Rust Pre-session

RUST PROGRAMMING LANGUAGE

TOPIC

1. Computer Programming Basic
2. Basic Syntax
3. Variable and Data Types
4. Operators
5. If-Else

Computer Programming Basic

Programming Language is the notation used for specifying the instruction to the computer and designed to be human readable

However, the computer doesn’t directly understand the programming language that we write (i.e., Java, C, C++, and Rust). They are just files on the computer.

The computer can only execute machine code, which is difficult to understand, non-human-readable, and platform/architecture specific.

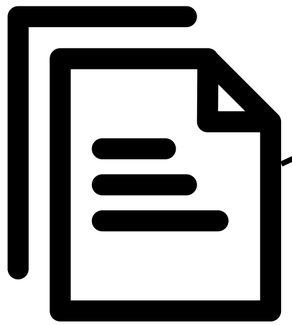
How Computer Execute our Code

Therefore, we need some tools to convert those programming language source codes into something that the computer undestands.

We call those tools the **Compiler.**

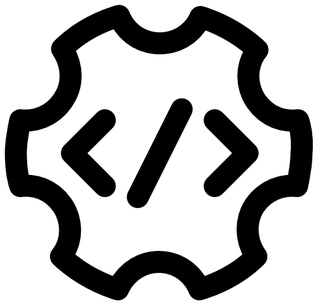
Rust Programming Language needs a compiler in order for computer to run the code

There's also an **Interpreter, which is another way the computer executes the code.*



The Source Code

We write our code into a file on the computer



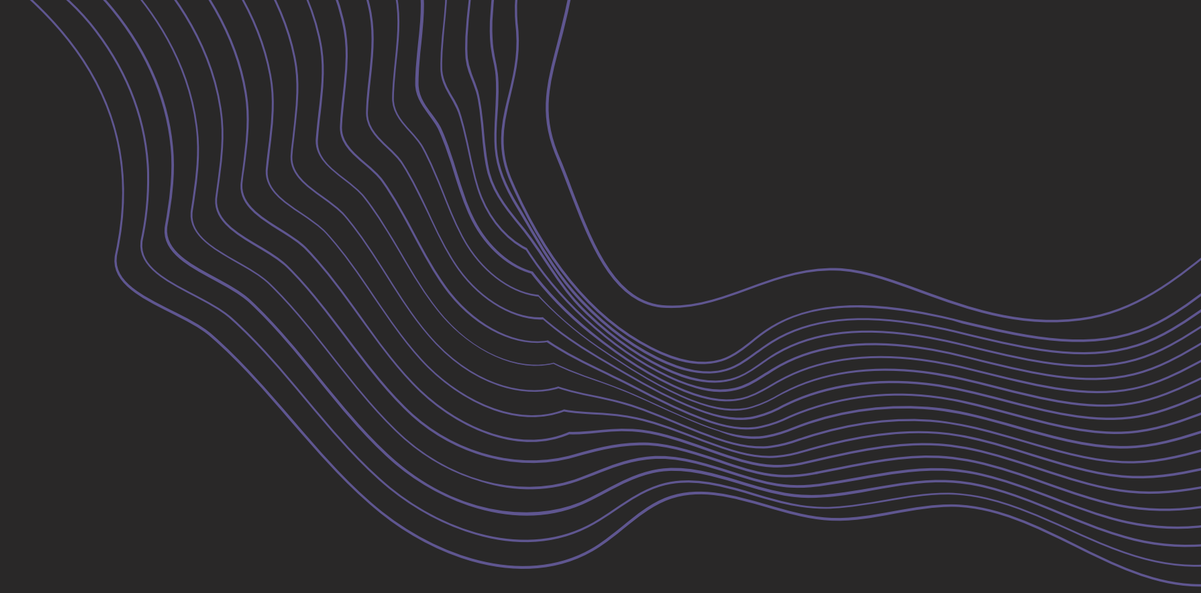
The Compiler

The compiler analyzes the source code we write, check for errors, and output it into machine code.

The Output

The format that the computer can execute code

Basic Syntax of Rust



Statements and println

This is called a statement. The statement is an instruction the computer will execute, always ending with a semi-colon.

```
fn main() {
    println!("Hello World");
}
```

This statement prints the word ***Hello World!*** to the console.

You can edit the word you want to print inside the quotation marks.

don’t forget semi-colon :)

Statements

```
fn main() {
    println!("Hello World!");
    println!("Hello SE16!");
}
```

Every statement inside the curly brace will be executed.

