RUSTikales Rust for beginners

Plan for today

Plan for today

1. Recap

Plan for today

1. Recap
2. Structs

Plan for today

1. Recap
2. Structs
3. Associated functions

Plan for today

1. Recap
2. Structs
3. Associated functions
4. Methods

1. Recap

1. Recap

- Ownership-Model
- References
- Borrow Checker

1. Recap

- Ownership-Model
- References
- Borrow Checker
- Functions are declared using the keyword fn
  - Functions can take in parameters
  - Functions can return values

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

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);
}
```

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);
}
```

Function call
→ Every parameter requires an argument

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);
}
```

Function call
→ Every parameter requires an argument
→ Returns a value which we can use

1. Recap

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

1. Recap

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

Rust does not support named arguments

1. Recap

```
fn no_overload() {}
fn no_overload(x: i32) {}
fn no_overload() -> i32 { 0 }
```

1. Recap

```rust
fn no_overload() {}
fn no_overload(x: i32) {}
fn no_overload() -> i32 { 0 }
```

Rust does not support function overloads

→ You can only have one function with any given name in a module

2. Structs

2. Structs

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

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    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;
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
} fn main
```

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    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;
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
} fn main
```

Normal loop, our application will never stop

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    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;
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
} fn main
```

Variables for the state of the application
→ Our world is 60x20 units big
→ Ball starts in the middle of the world
→ Ball moves to the bottom left

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    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;
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
} fn main
```

This function prints the current state to the console

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    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;
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
} fn main
```

Simple bouncing logic:
→ If we‘re at the edge of the world, invert the velocity
→ Ball moves in the opposite direction now
→ Breaks if the speed is not 1 :^)

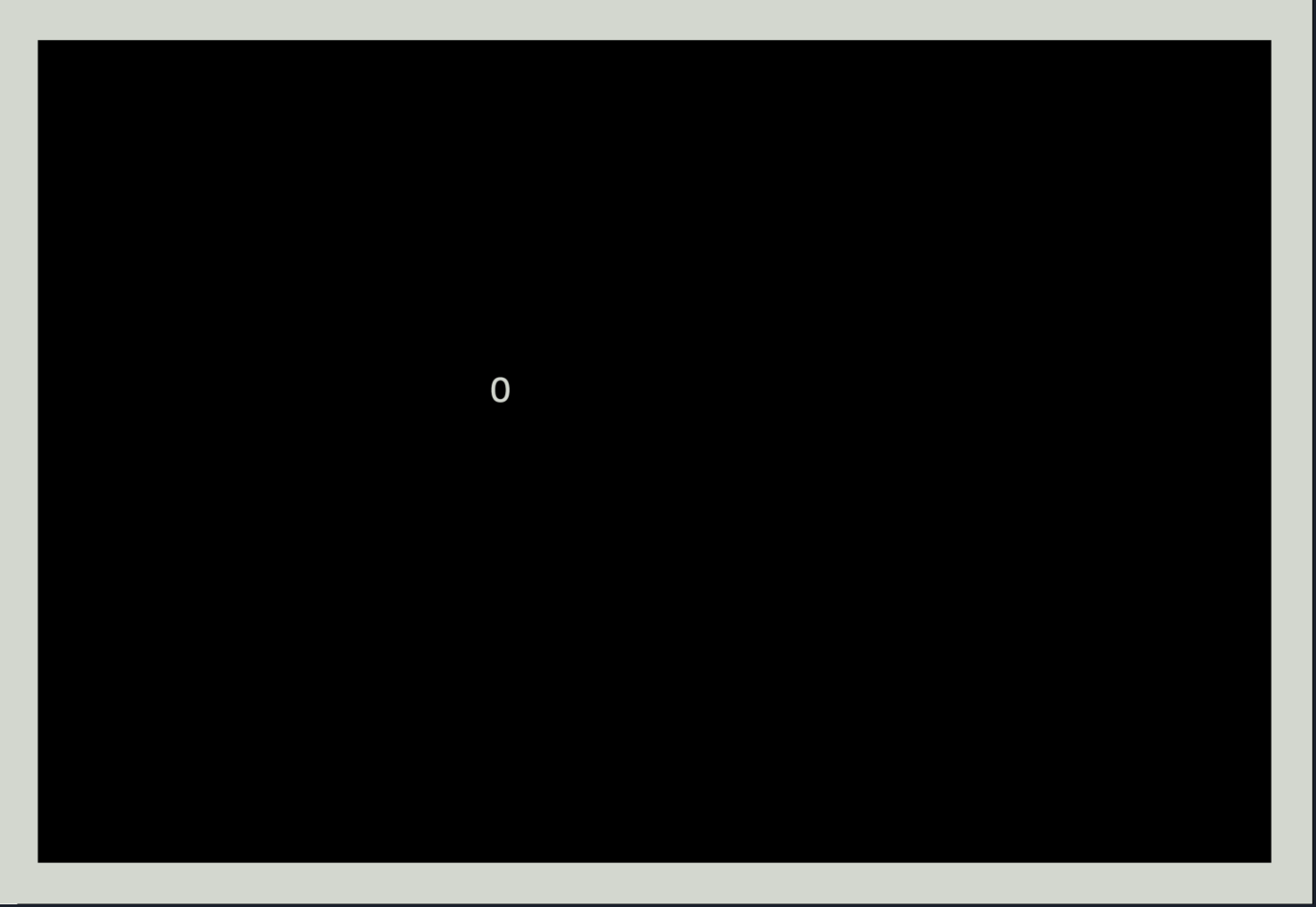
```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    let mut ball_x: i32 = width / 2;
    let mut ball_y: i32 = height / 2;
    let mut ball_x_speed: i32 = 1;
    let mut ball_y_speed: 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;
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
} fn main
```

