RUSTikales Rust for beginners

Plan for today

Plan for today

1. Recap

Plan for today

1. Recap
2. Function Extra

Plan for today

1. Recap
2. Function Extra
3. Structs

Plan for today

1. Recap
2. Function Extra
3. Structs
    a. Declaration

Plan for today

1. Recap
2. Function Extra
3. Structs
    a. Declaration
    b. Methods

1. Recap

1. Recap

- Ownership-Model
- Borrowing
- Borrow Checker

1. Recap

- Ownership-Model
- Borrowing
- Borrow Checker
- Functions are declared using the keyword fn
    - Functions can take in Parameters:
        - are almost identical to normal variables
        - define how you call the function
    - Functions can have a Return type:
        - If declared, all return statements need to return a value of the given type

1. Recap

- Ownership-Model
- Borrowing
- Borrow Checker
- Functions are declared using the keyword fn
  - Functions can take in Parameters:
    - are almost identical to normal variables
    - define how you call the function
  - Functions can have a Return type:
    - If declared, all return statements need to return a value of the given type
- Functions are called by passing arguments
  - For every parameter you use in the declaration, you need to pass one argument
  - Order of arguments matches order of parameters

1. Recap

```rust
fn add(n: i32, m: i32) -> i32 {
    return n + m;
}

▶ Run | Debug
fn main() {
    let n: i32 = 10;
    let m: i32 = 20;
    let sum: i32 = add(n, m);
    println!("{} + {} = {}", n, m, sum);
}
```

1. Recap

```rust
fn add(n: i32, m: i32) -> i32 {
    return n + m;
}

▶ Run | Debug
fn main() {
    let n: i32 = 10;
    let m: i32 = 20;
    let sum: i32 = add(n, m);
    println!("{} + {} = {}", n, m, sum);
}
```

2 parameters of types i32

1. Recap

```rust
fn add(n: i32, m: i32) -> i32 {
    return n + m;
}

▶ Run | Debug
fn main() {
    let n: i32 = 10;
    let m: i32 = 20;
    let sum: i32 = add(n, m);
    println!("{} + {} = {}", n, m, sum);
}
```

2 parameters of types i32

2 arguments of types i32

1. Recap

```rust
fn add(n: i32, m: i32) -> i32 {
    return n + m;
}

▶ Run | Debug
fn main() {
    let n: i32 = 10;
    let m: i32 = 20;
    let sum: i32 = add(n, m);
    println!("{} + {} = {}", n, m, sum);
}
```

Return type i32

1. Recap

```rust
fn add(n: i32, m: i32) -> i32 {
    return n + m;
}

▶ Run | Debug
fn main() {
    let n: i32 = 10;
    let m: i32 = 20;
    let sum: i32 = add(n, m);
    println!("{} + {} = {}", n, m, sum);
}
```

1. Recap

```rust
fn add(n: i32, m: i32) -> i32 {
    return n + m;
}

▶ Run | Debug
fn main() {
    let n: i32 = 10;
    let m: i32 = 20;
    let sum: i32 = add(n, m);
    println!("{} + {} = {}", n, m, sum);
}
```

Return type i32

Returns i32, so we can assign to i32

2. Function Extra

```
fn forbidden(n: i32 = 1) {

}
```

No default arguments in Rust

2. Function Extra

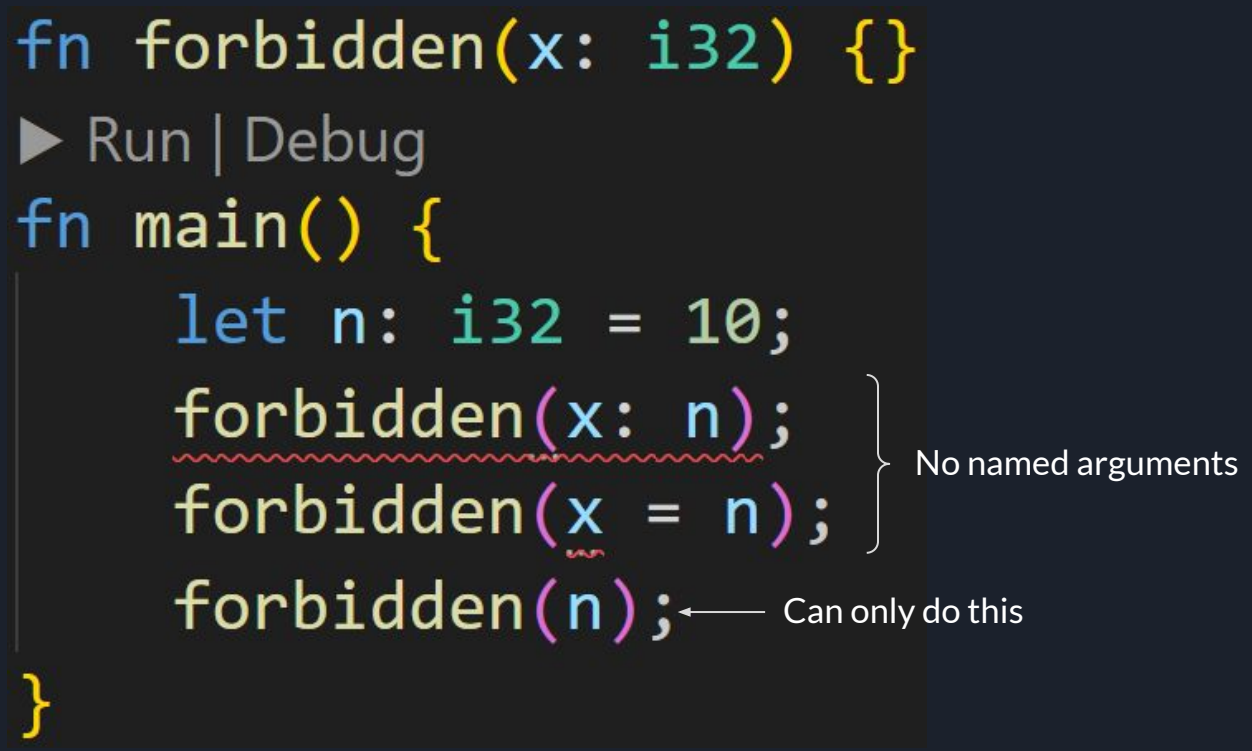
```
fn forbidden(x: i32) {}
▶ Run | Debug
fn main() {
    let n: i32 = 10;
    forbidden(x: n);
    forbidden(x = n);
    forbidden(n);
}
```

2. Function Extra

```
fn forbidden(x: i32) {}
▶ Run | Debug
fn main() {
    let n: i32 = 10;
    forbidden(x: n);
    forbidden(x = n);
    forbidden(n);
}
```

No named arguments

2. Function Extra



2. Function Extra

- Rust does not support overloaded functions:
  - There can only ever be a single function declaration with a given name in the module

```
fn no_overload() {}
fn no_overload(x: i32) {}
```

3. Structs

- Now that we're familiar with functions and some basic types, we can try to write our first real program:

3. Structs

- Now that we're familiar with functions and some basic types, we can try to write our first real program:
  - We want to create a console application where a ball bounces around the screen

3. Structs

- Now that we're familiar with functions and some basic types, we can try to write our first real program:
  - We want to create a console application where a ball bounces around the screen
- Note: You can find the source code for this application in this weeks directory, in a project called `ball_app`

```
▶ Run | Debug
fn main() {
    let width: i32 = 60;
    let height: i32 = 40;
    let mut ball_x: i32 = width / 2;
    let mut ball_y: i32 = height / 2;
    let mut ball_x_speed: i32 = 1;
    let mut ball_y_speed: i32 = 1;
    loop {
        print_window(ball_x, ball_y, width, height);
        // Ball bounce
        if ball_x == 0 || ball_x == width - 1 {
            ball_x_speed = -ball_x_speed;
        }
        if ball_y == 0 || ball_y == height - 1 {
            ball_y_speed = -ball_y_speed;
        }
        // Ball move
        ball_x += ball_x_speed;
        ball_y += ball_y_speed;
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
}
```

```
▶ Run | Debug
fn main() {
    let width: i32 = 60;
    let height: i32 = 40;
    let mut ball_x: i32 = width / 2;
    let mut ball_y: i32 = height / 2;
    let mut ball_x_speed: i32 = 1;
    let mut ball_y_speed: i32 = 1;
    loop {
        print_window(ball_x, ball_y, width, height);
        // Ball bounce
        if ball_x == 0 || ball_x == width - 1 {
            ball_x_speed = -ball_x_speed;
        }
        if ball_y == 0 || ball_y == height - 1 {
            ball_y_speed = -ball_y_speed;
        }
        // Ball move
        ball_x += ball_x_speed;
        ball_y += ball_y_speed;
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
}
```

