RUSTikales Rust for beginners

Plan for today

Plan for today

1. Recap

Plan for today

1. Recap
2. Function Calls

1. Recap

- Primitive Types in Rust: i8, u8, ..., i128, u128, bool, f32, f64

1. Recap

- Primitive Types in Rust: i8, u8, ..., i128, u128, bool, f32, f64
- Ownership-Model
- References
- Borrow Checker

1. Recap

- Primitive Types in Rust: i8, u8, ..., i128, u128, bool, f32, f64
- Ownership-Model
- References
- Borrow Checker
- Lifetimes
  - Every reference has a lifetime
  - Reference must be valid between creation and usage
  - Value it points to must be alive
  - Used by the Borrow Checker

1. Recap

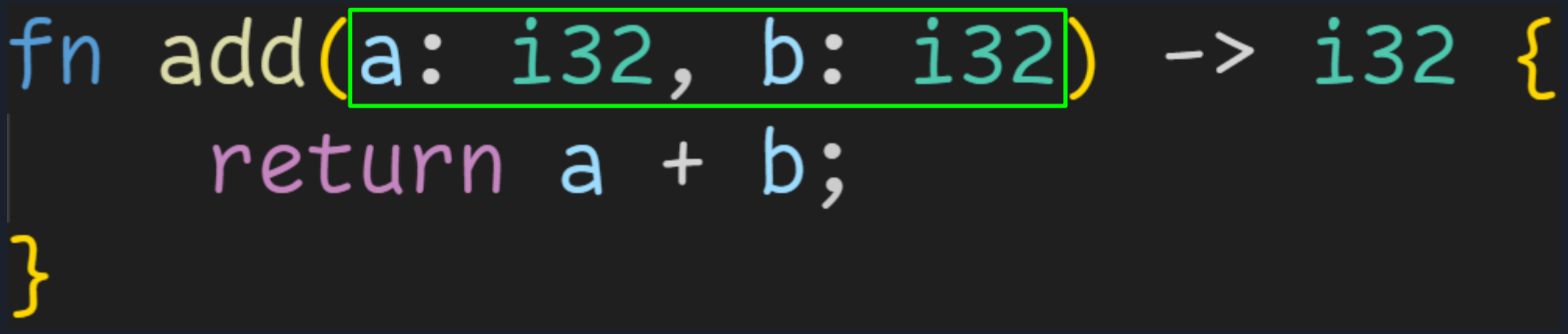
- Primitive Types in Rust: i8, u8, ..., i128, u128, bool, f32, f64
- Ownership-Model
- References
- Borrow Checker
- Lifetimes
- Functions are declared using the keyword fn
  - Functions can accept parameters
  - Functions can return values

1. Recap

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

1. Recap

Parameters



1. Recap

Return Type

```
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
```

1. Recap

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

return statement

→ Must return a value of the specified type

1. Recap

```rust
fn print_vec(vec: Vec<i32>) {
    println!("Vector: {:?}", vec);
}
fn create_vec() -> Vec<i32> {
    return vec![1, 2, 3];
}
```

1. Recap

Return Type is optional

```rust
fn print_vec(vec: Vec<i32>) {
    println!("Vector: {:?}", vec);
}
fn create_vec() -> Vec<i32> {
    return vec![1, 2, 3];
}
```

1. Recap

```rust
fn print_vec(vec: Vec<i32>) {
    println!("Vector: {:?}", vec);
}
fn create_vec() -> Vec<i32> {
    return vec![1, 2, 3];
}
```

Parameters are optional

1. Recap

1/3

```rust
fn add_vec(vec: Vec<i64>, num: i32) -> i64 {
    let mut prod: i64 = 1;
    for i: i64 in vec {
        prod *= i;
    }
    return prod + num;
}
```

1. Recap

1/3 Does this function compile?

```rust
fn add_vec(vec: Vec<i64>, num: i32) -> i64 {
    let mut prod: i64 = 1;
    for i: i64 in vec {
        prod *= i;
    }
    return prod + num;
}
```