Update the position of the ball

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    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;
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
} fn main
```

Rust is a fast language, we need to slow down our application so we can actually see the ball move

Our world looks like this:



Our world looks like this:

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    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;
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
} fn main
```

Notice how there is a lot of stuff to keep track of the ball state?

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    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;
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
} fn main
```

Notice how there is a lot of stuff to keep track of the ball state?

→ We need to initialize the ball

→ We need to update the ball

2. Structs

- Structs are collections of values

2. Structs

- Structs are collections of values
- Structs allow us to create more complex data structures than just i32 or bool

2. Structs

- Structs are collections of values
- Structs allow us to create more complex data structures than just i32 or bool
- Structs are declared using the keyword struct

2. Structs

```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);
}
```

2. Structs

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

Struct declaration

```rust
▶ 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);
}
```

2. Structs

```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);
}
```

Structs are made out of fields

2. Structs

```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);
}
```

Struct instantiation

2. Structs

```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);
}
```

Every field has to have a value

2. Structs

```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);
}
```

p1 and p2 are instances of type Point

2. Structs

```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);
}
```

Using the dot operator, we can access fields

2. Structs

```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);
}
```

```
p1.x = 1
p2.y = 4
```

2. Structs

```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);
}
```

```
p1.x = 1
p2.y = 4
```

2. Structs

```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);
}
```

```
p1.x = 1
p2.y = 4
```

2. Structs

```rust
struct Point {
    x: i32,
    y: i32,
}
0 implementations
struct Line {
    points: Vec<Point>,
}
fn create_line(start: &Point, end: &Point) -> Line {
    panic!("Too much effort to figure this out :^)")
}
▶ 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);
}
```

Structs are type definitions

2. Structs

```rust
struct Point {
    x: i32,
    y: i32,
}
0 implementations
struct Line {
    points: Vec<Point>,
}
fn create_line(start: &Point, end: &Point) -> Line {
    panic!("Too much effort to figure this out :^)")
}
▶ 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);
}
```

Structs are type definitions
→ Fields can be structs themselves

2. Structs

```rust
struct Point {
    x: i32,
    y: i32,
}
0 implementations
struct Line {
    points: Vec<Point>,
}
fn create_line(start: &Point, end: &Point) -> Line {
    panic!("Too much effort to figure this out :^)")
}
▶ 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);
}
```

Structs are type definitions
→ Fields can be structs themselves
→ We can pass structs as parameters

2. Structs