Basic loop, we never want to stop the application

```
▶ Run | Debug
fn main() {
    let width: i32 = 60;
    let height: i32 = 40;
    let mut ball_x: i32 = width / 2;
    let mut ball_y: i32 = height / 2;
    let mut ball_x_speed: i32 = 1;
    let mut ball_y_speed: i32 = 1;
    loop {
        print_window(ball_x, ball_y, width, height);
        // Ball bounce
        if ball_x == 0 || ball_x == width - 1 {
            ball_x_speed = -ball_x_speed;
        }
        if ball_y == 0 || ball_y == height - 1 {
            ball_y_speed = -ball_y_speed;
        }
        // Ball move
        ball_x += ball_x_speed;
        ball_y += ball_y_speed;
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
}
```

Variables to store the application state, like the ball position and velocity

```
▶ Run | Debug
fn main() {
    let width: i32 = 60;
    let height: i32 = 40;
    let mut ball_x: i32 = width / 2;
    let mut ball_y: i32 = height / 2;
    let mut ball_x_speed: i32 = 1;
    let mut ball_y_speed: i32 = 1;
    loop {
        print_window(ball_x, ball_y, width, height);
        // Ball bounce
        if ball_x == 0 || ball_x == width - 1 {
            ball_x_speed = -ball_x_speed;
        }
        if ball_y == 0 || ball_y == height - 1 {
            ball_y_speed = -ball_y_speed;
        }
        // Ball move
        ball_x += ball_x_speed;
        ball_y += ball_y_speed;
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
}
```

Variables to store the application state, like the ball position and velocity

Function to print the window to the console, defined above main()

```
▶ Run | Debug
fn main() {
    let width: i32 = 60;
    let height: i32 = 40;
    let mut ball_x: i32 = width / 2;
    let mut ball_y: i32 = height / 2;
    let mut ball_x_speed: i32 = 1;
    let mut ball_y_speed: i32 = 1;
    loop {
        print_window(ball_x, ball_y, width, height);
        // Ball bounce
        if ball_x == 0 || ball_x == width - 1 {
            ball_x_speed = -ball_x_speed;
        }
        if ball_y == 0 || ball_y == height - 1 {
            ball_y_speed = -ball_y_speed;
        }
        // Ball move
        ball_x += ball_x_speed;
        ball_y += ball_y_speed;
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
}
```

Variables to store the application state, like the ball position and velocity

Function to print the window to the console, defined above main()

If we're at the edge of the window, simulate bounce by inverting velocity

```
▶ Run | Debug
fn main() {
    let width: i32 = 60;
    let height: i32 = 40;
    let mut ball_x: i32 = width / 2;
    let mut ball_y: i32 = height / 2;
    let mut ball_x_speed: i32 = 1;
    let mut ball_y_speed: i32 = 1;
    loop {
        print_window(ball_x, ball_y, width, height);
        // Ball bounce
        if ball_x == 0 || ball_x == width - 1 {
            ball_x_speed = -ball_x_speed;
        }
        if ball_y == 0 || ball_y == height - 1 {
            ball_y_speed = -ball_y_speed;
        }
        // Ball move
        ball_x += ball_x_speed;
        ball_y += ball_y_speed;
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
}
```

Variables to store the application state, like the ball position and velocity

Function to print the window to the console, defined above main()

If we're at the edge of the window, simulate bounce by inverting velocity

Update position of ball

```
▶ Run | Debug
fn main() {
    let width: i32 = 60;
    let height: i32 = 40;
    let mut ball_x: i32 = width / 2;
    let mut ball_y: i32 = height / 2;
    let mut ball_x_speed: i32 = 1;
    let mut ball_y_speed: i32 = 1;
    loop {
        print_window(ball_x, ball_y, width, height);
        // Ball bounce
        if ball_x == 0 || ball_x == width - 1 {
            ball_x_speed = -ball_x_speed;
        }
        if ball_y == 0 || ball_y == height - 1 {
            ball_y_speed = -ball_y_speed;
        }
        // Ball move
        ball_x += ball_x_speed;
        ball_y += ball_y_speed;
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
}
```

Variables to store the application state, like the ball position and velocity

Function to print the window to the console, defined above main()

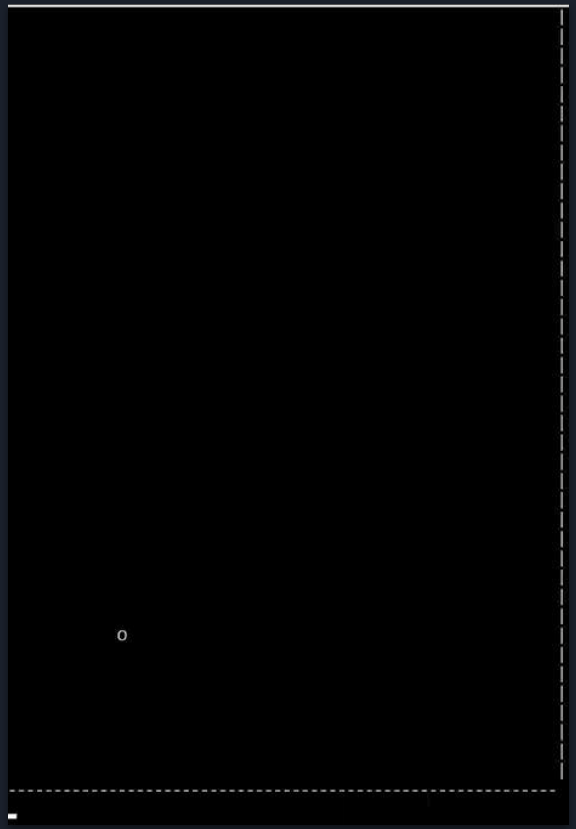
If we're at the edge of the window, simulate bounce by inverting velocity

Update position of ball

sleep() freezes the application for the given duration so we don't eat 100% of the CPU, here we're simulating 60 frames per second (we draw to the console 60 times per second)

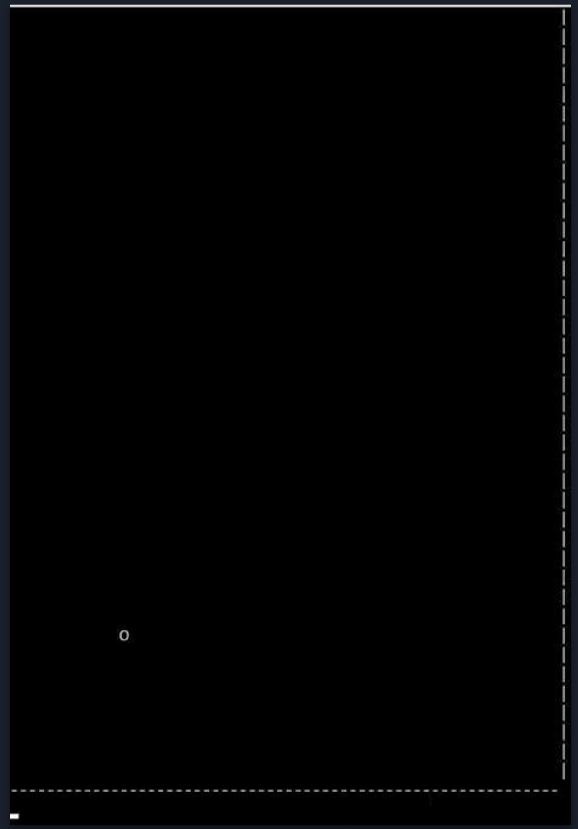
3. Structs

- Demo:



3. Structs

- Demo:
- Not pretty, but it works :^)



```
▶ Run | Debug
fn main() {
    let width: i32 = 60;
    let height: i32 = 40;
    let mut ball_x: i32 = width / 2;
    let mut ball_y: i32 = height / 2;
    let mut ball_x_speed: i32 = 1;
    let mut ball_y_speed: i32 = 1;
    loop {
        print_window(ball_x, ball_y, width, height);
        // Ball bounce
        if ball_x == 0 || ball_x == width - 1 {
            ball_x_speed = -ball_x_speed;
        }
        if ball_y == 0 || ball_y == height - 1 {
            ball_y_speed = -ball_y_speed;
        }
        // Ball move
        ball_x += ball_x_speed;
        ball_y += ball_y_speed;
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
}
```

Notice how there are a lot of variables dedicated to keeping track of the ball state?

```
▶ Run | Debug
fn main() {
    let width: i32 = 60;
    let height: i32 = 40;
    let mut ball_x: i32 = width / 2;
    let mut ball_y: i32 = height / 2;
    let mut ball_x_speed: i32 = 1;
    let mut ball_y_speed: i32 = 1;
    loop {
        print_window(ball_x, ball_y, width, height);
        // Ball bounce
        if ball_x == 0 || ball_x == width - 1 {
            ball_x_speed = -ball_x_speed;
        }
        if ball_y == 0 || ball_y == height - 1 {
            ball_y_speed = -ball_y_speed;
        }
        // Ball move
        ball_x += ball_x_speed;
        ball_y += ball_y_speed;
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
}
```

Initialization

Modification

Notice how there are a lot of variables dedicated to keeping track of the ball state?

3. Structs

- Is it possible to organize all those variables in a structured way, so we can easily generate and manage more balls?

3. Structs

- Is it possible to organize all those variables in a structured way, so we can easily generate and manage more balls?
- The answer is yes, we can!
- Today, you'll learn about structs, and how they make our small application more readable and extendable

3. Structs - Declaration

- Structs in Rust are collections of values

3. Structs - Declaration

- Structs in Rust are collections of values
- We've seen one before: Vector!
    - A Vector in Rust is also a struct
    - Each Vector has values:
        - length
        - capacity
        - elements

3. Structs - Declaration

- Structs in Rust are collections of values
- We've seen one before: Vector!
- Structs in Rust are declared using the keyword struct

3. Structs - Declaration

- Structs in Rust are collections of values
- We've seen one before: Vector!
- Structs in Rust are declared using the keyword struct
- Structs can contain any number of fields, even 0

3. Structs - Declaration

- Structs in Rust are collections of values
- We've seen one before: Vector!
- Structs in Rust are declared using the keyword struct
- Structs can contain any number of fields, even 0
    - A simple Vector has three fields:
        - length
        - capacity
        - elements

3. Structs - Declaration

- Structs in Rust are collections of values
- We've seen one before: Vector!
- Structs in Rust are declared using the keyword struct
- Structs can contain any number of fields, even 0
- Each instance of a struct has its own field values

3. Structs - Declaration

- Structs in Rust are collections of values
- We've seen one before: Vector!
- Structs in Rust are declared using the keyword struct
- Structs can contain any number of fields, even 0
- Each instance of a struct has its own field values
- You access fields of a struct instance using the dot syntax

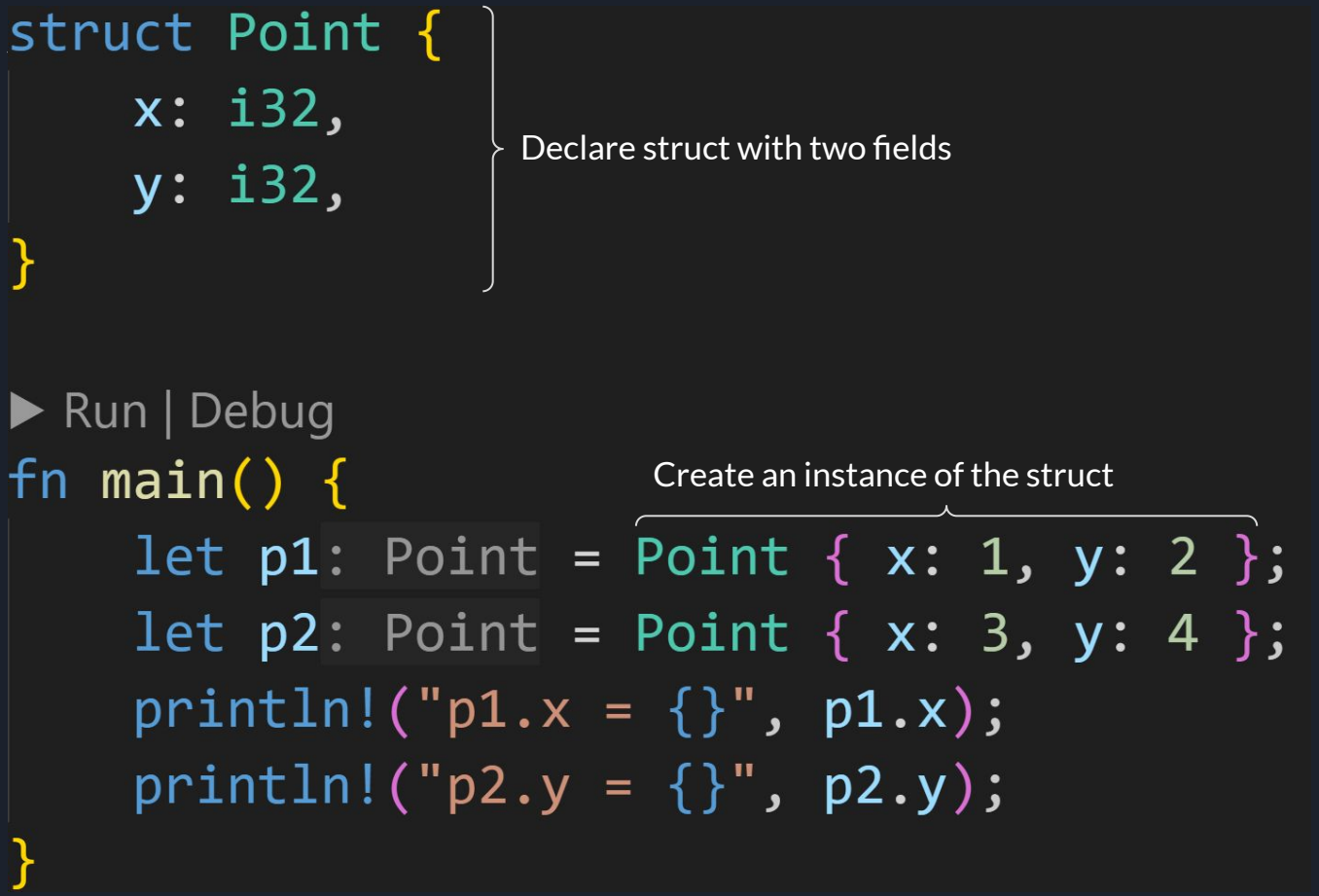
```rust
struct Point {
    x: i32,
    y: i32,
}

▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 1, y: 2 };
    let p2: Point = Point { x: 3, y: 4 };
    println!("p1.x = {}", p1.x);
    println!("p2.y = {}", p2.y);
}
```

```rust
struct Point {
    x: i32,
    y: i32,
}
```

Declare struct with two fields

▶ Run | Debug

```rust
fn main() {
    let p1: Point = Point { x: 1, y: 2 };
    let p2: Point = Point { x: 3, y: 4 };
    println!("p1.x = {}", p1.x);
    println!("p2.y = {}", p2.y);
}
```