1. Recap

1/3 Does this function compile?

```
fn add_vec(vec: Vec<i64>, num: i32) -> i64 {
    let mut prod: i64 = 1;
    for i: i64 in vec {
        prod *= i;
    }
    return prod + num;
}
```

prod is i64

1. Recap

1/3 Does this function compile?

```
fn add_vec(vec: Vec<i64>, num: i32) -> i64 {
    let mut prod: i64 = 1;
    for i: i64 in vec {
        prod *= i;
    }
    return prod + num;
}
```

prod is i64
num is i32

1. Recap

1/3 Does this function compile?

```
fn add_vec(vec: Vec<i64>, num: i32) -> i64 {
    let mut prod: i64 = 1;
    for i: i64 in vec {
        prod *= i;
    }
    return prod + num;
}
```

prod is i64
num is i32

→ i64 + i32 is not defined

2. Function Calls

2. Function Calls

- Function calls are expressions

2. Function Calls

- Function calls are expressions
- Functions are called by writing their name, followed by arguments in parenthesis

2. Function Calls

```rust
fn call_this() {
    println!("called call_this()!");
}
fn example1() {
    call_this();
}
```

2. Function Calls

```rust
fn call_this() {
    println!("called call_this()!");
}
```

Function declaration

```rust
fn example1() {
    call_this();
}
```

2. Function Calls

```rust
fn call_this() {
    println!("called call_this()!");
}
fn example1() {
    call_this();
}
```

Function call

2. Function Calls

```rust
fn call_this() {
    println!("called call_this()!");
}
fn example1() {
    call_this();
}
```

For every parameter, we have to provide an argument
→ call_this has no parameters, so don‘t need to provide anything

2. Function Calls

```rust
fn print_arg(vec: Vec<i32>) {
    println!("{:?}", vec);
}
fn example2() {
    let v: Vec<u8> = vec![1, 2, 3];
    print_arg(vec: v);
}
```

2. Function Calls

```rust
fn print_arg(vec: Vec<i32>) {
    println!("{:?}", vec);
}
fn example2() {
    let v: Vec<u8> = vec![1, 2, 3];
    print_arg(vec: v);
}
```

Every argument has to have the same type as the matching parameter

2. Function Calls

```rust
fn print_arg(vec: Vec<i32>) {
    println!("{:?}", vec);
}
fn example2() {
    let v: Vec<u8> = vec![1, 2, 3];
    print_arg(vec: v);
}
```

Type doesn‘t match
→ Compiler error

Every argument has to have the same type as the matching parameter

2. Function Calls

```rust
fn print_arg(vec: Vec<i32>) {
    println!("{:?}", vec);
}
fn example2() {
    let v: Vec<u8> = vec![1, 2, 3];
    print_arg(vec: v);
}
```

```
error[E0308]: mismatched types
  --> src\main.rs:14:15
   |
14 |     print_arg(v);
   |     --------- ^ expected `Vec<i32>`, found `Vec<u8>`
   |     |
   |     arguments to this function are incorrect
   |
   = note: expected struct `Vec<i32>`
              found struct `Vec<u8>`
note: function defined here
  --> src\main.rs:9:4
   |
9  | fn print_arg(vec: Vec<i32>) {
   |    ^^^^^^^^^ -------------
```

2. Function Calls

```rust
fn get_vec() -> Vec<i32> {
    return vec![1, 2, 3];
}
fn example3() {
    let result: Vec<i32> = get_vec();
    println!("{:?}", result);
}
```

2. Function Calls

```rust
fn get_vec() -> Vec<i32> {
    return vec![1, 2, 3];
}
fn example3() {
    let result: Vec<i32> = get_vec();
    println!("{:?}", result);
}
```

Return value can be used here

2. Function Calls

```rust
fn get_vec() -> Vec<i32> {
    return vec![1, 2, 3];
}
fn example3() {
    let result: Vec<i32> = get_vec();
    println!("{:?}", result);
}
```

result in example3 has this value after the function call

2. Function Calls

```rust
fn get_vec() -> Vec<i32> {
    return vec![1, 2, 3];
}
fn example3() {
    let result: Vec<i32> = get_vec();
    println!("{:?}", result);
}
```

result in example3 has this value after the function call

```
Running
[1, 2, 3]
```

2. Function Calls

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

2. Function Calls

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Function declared to take 2 parameters

2. Function Calls

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Function declared to take 2 parameters
→ Function takes 2 arguments

2. Function Calls

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Function declared to take 2 parameters
→ Function takes 2 arguments

Function declared to return i32

2. Function Calls

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Function declared to take 2 parameters
→ Function takes 2 arguments

Function declared to return i32
→ add(x, y) evaluates to an i32

2. Function Calls

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Function declared to take 2 parameters
→ Function takes 2 arguments

Function declared to return i32
→ add(x, y) evaluates to an i32
→ z is of type i32

2. Function Calls

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

```
z=31
```

Function declared to take 2 parameters
→ Function takes 2 arguments

Function declared to return i32
→ add(x, y) evaluates to an i32
→ z is of type i32

Code compiles, and prints 31

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

But how does this work?