```rust
struct Point {
    x: i32,
    y: i32,
}
0 implementations
struct Line {
    points: Vec<Point>,
}
fn create_line(start: &Point, end: &Point) -> Line {
    panic!("Too much effort to figure this out :^)")
}
▶ 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);
}
```

Structs are type definitions
→ Fields can be structs themselves
→ We can pass structs as parameters
→ We can return structs from functions

2. Structs

```rust
struct Point {
    x: i32,
    y: i32,
}
0 implementations
struct Line {
    points: Vec<Point>,
}
fn create_line(start: &Point, end: &Point) -> Line {
    panic!("Too much effort to figure this out :^)")
}
▶ 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);
}
```

Structs are type definitions
→ Fields can be structs themselves
→ We can pass structs as parameters
→ We can return structs from functions
→ And, of course, variables can have struct types

2. Structs

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

2. Structs

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

Normal Ownership and Borrow Checker rules apply to structs

→ By default, struct instances are *always* moved

2. Structs

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

p1 and p2 are moved into the line, we can‘t use them down here anymore!

→ Using traits we can fix that, those will be covered next week

2. Structs

```
struct Tree {
    value: i32,
    left: Tree,
    right: Tree
}
```

Struct instances, like other variables, are located on the Stack
→ Size of a struct must be known at compile time

2. Structs

```
struct Tree {
    value: i32,
    left: Tree,
    right: Tree
}
```

Struct instances, like other variables, are located on the Stack
→ Size of a struct must be known at compile time
→ The size of a struct is the size of all its fields

2. Structs

```
struct Tree {
    value: i32,
    left: Tree,
    right: Tree
}
```

Struct instances, like other variables, are located on the Stack
→ Size of a struct must be known at compile time
→ The size of a struct is the size of all its fields
→ Tree is recursive: We need the size of Tree to get its size!
→ Compiler error

2. Structs

```
struct Tree {
    value: i32,
    left: Box<Tree>,
    right: Box<Tree>
}
```

Struct instances, like other variables, are located on the Stack
→ Size of a struct must be known at compile time
→ The size of a struct is the size of all its fields
→ Tree is recursive: We need the size of Tree to get its size!
→ Compiler error

Solution: Add indirection
→ Box moves the value to the heap and is always pointer-sized

2. Structs

```
struct Tree {
    value: i32,
    left: Box<Tree>,
    right: Box<Tree>
}
```

Struct instances, like other variables, are located on the Stack
→ Size of a struct must be known at compile time
→ The size of a struct is the size of all its fields
→ Tree is recursive: We need the size of Tree to get its size!
→ Compiler error

Solution: Add indirection
→ Box moves the value to the heap and is always pointer-sized
→ sizeof<Tree> = sizeof<i32> + 2 * sizeof<Box> bytes
→ sizeof<Tree> = 4 + 2 * 8 = 20 bytes

3. Associated functions

3. Associated functions

- Associated functions are functions associated with a type

3. Associated functions

- Associated functions are functions associated with a type
- Associated functions allow us to implement logic for types

3. Associated functions

- Associated functions are functions associated with a type
- Associated functions allow us to implement logic for types
- Associated functions are declared using the keyword impl

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
impl Line {
    fn new(start: &Point, end: &Point) -> Self {
        panic!("Still an unsolved problem :^)")
    }
}
```

▶ Run | Debug

```rust
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let line: Line = Line::new(start: &p1, end: &p2);
}
```

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

Every function we declare in here
gets associated with the type Point

```rust
impl Line {
    fn new(start: &Point, end: &Point) -> Self {
        panic!("Still an unsolved problem :^)")
    }
}

▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let line: Line = Line::new(start: &p1, end: &p2);
}
```

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
impl Line {
    fn new(start: &Point, end: &Point) -> Self {
        panic!("Still an unsolved problem :^)")
    }
}
```

Every function we declare in here
gets associated with the type Line

```rust
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let line: Line = Line::new(start: &p1, end: &p2);
}
```

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
impl Line {
    fn new(start: &Point, end: &Point) -> Self {
        panic!("Still an unsolved problem :^)")
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let line: Line = Line::new(start: &p1, end: &p2);
}
```