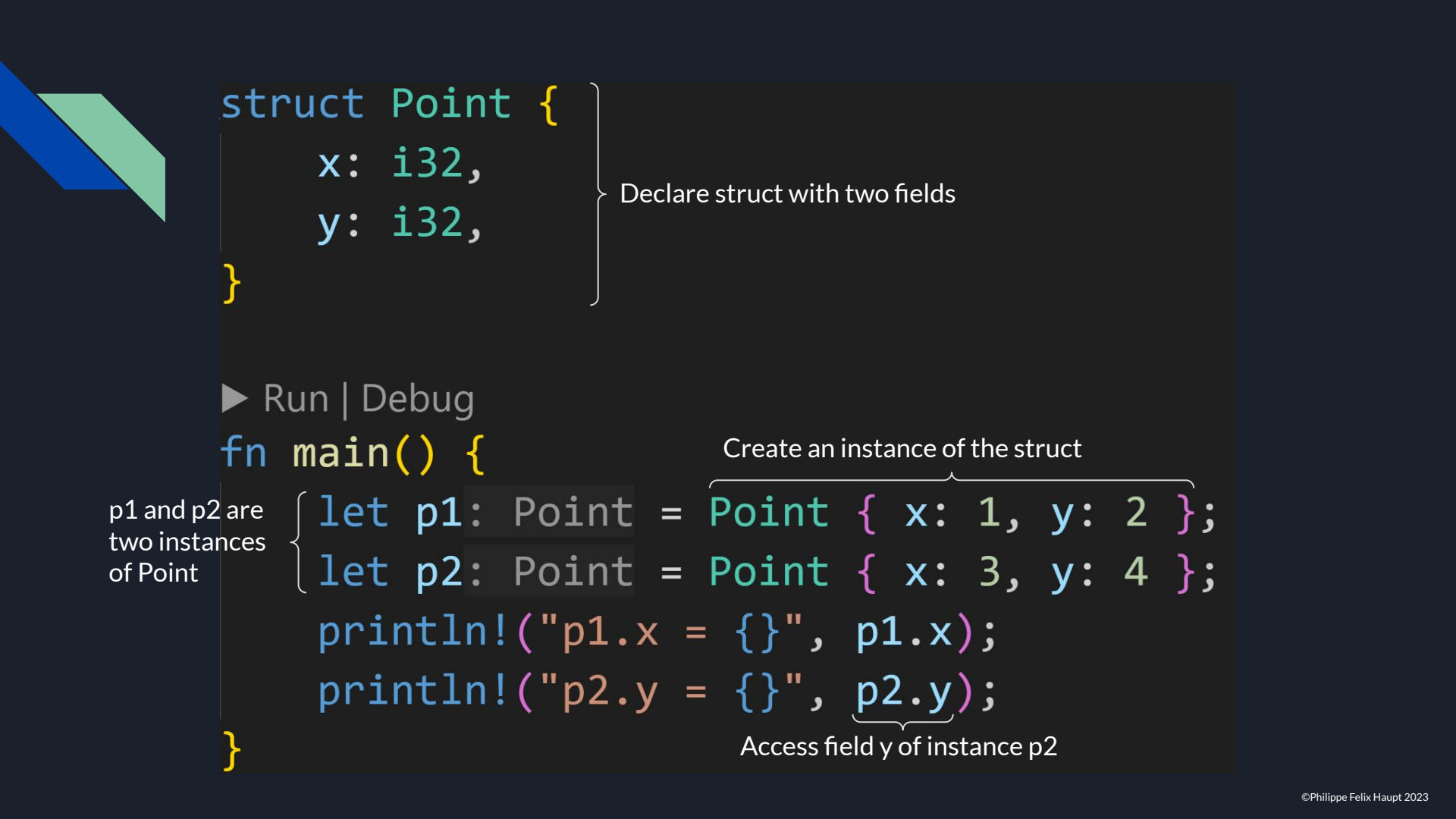


```rust
struct Point {
    x: i32,
    y: i32,
}

▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 1, y: 2 };
    let p2: Point = Point { x: 3, y: 4 };
    println!("p1.x = {}", p1.x);
    println!("p2.y = {}", p2.y);
}
```

Declare struct with two fields

Create an instance of the struct

p1 and p2 are two instances of Point



3. Structs - Declaration

- Structs are type definitions

3. Structs - Declaration

- Structs are type definitions
- By defining the struct Point, we can:
  - Pass instances of Point as arguments/parameters
  - Pass instances of Point as return values
  - Declare a struct with a field of type Point (With a caveat)
  - and much more

3. Structs - Declaration

- Structs are type definitions
- By defining the struct Point, we can:
  - Pass instances of Point as arguments/parameters
  - Pass instances of Point as return values
  - Declare a struct with a field of type Point (With a caveat)
  - and much more
- Normal Ownership and Borrow Checker rules apply to our structs as well:
  - By default, struct instances are always moved

3. Structs - Declaration

- Structs are type definitions
- By defining the struct Point, we can:
  - Pass instances of Point as arguments/parameters
  - Pass instances of Point as return values
  - Declare a struct with a field of type Point (With a caveat)
  - and much more
- Normal Ownership and Borrow Checker rules apply to our structs as well:
  - By default, struct instances are always moved

```
let p1: Point = Point { x: 1, y: 2 };
let p2: Point = p1;
println!("p1.x = {}", p1.x);
println!("p2.y = {}", p2.y);
```

3. Structs - Declaration

- Structs are type definitions
- By defining the struct Point, we can:
  - Pass instances of Point as arguments/parameters
  - Pass instances of Point as return values
  - Declare a struct with a field of type Point (With a caveat)
  - and much more
- Normal Ownership and Borrow Checker rules apply to our structs as well:
  - By default, struct instances are always moved

```
let p1: Point = Point { x: 1, y: 2 };
let p2: Point = p1;
println!("p1.x = {}", p1.x);
println!("p2.y = {}", p2.y);
```

We know what this means :^)

3. Structs - Declaration

- Structs are type definitions
- By defining the struct Point, we can:
  - Pass instances of Point as arguments/parameters
  - Pass instances of Point as return values
  - Declare a struct with a field of type Point (With a caveat)
  - and much more
- Normal Ownership and Borrow Checker rules apply to our structs as well:
  - By default, struct instances are always moved
  - Using traits we can tell Rust to not move the struct

3. Structs - Declaration

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

▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 1, y: 2 };
    let p2: Point = p1;
    println!("p1.x = {}", p1.x);
    println!("p2.y = {}", p2.y);
}
```

3. Structs - Declaration

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

▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 1, y: 2 };
    let p2: Point = p1;
    println!("p1.x = {}", p1.x);
    println!("p2.y = {}", p2.y);
}
```

Using Rust magic, we can now copy our Points!

p1 is copied now!