How does a know the value of x?
How does b know the value of y?
How does z know the return value?

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

To understand this, we need to revisit memory

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

To understand this, we need to revisit memory
→ Instructions are stored in memory

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

To understand this, we need to revisit memory
→ Instructions are stored in memory
→ Whenever the computer sees a function, it jumps there, and executes that code

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

To understand this, we need to revisit memory
→ Instructions are stored in memory
→ Whenever the computer sees a function, it jumps there, and executes that code
→ Variables and parameters are stored on the stack

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

To understand this, we need to revisit memory
→ Instructions are stored in memory
→ Whenever the computer sees a function, it jumps there, and executes that code
→ Variables and parameters are stored on the stack
→ Every function allocates its own stack region, and frees it when you leave

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	???
y	???
z	???

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	14
y	???
z	???

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	14
y	17
z	???

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	14
y	17
z	???

We want to jump to add, but we need to pass the arguments.
But we can‘t assign to a and b yet, they don‘t exist in memory!

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	14
y	17
z	???

We want to jump to add, but we need to pass the arguments.
But we can‘t assign to a and b yet, they don‘t exist in memory!

Solution: Our computer has temporary registers

Register 1	???
Register 2	???

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	14
y	17
z	???

We want to jump to add, but we need to pass the arguments.
But we can't assign to a and b yet, they don't exist in memory!

Solution: Our computer has temporary registers

Register 1	14
Register 2	???

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	14
y	17
z	???

We want to jump to add, but we need to pass the arguments.
But we can't assign to a and b yet, they don't exist in memory!

Solution: Our computer has temporary registers

Register 1	14
Register 2	17

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

We can now call the function

Stack	
x	14
y	17
z	???

Register 1	14
Register 2	17

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	14
y	17
z	???
a	???
b	???

Register 1	14
Register 2	17

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	14
y	17
z	???
a	???
b	???

We can now move the arguments into the variables

Register 1	14
Register 2	17

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	14
y	17
z	???
a	14
b	???

We can now move the arguments into the variables

Register 1	14
Register 2	17

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	14
y	17
z	???
a	14
b	17

We can now move the arguments into the variables

Register 1	14
Register 2	17

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	14
y	17
z	???
a	14
b	17

We can now move the arguments into the variables
And execute the code in our function!

Register 1	14
Register 2	17

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

However, we don‘t know where we should store this result.
It may be z, it may be in a vector, we don‘t know!

Stack	
x	14
y	17
z	???
a	14
b	17

Register 1	14
Register 2	17

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	14
y	17
z	???
a	14
b	17

However, we don‘t know where we should store this result.
It may be z, it may be in a vector, we don‘t know!

Solution: Store it in a register, and leave it to the caller

Register 1	31
Register 2	17

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

We return now, which means cleaning the stack

Stack	
x	14
y	17
z	???

Register 1	31
Register 2	17

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	14
y	17
z	???

We now have the result in Register 1, which we can use

Register 1	31
Register 2	17

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	14
y	17
z	31

We now have the result in Register 1, which we can use
→ We move the value into z

Register 1	31
Register 2	17

Intermission - Control Flow, again

```rust
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn example4() {
    let x: i32 = 14;
    let y: i32 = 17;
    let z: i32 = add(a: x, b: y);
    println!("z={}", z);
}
```

Stack	
x	14
y	17
z	31

Our program prints 31

```
z=31
```

2. Function Calls

```rust
fn sub(a: i32, b: i32) -> i32 {
    return a - b;
}
fn example5() {
    if sub(a: 10, b: 5) == 3 {
        println!("Something is wrong!");
    }
}
```

2. Function Calls

```rust
fn sub(a: i32, b: i32) -> i32 {
    return a - b;
}
fn example5() {
    if sub(a: 10, b: 5) == 3 {
        println!("Something is wrong!");
    }
}
```

Functions are expressions, we can use them basically everywhere

2. Function Calls

```rust
fn print_number(n: i32) {
    println!("{}", n);
}
fn add_one(n: i32) -> i32 {
    return n + 1;
}
fn example6() {
    print_number(add_one(5));
}
```

2. Function Calls

```rust
fn print_number(n: i32) {
    println!("{}", n);
}
fn add_one(n: i32) -> i32 {
    return n + 1;
}
fn example6() {
    print_number(add_one(5));
}
```

Including as arguments themselves!