The statements are executed line-by-line, from top to bottom. Therefore, the word ***Hello World!*** will be printed first, followed by ***Hello SE16!***

```
Hello World!
Hello SE16!
```

Syntax Error

```
fn main() {
    println!("Hello World!")
    println!(Hello SE16!)
}
```

ERROR: No semi-colon

ERROR: Must be enclosed with quotation marks.

**This code will be rejected by the Rust compiler !*

println vs print

```
fn main() {
    println!("Hello World!");
    print!("race");
    print!("car");
}
```

println! will enter a new line after done printing the word

while ***print!*** does not

```
Hello World!
racecar
```

Comment

When inserting double slashes, the rest of the code following them will be ignored by the compiler. This is beneficial for us programmers for taking down some notes directly on the source code

```
fn main() {
    // take some note
    print!("race"); // this will print "race"
    print!("car");
    // println!("Hello World");
}
```

Therefore, this println!(“Hello World”); statement won’t be executed.

```
racecar
```

Comment

```
fn main() {
    /*comment*/ print!("hello");
    /*i've got something to say ..
    hi SE16!*/
}
```

This is another way of writing comments.

Starting with the **/*** symbol and ending with ***/**, all the code in between is ignored.

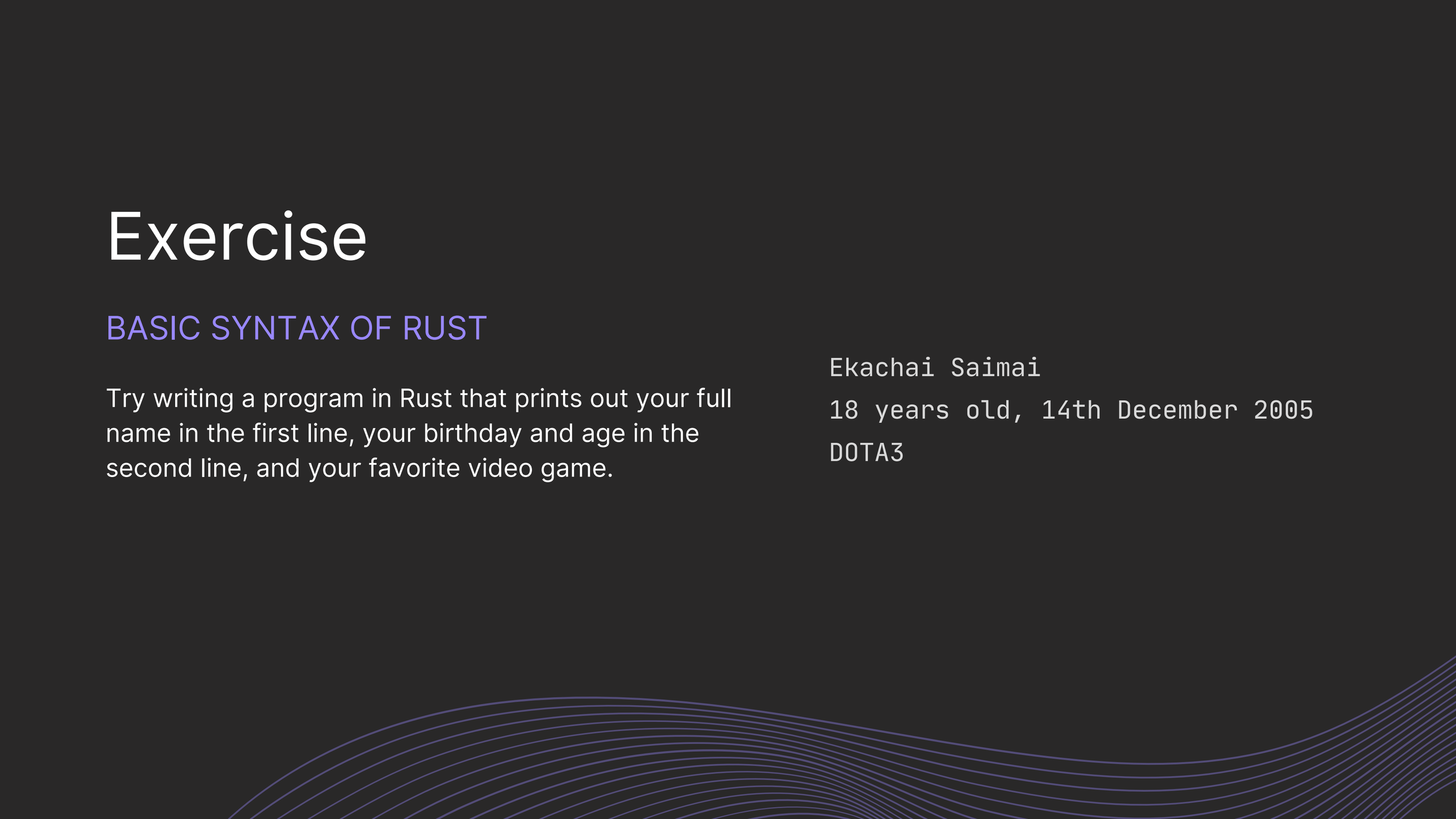
It can span across multiple lines.