3. Structs - Declaration

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

▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 1, y: 2 };
    let p2: Point = p1;
    println!("p1.x = {}", p1.x);
    println!("p2.y = {}", p2.y);
}
```

Using Rust magic, we can now copy our Points!

What does derive mean?
What does Copy mean?
What does Clone mean?
→ More on that later

p1 is copied now!

3. Structs - Declaration

- Structs in Rust are always located on the Stack

3. Structs - Declaration

- Structs in Rust are always located on the Stack
- This means that the size of a struct must be known at compile time

3. Structs - Declaration

- Structs in Rust are always located on the Stack
- This means that the size of a struct must be known at compile time
- But there’s an easy example for when this is impossible to know

3. Structs - Declaration

- Structs in Rust are always located on the Stack
- This means that the size of a struct must be known at compile time
- But there's an easy example for when this is impossible to know
    - When?

2/3

3. Structs - Declaration

- Structs in Rust are always located on the Stack
- This means that the size of a struct must be known at compile time
- But there's an easy example for when this is impossible to know
- Correct, Recursion :)

```
struct Point {
    x: i32,
    y: i32,
    p: Point,
}
```

3. Structs - Declaration

- Structs in Rust are always located on the Stack
- This means that the size of a struct must be known at compile time
- But there's an easy example for when this is impossible to know
- Correct, Recursion :)

```
struct Point {
    x: i32,
    y: i32,
    p: Point,
}
```

3. Structs - Declaration

- Structs in Rust are always located on the Stack
- This means that the size of a struct must be known at compile time
- But there's an easy example for when this is impossible to know
- Correct, Recursion :)

To get the size of a struct, we simply add the sizes of all fields together, but here we don't know the size of field `p`!
Point = i32 + i32 + Point

```
struct Point {
    x: i32,
    y: i32,
    p: Point,
}
```

3. Structs - Declaration

- Structs in Rust are always located on the Stack
- This means that the size of a struct must be known at compile time
- But there's an easy example for when this is impossible to know
- Correct, Recursion :)
- There's a solution to this problem: We break the size explosion by moving the field to the Heap:

```
struct Point {
    x: i32,
    y: i32,
    p: Box<Point>,
}
```

3. Structs - Declaration

- Structs in Rust are always located on the Stack
- This means that the size of a struct must be known at compile time
- But there's an easy example for when this is impossible to know
- Correct, Recursion :)
- There's a solution to this problem: We break the size explosion by moving the field to the Heap:

```
struct Point {
    x: i32,
    y: i32,
    p: Box<Point>,
}
```

Box is a Wrapper for a Pointer to the heap

3. Structs - Declaration

- Structs in Rust are always located on the Stack
- This means that the size of a struct must be known at compile time
- But there's an easy example for when this is impossible to know
- Correct, Recursion :)
- There's a solution to this problem: We break the size explosion by moving the field to the Heap:

3/3

```
struct Point {
    x: i32,
    y: i32,
    p: Box<Point>,
}
```

Box is a Wrapper for a Pointer to the heap

How does this solve our problem?

3. Structs - Declaration

- Structs in Rust are always located on the Stack
- This means that the size of a struct must be known at compile time
- But there's an easy example for when this is impossible to know
- Correct, Recursion :)
- There's a solution to this problem: We break the size explosion by moving the field to the Heap:

3/3

```
struct Point {
    x: i32,
    y: i32,
    p: Box<Point>,
}
```

Box is a Wrapper for a Pointer to the heap

How does this solve our problem?

We know the size of a Box, it's at most 8 bytes :) It does not care about the underlying struct

3. Structs - Methods

- Structs themselves are powerful, but we can go a step further, by declaring associated functions

3. Structs - Methods

- Structs themselves are powerful, but we can go a step further, by declaring associated functions
- Associated functions are associated with a type
  - We've seen some of those too
  - Vec::new() associates the function new() with the type Vec
  - f32::sqrt() associates the function sqrt() with the type f32

3. Structs - Methods

- Structs themselves are powerful, but we can go a step further, by declaring associated functions
- Associated functions are associated with a type
  - We've seen some of those too
  - Vec::new() associates the function new() with the type Vec
  - f32::sqrt() associates the function sqrt() with the type f32
- Associated functions have an advantage over normal functions:
  - We can have multiple associated functions with the same name
  - Point::new() does not overlap with Vec::new(), those are two different functions
    - Point::new() still overlaps with Point::new(x: i32) though

3. Structs - Methods

- Structs themselves are powerful, but we can go a step further, by declaring associated functions
- Associated functions are associated with a type
  - We've seen some of those too
  - Vec::new() associates the function new() with the type Vec
  - f32::sqrt() associates the function sqrt() with the type f32
- Associated functions have an advantage over normal functions:
  - We can have multiple associated functions with the same name
  - Point::new() does not overlap with Vec::new(), those are two different functions
    - Point::new() still overlaps with Point::new(x: i32) though
- Associated functions can be implemented for any type using the keyword impl

3. Structs - Methods

```
struct Point {
    x: i32,
    y: i32,
}
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
```

3. Structs - Methods

```
struct Point {
    x: i32,
    y: i32,
}
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
```

Any function defined in this bracket block is a function associated with Point and can be called using the syntax Point::function_name()
Here, we define Point::new()

3. Structs - Methods

```
struct Point {
    x: i32,
    y: i32,
}
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
```

We don’t need to type Point everywhere, Rust has Self: Self refers to the type of the impl-block, here it’s Point

3. Structs - Methods

```
struct Point {
    x: i32,
    y: i32,
}
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
```

We don't need to type Point everywhere, Rust has Self: Self refers to the type of the impl-block, here it's Point

Syntax sugar for: return Point { x, y };

3. Structs - Methods

```
struct Point {
    x: i32,
    y: i32,
}
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
```

We don't need to type Point everywhere, Rust has Self: Self refers to the type of the impl-block, here it's Point

Syntax sugar for: return Point { x, y };

Syntax sugar for:
x: x,
y: y

3. Structs - Methods

```
struct Point {
    x: i32,
    y: i32,
}
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
}
```

3. Structs - Methods

```rust
struct Point {
    x: i32,
    y: i32,
}
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
}
```

Because new() is an associated function, we need to specify the type before, otherwise Rust wouldn't know which new() to call

3. Structs - Methods

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(p1: &Point, p2: &Point) -> f64 {
        let dx: i32 = (p2.x - p1.x);
        let dy: i32 = (p2.y - p1.y);
        let x_squared: f64 = (dx * dx) as f64;
        let y_squared: f64 = (dy * dy) as f64;
        return f64::sqrt(self: x_squared + y_squared);
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let d: f64 = Point::distance(&p1, &p2);
    println!("d = {}", d);
```

3. Structs - Methods

1/3

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(p1: &Point, p2: &Point) -> f64 {
        let dx: i32 = (p2.x - p1.x);
        let dy: i32 = (p2.y - p1.y);
        let x_squared: f64 = (dx * dx) as f64;
        let y_squared: f64 = (dy * dy) as f64;
        return f64::sqrt(self: x_squared + y_squared);
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let d: f64 = Point::distance(&p1, &p2);
    println!("d = {}", d);
```

How many calls to associated functions are in this code snippet?

3. Structs - Methods

1/3

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(p1: &Point, p2: &Point) -> f64 {
        let dx: i32 = (p2.x - p1.x);
        let dy: i32 = (p2.y - p1.y);
        let x_squared: f64 = (dx * dx) as f64;
        let y_squared: f64 = (dy * dy) as f64;
        return f64::sqrt(self: x_squared + y_squared);
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let d: f64 = Point::distance(&p1, &p2);
    println!("d = {}", d);
```

How many calls to associated functions are in this code snippet?

Answer: 4

3. Structs - Methods

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(p1: &Point, p2: &Point) -> f64 {
        let dx: i32 = (p2.x - p1.x);
        let dy: i32 = (p2.y - p1.y);
        let x_squared: f64 = (dx * dx) as f64;
        let y_squared: f64 = (dy * dy) as f64;
        return f64::sqrt(self: x_squared + y_squared);
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let d: f64 = Point::distance(&p1, &p2);
    println!("d = {}", d);
```

Associated function that calculates the distance between two given points

3. Structs - Methods

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(p1: &Point, p2: &Point) -> f64 {
        let dx: i32 = (p2.x - p1.x);
        let dy: i32 = (p2.y - p1.y);
        let x_squared: f64 = (dx * dx) as f64;
        let y_squared: f64 = (dy * dy) as f64;
        return f64::sqrt(self: x_squared + y_squared);
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let d: f64 = Point::distance(&p1, &p2);
    println!("d = {}", d);
```

We're only borrowing, we don't need to take ownership

Associated function that calculates the distance between two given points

3. Structs - Methods

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(p1: &Point, p2: &Point) -> f64 {
        let dx: i32 = (p2.x - p1.x);
        let dy: i32 = (p2.y - p1.y);
        let x_squared: f64 = (dx * dx) as f64;
        let y_squared: f64 = (dy * dy) as f64;
        return f64::sqrt(self: x_squared + y_squared);
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let d: f64 = Point::distance(&p1, &p2);
    println!("d = {}", d);
```

Associated function that calculates the distance between two given points

Call that function, and get the distance between p1 and p2

3. Structs - Methods

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(p1: &Point, p2: &Point) -> f64 {
        let dx: i32 = (p2.x - p1.x);
        let dy: i32 = (p2.y - p1.y);
        let x_squared: f64 = (dx * dx) as f64;
        let y_squared: f64 = (dy * dy) as f64;
        return f64::sqrt(self: x_squared + y_squared);
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let d: f64 = Point::distance(&p1, &p2);
    println!("d = {}", d);
```

This is a lot of typing, can this be shorter?