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

You can call a function in itself
→ Recursion

2. Function Calls

Stack

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	???
result	???

Registers	
Ret	???
Arg	???

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	10
result	???

Registers	
Ret	???
Arg	???

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	10
result	???

Registers	
Ret	???
Arg	10

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	10
result	???
n	10

Registers	
Ret	???
Arg	10

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if false {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	10
result	???
n	10

Registers	
Ret	???
Arg	10

2. Function Calls

```
fn factorial(n: i32) -> i32 {
    if false {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Need to evaluate this expression before we can return a value

Stack	
n	10
result	???
n	10

Registers	
Ret	???
Arg	10

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if false {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	10
result	???
n	10

Registers	
Ret	???
Arg	9

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	10
result	???
n	10
n	9

Registers	
Ret	???
Arg	9

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	10
result	???
n	10
n	9

Registers	
Ret	???
Arg	9

Every time we call a function, we allocate new stack space
→ Every n we see here is in a different memory location!
→ Every function can only directly access its own stack space

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if false {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	10
result	???
n	10
n	9

Registers	
Ret	???
Arg	8

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	10
result	???
n	10
n	9
n	8

Registers	
Ret	???
Arg	8

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if false {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	10
result	???
n	10
n	9
n	8

Registers	
Ret	???
Arg	7

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	10
result	???
n	10
n	9
n	8
n	7

Registers	
Ret	???
Arg	7

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Much later...

Stack	
n	10
result	???
n	10
n	9
n	8
n	7
n	6
n	5
n	4
n	3
n	2
n	1

Registers	
Ret	???
Arg	1

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

This is where the name Stack Overflow comes from:
→ Every time we call a recursive function, we allocate more memory
→ Memory is limited, we can easily allocate too much

Stack	
n	10
result	???
n	10
n	9
n	8
n	7
n	6
n	5
n	4
n	3
n	2
n	1

Registers	
Ret	???
Arg	1

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if true {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	10
result	???
n	10
n	9
n	8
n	7
n	6
n	5
n	4
n	3
n	2
n	1

Registers	
Ret	1
Arg	1

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

The computer keeps track of called functions

→ We can now go up the call chain and compute the result

Stack	
n	10
result	???
n	10
n	9
n	8
n	7
n	6
n	5
n	4
n	3
n	2

Registers	
Ret	1
Arg	1

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

The computer keeps track of called functions

→ We can now go up the call chain and compute the result

Stack	
n	10
result	???
n	10
n	9
n	8
n	7
n	6
n	5
n	4
n	3

Registers	
Ret	2
Arg	1

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

The computer keeps track of called functions

→ We can now go up the call chain and compute the result

Stack	
n	10
result	???
n	10
n	9
n	8
n	7
n	6
n	5
n	4

Registers	
Ret	6
Arg	1

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

The computer keeps track of called functions

→ We can now go up the call chain and compute the result

Stack	
n	10
result	???
n	10
n	9
n	8
n	7
n	6
n	5

Registers	
Ret	24
Arg	1

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Much later...

Stack	
n	10
result	???
n	10

Registers	
Ret	362880
Arg	1

The computer keeps track of called functions

→ We can now go up the call chain and compute the result

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	10
result	???

Registers	
Ret	3628800
Arg	1

We have now computed the result, which is located in the return register

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	10
result	3628800

Registers	
Ret	3628800
Arg	1

2. Function Calls

```rust
fn factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
fn example7() {
    let n: i32 = 10;
    let result: i32 = factorial(n);
    println!("{}! = {}", n, result);
}
```

Stack	
n	10
result	3628800

Registers	
Ret	3628800
Arg	1