Exercise

BASIC SYNTAX OF RUST

Try writing a program in Rust that prints out your full name in the first line, your birthday and age in the second line, and your favorite video game.

```
Ekachai Saimai
18 years old, 14th December 2005
DOTA3
```



Variable and Data Type

Variable

This is a **Variable Declaration** statement.

A variable is a place in memory that can store the value.

Think of it as a file in the computer where you can save some information and load it later.

```
fn main() {
    let number = 23;
}
```

Start with the ***keyword let***

Give the variable a name; It can be any name you want (with some exceptions).

Give the variable value to store; in this case, an integer 23.

Load the Variable

```
fn main() {
    let x = 23;
    let y = x + 19;
    println!("the variable y is {y}");
}
```

The variable ***x*** is loaded here, which results in integer 23.

Then add the integer 19 to 23, resulting in integer 42.

To print out the value stored in the variable, surround the variable name with a curly brace. It will be replaced with the value stored in the variable

```
the variable y is 42
```

Variable

```
fn main() {
    let x = 23;
    let y = who_am_i + 19;
    println!("the variable y is {y}");
}
```

This will cause a compilation error since we've never declared a variable named ***who_am_i.***

**The code will be rejected by the Rust compiler*

Variable

```
fn main() {
    let x = 23;
    let y = who_am_i + 19;
    let who_am_i = 32;
    println!("the variable y is {y}");
}
```

This still causes a compilation error since the variable ***who_am_i*** must be declared before using it.

**The code will be rejected by the Rust compiler*

This naming convention is called the ***Snake* Case**. The words are separated by underscore. This is the preferred naming convention for variable declaration.

```
fn main() {
    let one_plus_one = 2;
    let twoPlusOne = 3;
    let ThreePlusOne = 4;
    let four plus one = 5;
}
```

This is called the **Camel Case,** where the first character of the word is capitalized except the first one.

This is called the **Pascal Case**, similar to the **Camel Case.**

This is **NOT** allowed. The name can not have spaces. The compiler will reject this code!

**You should always use Snake Case for the variable name. Choosing any other naming convention doesn't necessarily cause an error, but we highly suggest choosing Snake Case for the consistency of the code.*

```
fn main() {
    let age = 17;
    age = 18;
    println!("I'm {age} years old");
}
```

You can update the value of the already created variable by using assignment.

The syntax is similar to the variable declaration except that it doesn't start with the **keyword let**.

**However, this code doesn't compile... Why?*

```
fn main() {
    let mut age = 17;
    age = 18;
    println!("I'm {age} years old");
}
```

The keyword ***mut*** must be inserted before the variable name so that it can be updated later.

So if you want to update the value of a particular variable, you must insert the keyword ***mut*** to it.

The word mut is shortened from the word mutable.