3. Structs - Methods

- There’s a special form of associated functions, called methods

3. Structs - Methods

- There’s a special form of associated functions, called methods
- Methods are easy to spot: The first parameter is self

3. Structs - Methods

- There’s a special form of associated functions, called methods
- Methods are easy to spot: The first parameter is self
- Methods allow us to easily call associated functions on instances
    - They unlock the method call operator

3. Structs - Methods

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(&self, other: &Point) -> f64 {
        let dx: i32 = (other.x - self.x);
        let dy: i32 = (other.y - self.y);
        let x_squared: f64 = (dx * dx) as f64;
        let y_squared: f64 = (dy * dy) as f64;
        return f64::sqrt(self: x_squared + y_squared);
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let d: f64 = p1.distance(&p2);
    println!("d = {}", d);
```

3. Structs - Methods

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(&self, other: &Point) -> f64 {
        let dx: i32 = (other.x - self.x);
        let dy: i32 = (other.y - self.y);
        let x_squared: f64 = (dx * dx) as f64;
        let y_squared: f64 = (dy * dy) as f64;
        return f64::sqrt(self: x_squared + y_squared);
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let d: f64 = p1.distance(&p2);
    println!("d = {}", d);
```

Method that calculates the distances between self and other

3. Structs - Methods

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(&self, other: &Point) -> f64 {
        let dx: i32 = (other.x - self.x);
        let dy: i32 = (other.y - self.y);
        let x_squared: f64 = (dx * dx) as f64;
        let y_squared: f64 = (dy * dy) as f64;
        return f64::sqrt(self: x_squared + y_squared);
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let d: f64 = p1.distance(&p2);
    println!("d = {}", d);
```

First parameter is simply self, without any type
Rust knows what type self must have :)

Method that calculates the distances between self and other

3. Structs - Methods

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(&self, other: &Point) -> f64 {
        let dx: i32 = (other.x - self.x);
        let dy: i32 = (other.y - self.y);
        let x_squared: f64 = (dx * dx) as f64;
        let y_squared: f64 = (dy * dy) as f64;
        return f64::sqrt(self: x_squared + y_squared);
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let d: f64 = p1.distance(&p2);
    println!("d = {}", d);
```

First parameter is simply self, without any type
Rust knows what type self must have :)

Method that calculates the distances between self and other

Method call operator:
Note that we're only passing one argument, even though we defined two parameters above!
The first parameter self is implicit, here p1 is passed as self

3. Structs - Methods

```
    fn update(&mut self, x: i32, y: i32) {
        self.x = x;
        self.y = y;
    }
}
▶ Run | Debug
fn main() {
    let mut p1: Point = Point::new(x: 1, y: 2);
    p1.update(x: 3, y: 4);
```

3. Structs - Methods

```
    fn update(&mut self, x: i32, y: i32) {
        self.x = x;
        self.y = y;
    }
}
▶ Run | Debug
fn main() {
    let mut p1: Point = Point::new(x: 1, y: 2);
    p1.update(x: 3, y: 4);
```

Of course we can also mutable borrow!

3. Structs - Methods

```
    fn update(&mut self, x: i32, y: i32) {
        self.x = x;
        self.y = y;
    }
}
▶ Run | Debug
fn main() {
    let mut p1: Point = Point::new(x: 1, y: 2);
    p1.update(x: 3, y: 4);
```

Of course we can also mutable borrow!

After this line, p1.x=3 and p1.y=4

3. Structs - Methods

```
    fn nom(self) {
        println!("nomnom, I ate self");
    }
}
▶ Run | Debug
fn main() {
    let mut p1: Point = Point::new(x: 1, y: 2);
    p1.nom();
    println!("p1.x = {}", p1.x);
```

3. Structs - Methods

```
    fn nom(self) {
        println!("nomnom, I ate self");
    }
}
▶ Run | Debug
fn main() {
    let mut p1: Point = Point::new(x: 1, y: 2);
    p1.nom();
    println!("p1.x = {}", p1.x);
```

Method takes ownership of self, it's dropped at the end of the pink block

3. Structs - Methods

```
    fn nom(self) {
        println!("nomnom, I ate self");
    }
}

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

Method takes ownership of self, it's dropped at the end of the pink block

Can't use p1 down here anymore!

3. Structs - Methods

- Using our newfound knowledge, we can take a look at the ball application again

3. Structs - Methods

- Using our newfound knowledge, we can take a look at the ball application again
- We can replace our variables with a simple struct that keeps track of all the values

3. Structs - Methods

- Using our newfound knowledge, we can take a look at the ball application again
- We can replace our variables with a simple struct that keeps track of all the values

```
struct Ball {
    x: i32,
    y: i32,
    x_speed: i32,
    y_speed: i32,
}
```

3. Structs - Methods

- Then, we can implement some methods for the Ball struct

3. Structs - Methods

- Then, we can implement some methods for the Ball struct

```
impl Ball {
    fn check_bounce(&mut self, width: i32, height: i32) {
        if self.x == 0 || self.x == width - 1 {
            self.x_speed = -self.x_speed;
        }
        if self.y == 0 || self.y == height - 1 {
            self.y_speed = -self.y_speed;
        }
    }
    fn move_ball(&mut self) {
        self.x += self.x_speed;
        self.y += self.y_speed;
    }
}
```

3. Structs - Methods

- Then, we can implement some methods for the Ball struct

```
impl Ball {
    fn check_bounce(&mut self, width: i32, height: i32) {
        if self.x == 0 || self.x == width - 1 {
            self.x_speed = -self.x_speed;
        }
        if self.y == 0 || self.y == height - 1 {
            self.y_speed = -self.y_speed;
        }
    }
    fn move_ball(&mut self) {
        self.x += self.x_speed;
        self.y += self.y_speed;
    }
}
```

Putting ball in the name is kind of redundant, but move is a built in keyword and is reserved :^)

3. Structs - Methods

- Finally, we can update our main function!

3. Structs - Methods

- Finally, we can update our main function!

```
fn main() {
    let width: i32 = 60;
    let height: i32 = 40;
    let mut ball: Ball = Ball {
        x: width / 2,
        y: height / 2,
        x_speed: 1,
        y_speed: 1,
    };
    loop {
        print_window(&ball, width, height);
        ball.check_bounce(width, height);
        ball.move_ball();
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
}
```

3. Structs - Methods

- Finally, we can update our main function!

```
fn main() {
    let width: i32 = 60;
    let height: i32 = 40;
    let mut ball: Ball = Ball {
        x: width / 2,
        y: height / 2,
        x_speed: 1,
        y_speed: 1,
    };
    loop {
        print_window(&ball, width, height);
        ball.check_bounce(width, height);
        ball.move_ball();
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
}
```

Initialize the instance

3. Structs - Methods

- Finally, we can update our main function!

```
fn main() {
    let width: i32 = 60;
    let height: i32 = 40;
    let mut ball: Ball = Ball {
        x: width / 2,
        y: height / 2,
        x_speed: 1,
        y_speed: 1,
    };
    loop {
        print_window(&ball, width, height);
        ball.check_bounce(width, height);
        ball.move_ball();
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
}
```

Initialize the instance

Call some methods

3. Structs - Methods

- Now that we've switched to a struct, we can easily extend our application by adding more balls!

3. Structs - Methods

- Now that we've switched to a struct, we can easily extend our application by adding more balls!