```
10! = 3628800
```

2. Function Calls

```rust
fn print_vec(arg: Vec<i32>) {
    println!("{:?}", arg);
}
fn example8() {
    let v: Vec<i32> = vec![1, 2, 3];
    print_vec(arg: v);
    println!("{:?}", v);
}
```

2. Function Calls

```rust
fn print_vec(arg: Vec<i32>) {
    println!("{:?}", arg);
}
fn example8() {
    let v: Vec<i32> = vec![1, 2, 3];
    print_vec(arg: v);
    println!("{:?}", v);
}
```

Normal Ownership rules apply
→ This vector is moved!

2. Function Calls

Not a reference

```rust
fn print_vec(arg: Vec<i32>) {
    println!("{:?}", arg);
}
fn example8() {
    let v: Vec<i32> = vec![1, 2, 3];
    print_vec(arg: v);
    println!("{:?}", v);
}
```

2. Function Calls

```rust
fn print_vec(arg: &Vec<i32>) {
    println!("{:?}", arg);
}
fn example8() {
    let v: Vec<i32> = vec![1, 2, 3];
    print_vec(arg: &v);
    println!("{:?}", v);
}
```

By borrowing values, you can still use the vector after the function call

2. Function Calls

```rust
fn print_vec(arg: &Vec<i32>) {
    println!("{:?}", arg);
}
fn example8() {
    let v: Vec<i32> = vec![1, 2, 3];
    print_vec(arg: &v);
    println!("{:?}", v);
}
```

Lifetime of &v

2. Function Calls

```rust
fn modify_vec(arg: &mut Vec<i32>) {
    arg.push(1);
    if arg.len() < 10 {
        modify_vec(arg);
        arg.push(2);
    }
}
fn example9() {
    let mut vec: Vec<i32> = vec![1];
    modify_vec(arg: &mut vec);
    println!("{:?}", vec);
}
```

2. Function Calls

```rust
fn modify_vec(arg: &mut Vec<i32>) {
    arg.push(1);
    if arg.len() < 10 {
        modify_vec(arg);
        arg.push(2);
    }
}
fn example9() {
    let mut vec: Vec<i32> = vec![1];
    modify_vec(arg: &mut vec); We can also pass mutable references
    println!("{:?}", vec);
}
```

2. Function Calls

```rust
fn modify_vec(arg: &mut Vec<i32>) {
    arg.push(1);
    if arg.len() < 10 {
        modify_vec(arg);
        arg.push(2);
    }
}
fn example9() {
    let mut vec: Vec<i32> = vec![1];
    modify_vec(arg: &mut vec);
    println!("{:?}", vec);
}
```

Lifetime of arg

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

Returning references is only possible in special situations
→ This code will not compile

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

Addr	Stack	
0x4000	num	???

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

Addr	Stack	
0x4000	num	???

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

Addr	Stack	
0x4000	num	???
0x4004	number	???

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

Addr	Stack	
0x4000	num	???
0x4004	number	15

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

Register	0x4004

Addr	Stack	
0x4000	num	???
0x4004	number	15

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

Register	0x4004

Addr	Stack	
0x4000	num	0x4004

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

Register	0x4004

Addr	Stack	
0x4000	num	0x4004

The underlying value has been freed
→ Reference points to invalid memory!

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

Register	0x4004

Addr	Stack	
0x4000	num	0x4004

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

Register	0x4004

Addr	Stack	
0x4000	num	0x4004
0x4004	x	15

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

Register	0x4004

Addr	Stack	
0x4000	num	0x4004
0x4004	x	15

return_ref() just set this memory location to 15

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

Register	0x4004

Addr	Stack	
0x4000	num	0x4004
0x4004	x	12

return_ref() just set this memory location to 15
→ other_func overwrites the memory!

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

Register	0x4004

Addr	Stack	
0x4000	num	0x4004

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

Register	0x4004

Addr	Stack	
0x4000	num	0x4004

num still points to invalid memory!
We would now read a 12 instead of the 15 we wanted

2. Function Calls

```
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = return_ref();
    other_func();
}
```

Our code doesn‘t compile.

2. Function Calls

```rust
fn return_ref() -> &i32 {
    let number: i32 = 15;
    return &number;
}
fn other_func() {
    let x: i32 = 12;
}
fn example10() {
    let num: &i32 = retu
    other_func();
}
```

Our code doesn't compile.