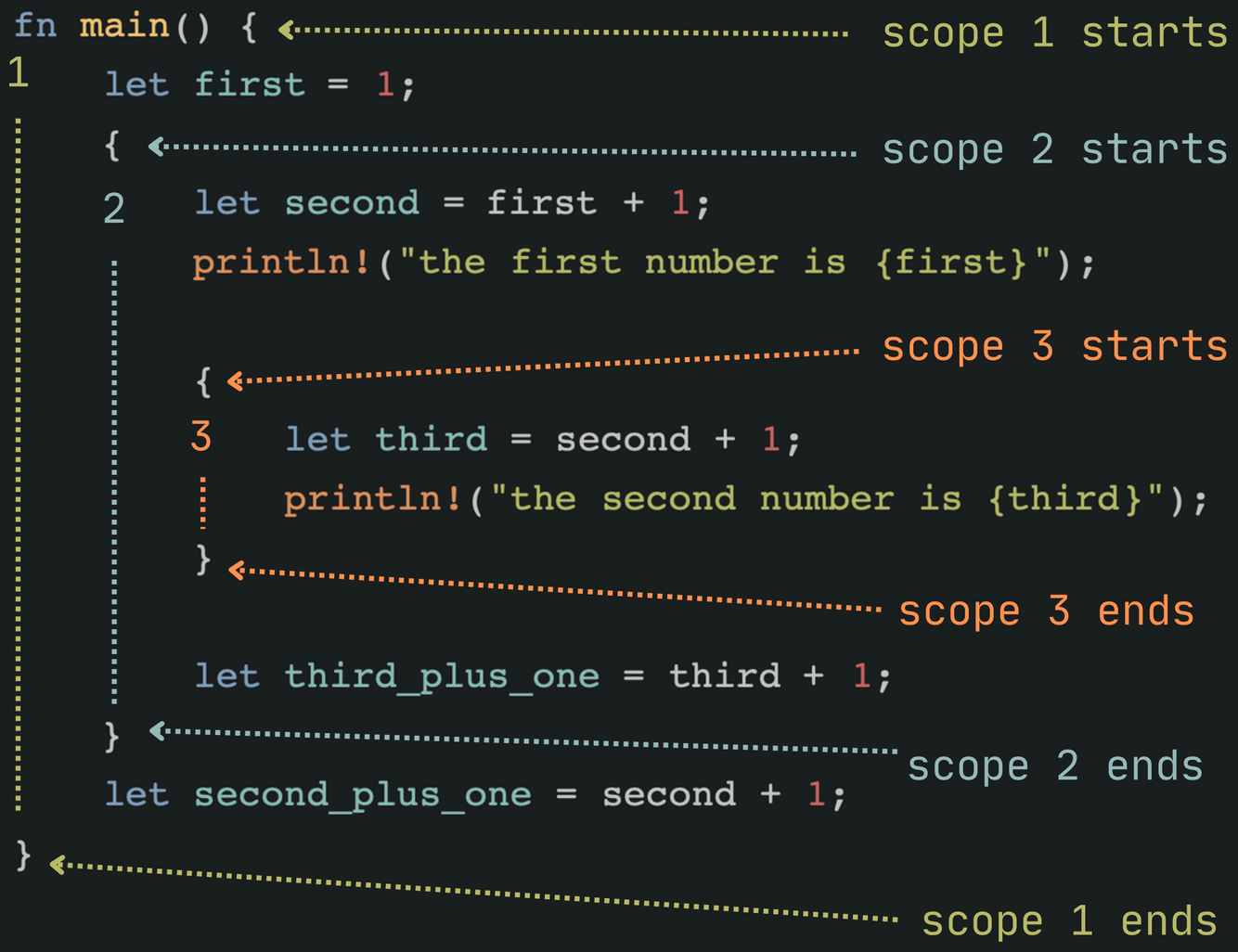
**The code now compiles*

```
I'm 18 years old
```

Scope

```
fn main() {
1   let first = 1;
    {
    2   let second = first + 1;
        println!("the first number is {first}");

        {
        3   let third = second + 1;
            println!("the second number is {third}");
        }

        let third_plus_one = third + 1;
    }
    let second_plus_one = second + 1;
}
```

Scope



The ***scope*** in Rust starts with opening curly-brace `{` and ends with closing curly-brace `}`. It can contain multiple statements in it.

Every time you see a curly-brace in Rust, it must be paired, and most of the time you can expect it to be a ***scope***.

The scope can be declared anywhere.

Scope

```
fn main() {
1   let first = 1;
    {
    2   let second = first + 1;
        println!("the first number is {first}");

        {
        3   let third = second + 1;
            println!("the second number is {third}");
        }

        let third_plus_one = third + 1;
    }
    let second_plus_one = second + 1;
}
```

Scope 1 is the biggest scope.

Scope 2 is declared on top of the Scope 1.