```
        y: height / 4,
        x_speed: 1,
        y_speed: 1,
    };
    let mut balls: Vec<Ball> = vec![ball, ball1, ball2];
    loop {
        print_window(&balls, width, height);
        for b: &mut Ball in &mut balls {
            b.check_bounce(width, height);
            b.move_ball();
        }
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
} fn main
```

3. Structs - Methods

- Now that we've switched to a struct, we can easily extend our application by adding more balls!

```
        y: height / 4,
        x_speed: 1,
        y_speed: 1,
    };
    let mut balls: Vec<Ball> = vec![ball, ball1, ball2];
    loop {
        print_window(&balls, width, height);
        for b: &mut Ball in &mut balls {
            b.check_bounce(width, height);
            b.move_ball();
        }
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
} fn main
```

Declare balls above

3. Structs - Methods

- Now that we've switched to a struct, we can easily extend our application by adding more balls!

```
        y: height / 4,
        x_speed: 1,
        y_speed: 1,
    };
    let mut balls: Vec<Ball> = vec![ball, ball1, ball2];
    loop {
        print_window(&balls, width, height);
        for b: &mut Ball in &mut balls {
            b.check_bounce(width, height);
            b.move_ball();
        }
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
} fn main
```

Define list of balls

3. Structs - Methods

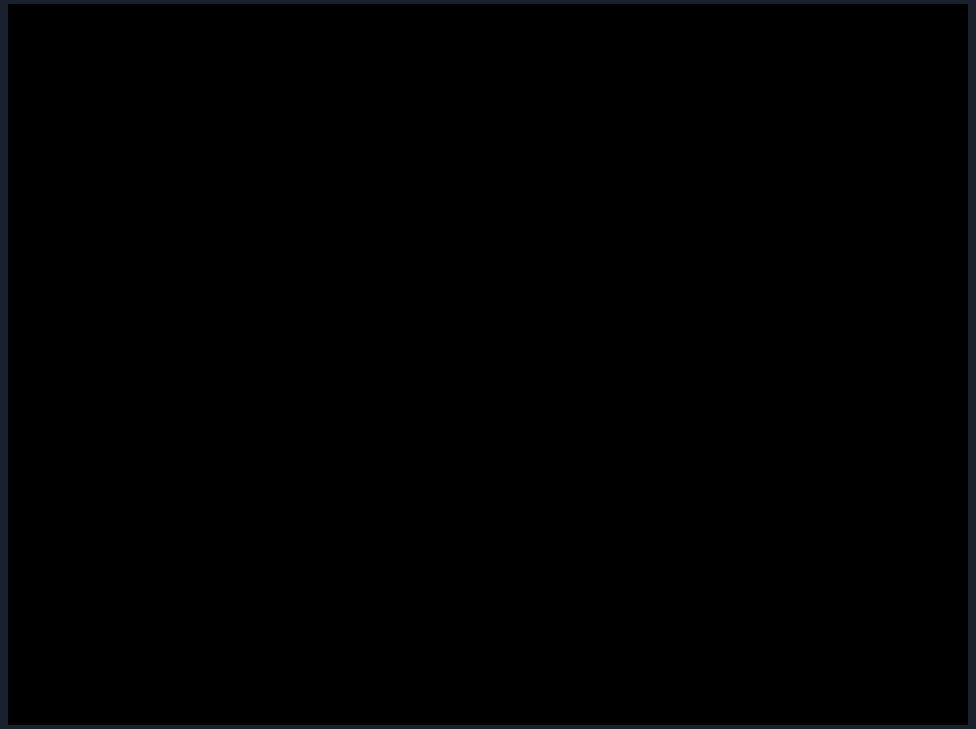
- Now that we've switched to a struct, we can easily extend our application by adding more balls!

```
        y: height / 4,
        x_speed: 1,
        y_speed: 1,
    };
    let mut balls: Vec<Ball> = vec![ball, ball1, ball2];
    loop {
        print_window(&balls, width, height);
        for b: &mut Ball in &mut balls {
            b.check_bounce(width, height);
            b.move_ball();
        }
        std::thread::sleep(dur: std::time::Duration::from_millis(1000 / 60)); // ~60 fps
    }
} fn main
```

Our logic doesn't change, except that we're now iterating over a list of balls :)

3. Structs - Methods

- Demo:



Intermission - Exercise

- Time for exercises!

Intermission - Exercise

1/3

```
struct Point {
    x: i32,
    y: i32,
    y: i32
}
pub fn main() {
    let p: Point = Point { x: 1, y: 2 };
    println!("p.x = {}", p.x);
    println!("p.y = {}", p.y);
}
```

Intermission - Exercise

1/3

```
struct Point {
    x: i32,
    y: i32,
    y: i32
}
pub fn main() {
    let p: Point = Point { x: 1, y: 2 };
    println!("p.x = {}", p.x);
    println!("p.y = {}", p.y);
}
```

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

Intermission - Exercise

1/3

```
struct Point {
    x: i32,
    y: i32,
    y: i32
}
pub fn main() {
    let p: Point = Point { x: 1, y: 2 };
    println!("p.x = {}", p.x);
    println!("p.y = {}", p.y);
}
```

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

Nope, fields must have unique names :)

Intermission - Exercise

2/3

```rust
struct Population {
    individuals: Vec<Individual>,
}
impl Population {
    fn get_average_age(&self) -> u32 {
        let mut sum: u32 = 0;
        for individual: &Individual in &self.individuals {
            sum += individual.age;
        }
        sum / self.individuals.len() as u32
    }
}
0 implementations
struct Individual {
    age: u32,
    height: u32,
}
pub fn main() {
    let bob: Individual = Individual { age: 32, height: 172 };
    let alice: Individual = Individual { age: 27, height: 160 };
    let mut population: Population = Population { individuals: vec![bob, alice] };
    population.individuals.push(Individual { age: 1, height: 50 });
    println!("Average age: {}", population.get_average_age());
}
```

Intermission - Exercise

2/3

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

```rust
struct Population {
    individuals: Vec<Individual>,
}
impl Population {
    fn get_average_age(&self) -> u32 {
        let mut sum: u32 = 0;
        for individual: &Individual in &self.individuals {
            sum += individual.age;
        }
        sum / self.individuals.len() as u32
    }
}
0 implementations
struct Individual {
    age: u32,
    height: u32,
}
pub fn main() {
    let bob: Individual = Individual { age: 32, height: 172 };
    let alice: Individual = Individual { age: 27, height: 160 };
    let mut population: Population = Population { individuals: vec![bob, alice] };
    population.individuals.push(Individual { age: 1, height: 50 });
    println!("Average age: {}", population.get_average_age());
}
```

Intermission - Exercise

2/3

```
struct Population {
    individuals: Vec<Individual>,
}
impl Population {
    fn get_average_age(&self) -> u32 {
        let mut sum: u32 = 0;
        for individual: &Individual in &self.individuals {
            sum += individual.age;
        }
        sum / self.individuals.len() as u32
    }
}
0 implementations
struct Individual {
    age: u32,
    height: u32,
}
pub fn main() {
    let bob: Individual = Individual { age: 32, height: 172 };
    let alice: Individual = Individual { age: 27, height: 160 };
    let mut population: Population = Population { individuals: vec![bob, alice] };
    population.individuals.push(Individual { age: 1, height: 50 });
    println!("Average age: {}", population.get_average_age());
}
```

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

It does compile! This code is valid!

Intermission - Exercise

2/3

```rust
struct Population {
    individuals: Vec<Individual>,
}
impl Population {
    fn get_average_age(&self) -> u32 {
        let mut sum: u32 = 0;
        for individual: &Individual in &self.individuals {
            sum += individual.age;
        }
        sum / self.individuals.len() as u32
    }
}
0 implementations
struct Individual {
    age: u32,
    height: u32,
}
pub fn main() {
    let bob: Individual = Individual { age: 32, height: 172 };
    let alice: Individual = Individual { age: 27, height: 160 };
    let mut population: Population = Population { individuals: vec![bob, alice] };
    population.individuals.push(Individual { age: 1, height: 50 });
    println!("Average age: {}", population.get_average_age());
}
```

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

It does compile! This code is valid!

You can omit the return keyword if the return value is the last expression in a function!
If you do that, you must also omit the semicolon.

Intermission - Exercise

2/3

