(&self) -> &String;
    fn make_sound(&self);
}
1 implementation
struct Cat { name: String }
1 implementation
struct Dog { name: String }
impl Animal for Cat {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Meow!"); }
}
impl Animal for Dog {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Woof!"); }
}
pub fn main() {
    let cat: Cat = Cat { name: String::from("Misty") };
    cat.make_sound();
    let dog: Dog = Dog { name: String::from("Rusty") };
    dog.make_sound();
}
```

Does this example compile?
If yes, what does it print?

Each struct defines all necessary functions, there are no supertraits or other things

```rust
trait Animal {
    fn get_name(&self) -> &String;
    fn make_sound(&self);
}
1 implementation
struct Cat { name: String }
1 implementation
struct Dog { name: String }
impl Animal for Cat {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Meow!"); }
}
impl Animal for Dog {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Woof!"); }
}
pub fn main() {
    let cat: Cat = Cat { name: String::from("Misty") };
    cat.make_sound();
    let dog: Dog = Dog { name: String::from("Rusty") };
    dog.make_sound();
}
```

Does this example compile?
If yes, what does it print?

Each struct defines all necessary functions, there are no supertraits or other things

This code compiles :)

1/3

```rust
trait Forgettable {
    fn forget(&self);
}
1 implementation
struct Thing { name: String }
impl Forgettable for Thing {
    fn forget(&self) {
        println!("I'm forgetting {}", self.name);
    }
}
pub fn main() {
    let thing: Thing = Thing { name: String::from("my thing") };
    thing.forget();
    let thing2: Thing = Thing { name: String::from("my other thing") };
    thing2.forget();
}
```

Does this example compile?
If yes, what does it print?

1/3

```
trait Forgettable {
    fn forget(&self);
}
1 implementation
struct Thing { name: String }
impl Forgettable for Thing {
    fn forget(&self) {
        println!("I'm forgetting {}", self.name);
    }
}
pub fn main() {
    let thing: Thing = Thing { name: String::from("my thing") };
    thing.forget();
    let thing2: Thing = Thing { name: String::from("my other thing") };
    thing2.forget();
}
```

Does this example compile?
If yes, what does it print?

1/3

```
trait Forgettable {
    fn forget(&self);
}
```

Trait declaration

```
1 implementation
struct Thing { name: String }
impl Forgettable for Thing {
    fn forget(&self) {
        println!("I'm forgetting {}", self.name);
    }
}
```

Trait implementation

```
pub fn main() {
    let thing: Thing = Thing { name: String::from("my thing") };
    thing.forget();
    let thing2: Thing = Thing { name: String::from("my other thing") };
    thing2.forget();
}
```

1/3

Does this example compile?
If yes, what does it print?

```
trait Forgettable {
    fn forget(&self);
}
1 implementation
struct Thing { name: String }
impl Forgettable for Thing {
    fn forget(&self) {
        println!("I'm forgetting {}", self.name);
    }
}
pub fn main() {
    let thing: Thing = Thing { name: String::from("my thing") };
    thing.forget();
    let thing2: Thing = Thing { name: String::from("my other thing") };
    thing2.forget();
}
```

Each struct defines all necessary functions, there are no supertraits or other things

1/3

Does this example compile?
If yes, what does it print?

```
trait Forgettable {
    fn forget(&self);
}
1 implementation
struct Thing { name: String }
impl Forgettable for Thing {
    fn forget(&self) {
        println!("I'm forgetting {}", self.name);
    }
}
pub fn main() {
    let thing: Thing = Thing { name: String::from("my thing") };
    thing.forget();
    let thing2: Thing = Thing { name: String::from("my other thing") };
    thing2.forget();
}
```

Each struct defines all necessary functions, there are no supertraits or other things

This code compiles :)

```
I'm forgetting my thing
I'm forgetting my other thing
```

3. Next time

- Enums
- match