Self is a type alias for the type we're currently implementing

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
impl Line {
    fn new(start: &Point, end: &Point) -> Self {
        panic!("Still an unsolved problem :^)")
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let line: Line = Line::new(start: &p1, end: &p2);
}
```

Point

Point

Line

Self is a type alias for the type we're currently implementing

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
impl Line {
    fn new(start: &Point, end: &Point) -> Self {
        panic!("Still an unsolved problem :^)")
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let line: Line = Line::new(start: &p1, end: &p2);
}
```

In struct instantiations, if we assign the value of a variable to a field with the same name, we can save some typing

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
impl Line {
    fn new(start: &Point, end: &Point) -> Self {
        panic!("Still an unsolved problem :^)")
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let line: Line = Line::new(start: &p1, end: &p2);
}
```

This line is equivalent to
return Point { x: x, y: y };

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
impl Line {
    fn new(start: &Point, end: &Point) -> Self {
        panic!("Still an unsolved problem :^)")
    }
}
```

Because we‘re associating functions with a specific type, we can declare functions with the same name

→ Point::new(x, y) is different from Line::new(start, end)

→ Vec::new() is different from Point::new(x, y)

→ new() is different from Vec::new()

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
}
impl Line {
    fn new(start: &Point, end: &Point) -> Self {
        panic!("Still an unsolved problem :^)")
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point::new(x: 1, y: 2);
    let p2: Point = Point::new(x: 3, y: 4);
    let line: Line = Line::new(start: &p1, end: &p2);
}
```

When calling associated functions, we always have to specify the type

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(p1: &Point, p2: &Point) -> f64 {
        let dx: f64 = (p2.x - p1.x) as f64;
        let dy: f64 = (p2.y - p1.y) as f64;
        let xs: f64 = dx * dx;
        let ys: f64 = dy * dy;
        f64::sqrt(self: xs + ys)
    }
}
▶ 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!("distance = {}", d);
}
```

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(p1: &Point, p2: &Point) -> f64 {
        let dx: f64 = (p2.x - p1.x) as f64;
        let dy: f64 = (p2.y - p1.y) as f64;
        let xs: f64 = dx * dx;
        let ys: f64 = dy * dy;
        f64::sqrt(self: xs + ys)
    }
}
```

We can now implement special functions for our types

→ Here: Distance between two given points

```rust
▶ 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!("distance = {}", d);
}
```

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(p1: &Point, p2: &Point) -> f64 {
        let dx: f64 = (p2.x - p1.x) as f64;
        let dy: f64 = (p2.y - p1.y) as f64;
        let xs: f64 = dx * dx;
        let ys: f64 = dy * dy;
        f64::sqrt(self: xs + ys)
    }
}
▶ 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!("distance = {}", d);
}
```

The last expression in a function is also the return value
→ Writing return is optional

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(p1: &Point, p2: &Point) -> f64 {
        let dx: f64 = (p2.x - p1.x) as f64;
        let dy: f64 = (p2.y - p1.y) as f64;
        let xs: f64 = dx * dx;
        let ys: f64 = dy * dy;
        f64::sqrt(self: xs + ys)
    }
}
▶ 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!("distance = {}", d);
}
```

1/3

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

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(p1: &Point, p2: &Point) -> f64 {
        let dx: f64 = (p2.x - p1.x) as f64;
        let dy: f64 = (p2.y - p1.y) as f64;
        let xs: f64 = dx * dx;
        let ys: f64 = dy * dy;
        f64::sqrt(self: xs + ys)
    }
}
▶ 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!("distance = {}", d);
}
```

1/3

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

→ 4

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(p1: &Point, p2: &Point) -> f64 {
        let dx: f64 = (p2.x - p1.x) as f64;
        let dy: f64 = (p2.y - p1.y) as f64;
        let xs: f64 = dx * dx;
        let ys: f64 = dy * dy;
        f64::sqrt(self: xs + ys)
    }
}
▶ 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!("distance = {}", d);
}
```

Because we only borrow p1 and p2, we can still use them after this line

4. Methods

4. Methods

- Methods are a special kind of associated functions

4. Methods

- Methods are a special kind of associated functions
- Methods are all associated functions where the first parameter is either self, &self or &mut self

4. Methods

- Methods are a special kind of associated functions
- Methods are all associated functions where the first parameter is either self, &self or &mut self
- Methods allow us to easily call associated functions on struct instances

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(p1: &Point, p2: &Point) -> f64 {
        let dx: f64 = (p2.x - p1.x) as f64;
        let dy: f64 = (p2.y - p1.y) as f64;
        let xs: f64 = dx * dx;
        let ys: f64 = dy * dy;
        f64::sqrt(self: xs + ys)
    }
}
▶ 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!("distance = {}", d);
}
```