```
error[E0106]: missing lifetime specifier
  --> src\main.rs:81:20
   |
81 | fn return_ref() -> &i32 {
   |                    ^ expected named lifetime parameter
   |
   = help: this function's return type contains a borrowed value, but there
is no value for it to be borrowed from
help: consider using the `'static` lifetime, but this is uncommon unless you
're returning a borrowed value from a `const` or a `static`
   |
81 | fn return_ref() -> &'static i32 {
   |                     +++++++
help: instead, you are more likely to want to return an owned value
   |
81 - fn return_ref() -> &i32 {
81 + fn return_ref() -> i32 {
   |
```

This error is out of the scope of this beginners course, but the idea is:
→ We can only return references to values that outlive the function

2. Function Calls

Error suggested providing a lifetime parameter, whatever that is

```
fn return_ref<'a>() -> &'a i32 {
    let number: i32 = 15;
    return &number;
}
```

2. Function Calls

Error suggested providing a lifetime parameter, whatever that is

```rust
fn return_ref<'a>() -> &'a i32 {
    let number: i32 = 15;
    return &number;
}
```

Error is easier to understand now :^) We can‘t outsmart the Borrow Checker

```
error[E0515]: cannot return reference to local variable `number`
  --> src\main.rs:83:12
   |
83 |     return &number;
   |            ^^^^^^^ returns a reference to data owned by the current function
```

2. Function Calls

```rust
fn element_of_vec(vec: &Vec<i32>) -> &i32 {
    return &vec[0]
}
fn example11() {
    let v: Vec<i32> = vec![10, 1, 2];
    let elem: &i32 = element_of_vec(&v);
    println!("{}", *elem);
}
```

2. Function Calls

```rust
fn element_of_vec(vec: &Vec<i32>) -> &i32 {
    return &vec[0]
}
fn example11() {
    let v: Vec<i32> = vec![10, 1, 2];
    let elem: &i32 = element_of_vec(&v);
    println!("{}", *elem);
}
```

We *can* return a reference here

2. Function Calls

```rust
fn element_of_vec(vec: &Vec<i32>) -> &i32 {
    return &vec[0]
}
fn example11() {
    let v: Vec<i32> = vec![10, 1, 2];
    let elem: &i32 = element_of_vec(&v);
    println!("{}", *elem);
}
```

Lifetime elision: The return reference has the same lifetime as the parameter

2. Function Calls

```rust
fn element_of_vec(vec: &Vec<i32>) -> &i32 {
    return &vec[0]
}
fn example11() {
    let v: Vec<i32> = vec![10, 1, 2];
    let elem: &i32 = element_of_vec(&v);
    println!("{}", *elem);
}
```

elem points to the first element of v

2. Function Calls

```rust
fn element_of_vec(vec: &Vec<i32>) -> &i32 {
    return &vec[0]
}
fn example11() {
    let v: Vec<i32> = vec![10, 1, 2];
    let elem: &i32 = element_of_vec(&v);
    println!("{}", *elem);
}
```

```
10
```

elem points to the first element of v

Intermission - Exercises

- Time for exercises!

Intermission - Exercises

1/3

```rust
fn add_one_ref(a: &mut i32) {
    *a = *a + 1;
}
▶ Run | Debug
fn main() {
    let mut c: i32 = 1;
    add_one_ref(&mut c);
    println!("c={}", c);
}
```

Intermission - Exercises

1/3

```rust
fn add_one_ref(a: &mut i32) {
    *a = *a + 1;
}
▶ Run | Debug
fn main() {
    let mut c: i32 = 1;
    add_one_ref(&mut c);
    println!("c={}", c);
}
```

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

Intermission - Exercises

1/3

```rust
fn add_one_ref(a: &mut i32) {
    *a = *a + 1;
}
▶ Run | Debug
fn main() {
    let mut c: i32 = 1;
    add_one_ref(&mut c);
    println!("c={}", c);
}
```

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

Yes, it does compile!

Intermission - Exercises

1/3

```rust
fn add_one_ref(a: &mut i32) {
    *a = *a + 1;
}

▶ Run | Debug
fn main() {
    let mut c: i32 = 1;
    add_one_ref(&mut c);
    println!("c={}", c);
}
```

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

Yes, it does compile!

This line modifies the original c

Intermission - Exercises

1/3