```
struct Population {
    individuals: Vec<Individual>,
}
impl Population {
    fn get_average_age(&self) -> u32 {
        let mut sum: u32 = 0;
        for individual: &Individual in &self.individuals {
            sum += individual.age;
        }
        sum / self.individuals.len() as u32
    }
}
0 implementations
struct Individual {
    age: u32,
    height: u32,
}
pub fn main() {
    let bob: Individual = Individual { age: 32, height: 172 };
    let alice: Individual = Individual { age: 27, height: 160 };
    let mut population: Population = Population { individuals: vec![bob, alice] };
    population.individuals.push(Individual { age: 1, height: 50 });
    println!("Average age: {}", population.get_average_age());
}
```

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

It does compile! This code is valid!

After the marked location, there are 3 individuals in the population:
- Bob, aged 32
- Alice, aged 27
- Baby, aged 1

Intermission - Exercise

2/3

```rust
struct Population {
    individuals: Vec<Individual>,
}
impl Population {
    fn get_average_age(&self) -> u32 {
        let mut sum: u32 = 0;
        for individual: &Individual in &self.individuals {
            sum += individual.age;
        }
        sum / self.individuals.len() as u32
    }
}
0 implementations
struct Individual {
    age: u32,
    height: u32,
}
pub fn main() {
    let bob: Individual = Individual { age: 32, height: 172 };
    let alice: Individual = Individual { age: 27, height: 160 };
    let mut population: Population = Population { individuals: vec![bob, alice] };
    population.individuals.push(Individual { age: 1, height: 50 });
    println!("Average age: {}", population.get_average_age());
}
```

```
Average age: 20
```

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

It does compile! This code is valid!

After the marked location, there are 3 individuals in the population:
- Bob, aged 32
- Alice, aged 27
- Baby, aged 1

The average age is 32+27+1=60/3=20 :)

Intermission - Exercise

3/3

```rust
struct Population {
    individuals: Vec<Individual>,
}
impl Population {
    fn get_average_age(&self) -> u32 {
        let mut sum: u32 = 0;
        for individual: &Individual in &self.individuals {
            sum += individual.age;
        }
        sum / self.individuals.len() as u32
    }
}
0 implementations
struct Individual {
    age: u32,
    height: u32,
}
pub fn main() {
    let bob: Individual = Individual { age: 32, height: 172 };
    let alice: Individual = Individual { age: 27, height: 160 };
    let mut population: Population = Population { individuals: vec![bob, alice] };
    population.individuals.push(Individual { age: 1, height: 50 });
    println!("Average age: {}", population.get_average_age());
}
```

Do you see any potential problems with this code snippet?

Intermission - Exercise

3/3

```rust
struct Population {
    individuals: Vec<Individual>,
}
impl Population {
    fn get_average_age(&self) -> u32 {
        let mut sum: u32 = 0;
        for individual: &Individual in &self.individuals {
            sum += individual.age;
        }
        sum / self.individuals.len() as u32
    }
}
0 implementations
struct Individual {
    age: u32,
    height: u32,
}
pub fn main() {
    let bob: Individual = Individual { age: 32, height: 172 };
    let alice: Individual = Individual { age: 27, height: 160 };
    let mut population: Population = Population { individuals: vec![bob, alice] };
    population.individuals.push(Individual { age: 1, height: 50 });
    println!("Average age: {}", population.get_average_age());
}
```

Do you see any potential problems with this code snippet?

if the population is empty, we divide by 0

Intermission - Exercise

3/3

```rust
struct Population {
    individuals: Vec<Individual>,
}
impl Population {
    fn get_average_age(&self) -> u32 {
        let mut sum: u32 = 0;
        for individual: &Individual in &self.individuals {
            sum += individual.age;
        }
        sum / self.individuals.len() as u32
    }
}
0 implementations
struct Individual {
    age: u32,
    height: u32,
}
pub fn main() {
    let bob: Individual = Individual { age: 32, height: 172 };
    let alice: Individual = Individual { age: 27, height: 160 };
    let mut population: Population = Population { individuals: vec![bob, alice] };
    population.individuals.push(Individual { age: 1, height: 50 });
    println!("Average age: {}", population.get_average_age());
}
```

Do you see any potential problems with this code snippet?

Not using an associated function means we could set inhuman ages and heights

Intermission - Exercise

3/3

```rust
struct Population {
    individuals: Vec<Individual>,
}
impl Population {
    fn get_average_age(&self) -> u32 {
        let mut sum: u32 = 0;
        for individual: &Individual in &self.individuals {
            sum += individual.age;
        }
        sum / self.individuals.len() as u32
    }
}
0 implementations
struct Individual {
    age: u32,
    height: u32,
}
pub fn main() {
    let bob: Individual = Individual { age: 32, height: 172 };
    let alice: Individual = Individual { age: 27, height: 160 };
    let mut population: Population = Population { individuals: vec![bob, alice] };
    population.individuals.push(Individual { age: 1, height: 50 });
    println!("Average age: {}", population.get_average_age());
}
```

Do you see any potential problems with this code snippet?

We could add the same individual multiple times

Intermission - Exercise

1/3

```
0 implementations
struct A { child: D }
0 implementations
struct B { child: A }
0 implementations
struct C { child: B }
0 implementations
struct D { child: C }
```

Intermission - Exercise

1/3

```
0 implementations
struct A { child: D }
0 implementations
struct B { child: A }
0 implementations
struct C { child: B }
0 implementations
struct D { child: C }
```

Does this work?

Intermission - Exercise

1/3

```
0 implementations
struct A { child: D }
0 implementations
struct B { child: A }
0 implementations
struct C { child: B }
0 implementations
struct D { child: C }
```

Does this work?

Size of A = Size of D

Intermission - Exercise

1/3

```
0 implementations
struct A { child: D }
0 implementations
struct B { child: A }
0 implementations
struct C { child: B }
0 implementations
struct D { child: C }
```

Does this work?

Size of A = Size of D
Size of B = Size of A = Size of D

Intermission - Exercise

1/3

```
0 implementations
struct A { child: D }
0 implementations
struct B { child: A }
0 implementations
struct C { child: B }
0 implementations
struct D { child: C }
```

Does this work?

Size of A = Size of D
Size of B = Size of A = Size of D
Size of C = Size of B = Size of A = Size of D

Intermission - Exercise

1/3

```
0 implementations
struct A { child: D }
0 implementations
struct B { child: A }
0 implementations
struct C { child: B }
0 implementations
struct D { child: C }
```

Does this work?

Size of A = Size of D
Size of B = Size of A = Size of D
Size of C = Size of B = Size of A = Size of D
Size of D = Size of C = Size of B = Size of C = Size of D

Intermission - Exercise

1/3

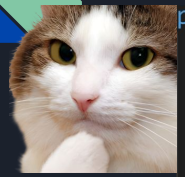
```
0 implementations
struct A { child: D }
0 implementations
struct B { child: A }
0 implementations
struct C { child: B }
0 implementations
struct D { child: C }
```

Does this work?

Size of A = Size of D
Size of B = Size of A = Size of D
Size of C = Size of B = Size of A = Size of D
Size of D = Size of C = Size of B = Size of C = Size of D

To calculate the size of D, we need the size of D!
Recursive! Infinite!

Intermission - Exercise



yeah...

```
pub fn main() {
    let d: D = D {
        child: C {
            child: B {
                child: A {
                    child: D {
                        child: C {
                            child: B {
                                child: A {
                                    child: D {
                                        child: C {
                                            child: B {
                                                child: A {
                                                    child: D {
                                                        child: C {
                                                            child: B {
                                                                child: A {
                                                                    child: D {
                                                                        child: C {
                                                                            child: B {
                                                                                child: A {
                                                                                    child: D
```

4. Next time

- Traits