Currently an associated function
→ First parameter is not self

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(&self, p2: &Point) -> f64 {
        let dx: f64 = (p2.x - self.x) as f64;
        let dy: f64 = (p2.y - self.y) as f64;
        let xs: f64 = dx * dx;
        let ys: f64 = dy * dy;
        f64::sqrt(self: xs + ys)
    }
}
```

▶ Run | Debug

```rust
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!("distance = {}", d);
}
```

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(&self, p2: &Point) -> f64 {
        let dx: f64 = (p2.x - self.x) as f64;
        let dy: f64 = (p2.y - self.y) as f64;
        let xs: f64 = dx * dx;
        let ys: f64 = dy * dy;
        f64::sqrt(self: xs + ys)
    }
}
▶ 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!("distance = {}", d);
}
```

distance is now a method
→ First parameter is &self

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(&self, p2: &Point) -> f64 {
        let dx: f64 = (p2.x - self.x) as f64;
        let dy: f64 = (p2.y - self.y) as f64;
        let xs: f64 = dx * dx;
        let ys: f64 = dy * dy;
        f64::sqrt(self: xs + ys)
    }
}
▶ 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!("distance = {}", d);
}
```

self is now a specific instance of type Point

→ We don‘t need to provide the type, there‘s only one option what it can be

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(&self, p2: &Point) -> f64 {
        let dx: f64 = (p2.x - self.x) as f64;
        let dy: f64 = (p2.y - self.y) as f64;
        let xs: f64 = dx * dx;
        let ys: f64 = dy * dy;
        f64::sqrt(self: xs + ys)
    }
}
▶ 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!("distance = {}", d);
}
```

We can use the dot operator to call the method
→ p1 is passed as &self

```rust
impl Point {
    fn new(x: i32, y: i32) -> Self {
        return Self { x, y };
    }
    fn distance(&self, p2: &Point) -> f64 {
        let dx: f64 = (p2.x - self.x) as f64;
        let dy: f64 = (p2.y - self.y) as f64;
        let xs: f64 = dx * dx;
        let ys: f64 = dy * dy;
        f64::sqrt(self: xs + ys)
    }
}
```

We can use the dot operator to call the method
→ p1 is passed as &self
→ We now need to provide one less argument

▶ Run | Debug

```rust
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!("distance = {}", d);
}
```

```rust
    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);
    println!("before: {}", p1.x);
    p1.update(x: 3, y: 4);
    println!("after:  {}", p1.x);
}
```

```rust
    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);
    println!("before: {}", p1.x);
    p1.update(x: 3, y: 4);
    println!("after:  {}", p1.x);
}
```

Methods can also take mutable references to the instance

```rust
    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);
    println!("before: {}", p1.x);
    p1.update(x: 3, y: 4);
    println!("after:  {}", p1.x);
}
```

To call a mutable method, we need to have a mutable instance

```rust
    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);
    println!("before: {}", p1.x);
    p1.update(x: 3, y: 4);
    println!("after:  {}", p1.x);
}
```