```rust
fn add_one_ref(a: &mut i32) {
    *a = *a + 1;
}

▶ Run | Debug
fn main() {
    let mut c: i32 = 1;
    add_one_ref(&mut c);
    println!("c={}", c);
}
```

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

Yes, it does compile!

This line modifies the original c

c=2

Intermission - Exercises

2/3

```rust
fn contains(vec: &Vec<i32>, value: i32) -> bool {
    for i: &i32 in vec {
        if i == value {
            return true;
        }
    }
    return false;
}
▶ Run | Debug
fn main() {
    let v: Vec<i32> = vec![1, 2, 3];
    let res: bool = contains(vec: &v, value: 2);
    println!("res={}", res);
}
```

Intermission - Exercises

2/3

```rust
fn contains(vec: &Vec<i32>, value: i32) -> bool {
    for i: &i32 in vec {
        if i == value {
            return true;
        }
    }
    return false;
}
▶ Run | Debug
fn main() {
    let v: Vec<i32> = vec![1, 2, 3];
    let res: bool = contains(vec: &v, value: 2);
    println!("res={}", res);
}
```

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

Intermission - Exercises

2/3

```rust
fn contains(vec: &Vec<i32>, value: i32) -> bool {
    for i: &i32 in vec {
        if i == value {
            return true;
        }
    }
    return false;
}
▶ Run | Debug
fn main() {
    let v: Vec<i32> = vec![1, 2, 3];
    let res: bool = contains(vec: &v, value: 2);
    println!("res={}", res);
}
```

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

This code is almost perfect, there are no errors in the arguments or return values

Intermission - Exercises

2/3

```rust
fn contains(vec: &Vec<i32>, value: i32) -> bool {
    for i: &i32 in vec {
        if i == value {
            return true;
        }
    }
    return false;
}
▶ Run | Debug
fn main() {
    let v: Vec<i32> = vec![1, 2, 3];
    let res: bool = contains(vec: &v, value: 2);
    println!("res={}", res);
}
```

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

BUT: &i32 == i32 is not defined :^)
The correct implementation would‘ve been *i == value

This code is almost perfect, there are no errors in the arguments or return values

Intermission - Exercises

2/3

```rust
fn contains(vec: &Vec<i32>, value: i32) -> bool {
    for i: &i32 in vec {
        if i == value {
            return true;
        }
    }
    return false;
}
▶ Run | Debug
fn main() {
    let v: Vec<i32> = vec![1, 2, 3];
    let res: bool = contains(vec: &v, value: 2);
    println!("res={}", res);
}
```

```
can't compare `&i32` with `i32`
```

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

Intermission - Exercises

1/3

```rust
fn fib(n: u32) -> u32 {
    if n == 0 { return 0; }
    if n == 1 { return 1; }
    return fib(n - 1) + fib(n - 2);
}

▶ Run | Debug
fn main() {
    let n: u32 = 7;
    let result: u32 = fib(n);
    println!("fib({})={}", n, result);
}
```

Intermission - Exercises

1/3

```rust
fn fib(n: u32) -> u32 {
    if n == 0 { return 0; }
    if n == 1 { return 1; }
    return fib(n - 1) + fib(n - 2);
}

▶ Run | Debug
fn main() {
    let n: u32 = 7;
    let result: u32 = fib(n);
    println!("fib({})={}", n, result);
}
```

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

Intermission - Exercises

1/3

```rust
fn fib(n: u32) -> u32 {
    if n == 0 { return 0; }
    if n == 1 { return 1; }
    return fib(n - 1) + fib(n - 2);
}

▶ Run | Debug
fn main() {
    let n: u32 = 7;
    let result: u32 = fib(n);
    println!("fib({})={}", n, result);
}
```

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

This code is very inefficient, but it gets the job done!
It compiles!

Intermission - Exercises

1/3

```rust
fn fib(n: u32) -> u32 {
    if n == 0 { return 0; }
    if n == 1 { return 1; }
    return fib(n - 1) + fib(n - 2);
}

▶ Run | Debug
fn main() {
    let n: u32 = 7;
    let result: u32 = fib(n);
    println!("fib({})={}", n, result);
}
```

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

This code is very inefficient, but it gets the job done!
It compiles!

It successfully calculates fibonacci numbers, and prints:

```
fib(7)=13
```

3. Next time

- Structs
- Methods