Rust automatically borrows for us, we don‘t need to write (&mut p1).update(...)

```rust
    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);
    println!("before: {}", p1.x);
    p1.update(x: 3, y: 4);
    println!("after:  {}", p1.x);
}
```

```
before: 1
after:  3
```

```rust
    fn zero() -> Self {
        Self { x: 0, y: 0 }
    }
    fn x(mut self, x: i32) -> Self {
        self.x = x;
        self
    }
    fn y(mut self, y: i32) -> Self {
        self.y = y;
        self
    }
} impl Point
▶ Run | Debug
fn main() {
    let p1: Point = Point::zero().x(3).y(4);
    println!("p1.x = {}", p1.x);
    println!("p1.y = {}", p1.y);
}
```

```rust
    fn zero() -> Self {
        Self { x: 0, y: 0 }
    }
    fn x(mut self, x: i32) -> Self {
        self.x = x;
        self
    }
    fn y(mut self, y: i32) -> Self {
        self.y = y;
        self
    }
} impl Point
▶ Run | Debug
fn main() {
    let p1: Point = Point::zero().x(3).y(4);
    println!("p1.x = {}", p1.x);
    println!("p1.y = {}", p1.y);
}
```

Final version of methods:
→ Taking ownership of the instance

```rust
    fn zero() -> Self {
        Self { x: 0, y: 0 }
    }
    fn x(mut self, x: i32) -> Self {
        self.x = x;
        self
    }
    fn y(mut self, y: i32) -> Self {
        self.y = y;
        self
    }
} impl Point
▶ Run | Debug
fn main() {
    let p1: Point = Point::zero().x(3).y(4);
    println!("p1.x = {}", p1.x);
    println!("p1.y = {}", p1.y);
}
```

Commonly seen with the builder pattern

→ Allows us to define a default instance, and set specific fields

4. Methods

- Using our newfound knowledge, we can now use structs and methods for our ball application

4. Methods

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

4. Methods

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

ball position in the world

4. Methods

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

velocity

4. Methods

```rust
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;
    }
}
```

4. Methods

```
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;
    }
}
```

Usually you would not use a redundant name, but move is a keyword and can‘t be a function name :^)

→ Redundant because we‘re implementing the struct Ball

4. Methods

```
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;
    }
}
```

Check if our ball instance hit the wall of the given world

4. Methods

```
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;
    }
}
```

Move the ball in our world

4. Methods

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    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();
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
}
```

4. Methods

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    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();
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
}
```

We now collected every relevant value for the ball state into a single struct

4. Methods

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    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();
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
}
```

print_window() in the background was also updated to take an instance of type Ball

4. Methods

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    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();
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
}
```

Our main loop is now more concise and readable, we can easily see what is going on

4. Methods

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    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();
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
}
```

Additionally, because we isolated the ball logic, we can now easily create a whole list of balls :^)

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    let ball: Ball = Ball {…
    };
    let ball1: Ball = Ball {…
    };
    let ball2: Ball = Ball {…
    };
    let mut balls: Vec<Ball> = vec![ball, ball1, ball2];
    loop {
        print_window(&balls, width, height);
        for ball: &mut Ball in &mut balls {
            ball.check_bounce(width, height);
            ball.move_ball();
        }
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
} fn main
```

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    let ball: Ball = Ball {…
    };
    let ball1: Ball = Ball {…
    };
    let ball2: Ball = Ball {…
    };
    let mut balls: Vec<Ball> = vec![ball, ball1, ball2];
    loop {
        print_window(&balls, width, height);
        for ball: &mut Ball in &mut balls {
            ball.check_bounce(width, height);
            ball.move_ball();
        }
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
} fn main
```

→ We simply create more instances

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    let ball: Ball = Ball {…
    };
    let ball1: Ball = Ball {…
    };
    let ball2: Ball = Ball {…
    };
    let mut balls: Vec<Ball> = vec![ball, ball1, ball2];
    loop {
        print_window(&balls, width, height);
        for ball: &mut Ball in &mut balls {
            ball.check_bounce(width, height);
            ball.move_ball();
        }
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
} fn main
```

→ We simply create more instances

→ Add them to a list

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    let ball: Ball = Ball {…
    };
    let ball1: Ball = Ball {…
    };
    let ball2: Ball = Ball {…
    };
    let mut balls: Vec<Ball> = vec![ball, ball1, ball2];
    loop {
        print_window(&balls, width, height);
        for ball: &mut Ball in &mut balls {
            ball.check_bounce(width, height);
            ball.move_ball();
        }
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
} fn main
```

→ We simply create more instances
→ Add them to a list
→ And apply our logic to all of them!

```rust
pub fn main() {
    // Clear screen so we can freely use the cursor in print_window()
    print!("{}[2J", 27 as char);
    let width: i32 = 60;
    let height: i32 = 20;
    let ball: Ball = Ball {…
    };
    let ball1: Ball = Ball {…
    };
    let ball2: Ball = Ball {…
    };
    let mut balls: Vec<Ball> = vec![ball, ball1, ball2];
    loop {
        print_window(&balls, width, height);
        for ball: &mut Ball in &mut balls {
            ball.check_bounce(width, height);
            ball.move_ball();
        }
        // Pause our application to simulate 60 FPS
        sleep(dur: Duration::from_millis(1000 / 60));
    }
} fn main
```

→ We simply create more instances
→ Add them to a list
→ And apply our logic to all of them!
→ And don‘t forget to draw all :^)


0
0
0
And just like that, we have three balls :^)

Intermission - Exercises

- Time for exercises!

Intermission - Exercises

1/3

```rust
struct Point {
    x: i32,
    y: i32,
    y: u32
}
pub fn main() {
    let p: Point = Point { x: 1, y: 2 };
    println!("p.x = {}", p.x);
    println!("p.y = {}", p.y);
}
```

Intermission - Exercises

1/3

```rust
struct Point {
    x: i32,
    y: i32,
    y: u32
}
pub fn main() {
    let p: Point = Point { x: 1, y: 2 };
    println!("p.x = {}", p.x);
    println!("p.y = {}", p.y);
}
```

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

Intermission - Exercises

1/3

```rust
struct Point {
    x: i32,
    y: i32,
    y: u32
}
pub fn main() {
    let p: Point = Point { x: 1, y: 2 };
    println!("p.x = {}", p.x);
    println!("p.y = {}", p.y);
}
```

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

Nope, field names must be unique :)

```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

```rust
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 the 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());
}
```

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 the code compile?
If yes, what does it print?

The code does compile! This code is valid!

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 the code compile?
If yes, what does it print?

The code does compile! This code is valid!

Again, we can omit the return if the return value is the last expression in a function
→ Don‘t forget to also omit the semicolon!

```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 the code compile?
If yes, what does it print?

The code does compile! This code is valid!

After this line, there are three individuals in the population
→ Bob, aged 32
→ Alice, aged 27
→ Baby, aged 1

```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 the code compile?
If yes, what does it print?

The code does compile! This code is valid!

After this line, there are three individuals in the population
→ Bob, aged 32
→ Alice, aged 27
→ Baby, aged 1

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

```
Average age: 20
```

Do you see any potential problems with this code snippet?

```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?

```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());
}
```

If the population is empty, we divide by 0

Do you see any potential problems with this code snippet?

```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());
}
```

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

Do you see any potential problems with this code snippet?

```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());
}
```

We could add the same individual multiple times

Intermission - Exercises

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 - Exercises

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 - Exercises

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?

→ sizeof<A> = sizeof<D>

Intermission - Exercises

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?

→ sizeof<A> = sizeof<D>
→ sizeof<B> = sizeof<A> = sizeof<D>

Intermission - Exercises

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?

→ sizeof<A> = sizeof<D>
→ sizeof<B> = sizeof<A> = sizeof<D>
→ sizeof<C> = sizeof<B> = sizeof<A> = sizeof<D>

Intermission - Exercises

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?

→ sizeof<A> = sizeof<D>
→ sizeof<B> = sizeof<A> = sizeof<D>
→ sizeof<C> = sizeof<B> = sizeof<A> = sizeof<D>
→ sizeof<D> = sizeof<C> = sizeof<B> = sizeof<A> = sizeof<D>

Intermission - Exercises

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?

→ sizeof<A> = sizeof<D>
→ sizeof<B> = sizeof<A> = sizeof<D>
→ sizeof<C> = sizeof<B> = sizeof<A> = sizeof<D>
→ sizeof<D> = sizeof<C> = sizeof<B> = sizeof<A> = sizeof<D>

→ To calculate the size of D, we need the size of D
→ Recursive
→ Does not work

Intermission - Exercises

```rust
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:
```

Every field must have a value

→ We could never instantiate this struct anyway :(

5. Next time

- Traits