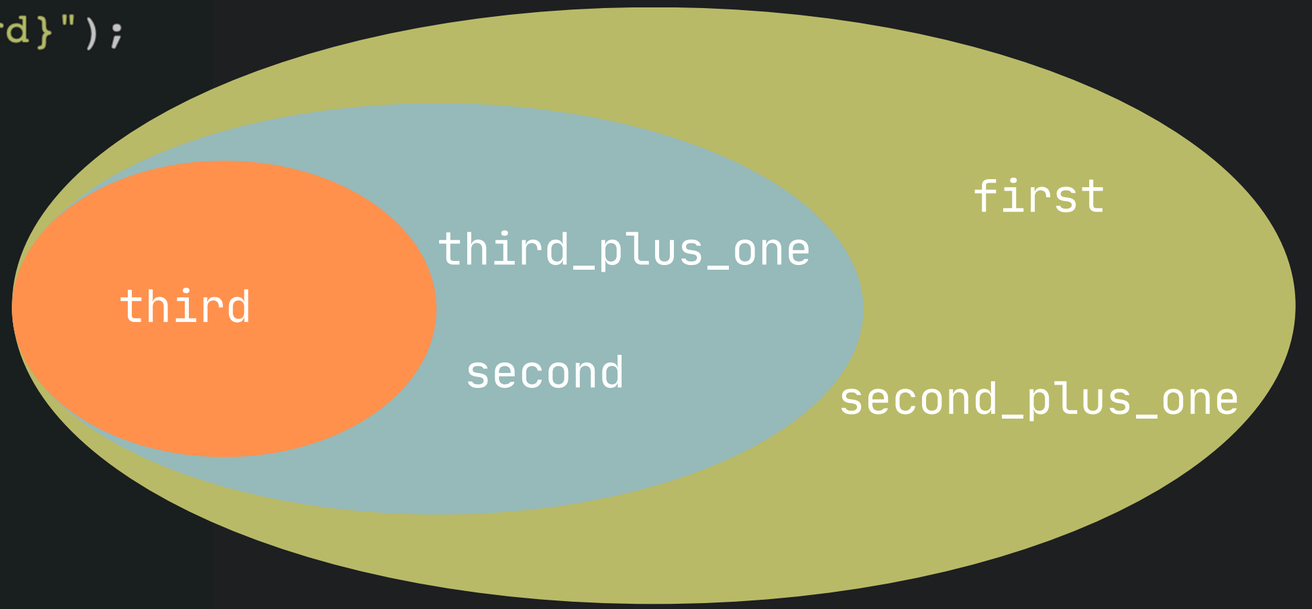
Scope 3 is declared on top of the Scope 2 and Scope 1

Scope

```
fn main() {
1   let first = 1;
    {
    2   let second = first + 1;
        println!("the first number is {first}");

        {
        3   let third = second + 1;
            println!("the second number is {third}");
        }

        let third_plus_one = third + 1;
    }
    let second_plus_one = second + 1;
}
```

Every variable in Rust is tied to the Scope where it's declared.

Therefore, the variable ***first*** is tied to scope number 1; the ***second*** is tied to scope 2; the ***third*** is tied to scope 3; and so on.



Scope

```
fn main() {
1   let first = 1;
    {
    2   let second = first + 1;
        println!("the first number is {first}");

        {
        3   let third = second + 1;
            println!("the second number is {third}");
        }

        let third_plus_one = third + 1;
    }
    let second_plus_one = second + 1;
}
```

When the scope ends, all the variables tied to it (declared in it) will be destroyed, and you can no longer access them.

Scope 3 ends, and the variable `third` is destroyed

Scope 2 ends, and the variable `second` and `third_plus_one` are destroyed

Scope 1 ends, and the variable `first` and `second_plus_one` are destroyed

Scope

```
fn main() {
1   let first = 1;
    {
    2   let second = first + 1;
        println!("the first number is {first}");

        {
        3   let third = second + 1;
            println!("the second number is {third}");
        }

        let third_plus_one = third + 1;
    }
    let second_plus_one = second + 1;
}
```

These two statements are referring two variables that are alive.

Therefore this is valid.

However, in these two statements, they are referring to those two variables that have already been destroyed.

Therefore, this code is **INVALID!**

**This code will be rejected by the Rust compiler !*

Exercise

VARIABLE

Try to spot an error found within this code and give the explanation of why.

Hint: there are 3 errors.

```
fn main() {
    let age = 32;

    {
        let year = 2024;
        println!("in {year} i'm {age} years old");
    }

    {
        println!("in {year} i'm studying in year {study_year}");
        let study_year = 1;
    }

    // 1 year later ...
    age = age + 1; // increase my age by 1

    println!("now i'm {age} years old");
}
```

More Hints ...

```
fn main() {
    let age = 32;

    {
        let year = 2024;
        println!("in {year} i'm {age} years old");
    }

    {
        println!("in {year} i'm studying in year {study_year}");
        let study_year = 1;
    }

    // 1 year later ...
    age = age + 1; // increase my age by 1

    println!("now i'm {age} years old");
}
```

Variable `***year***` is destroyed here

Variable `***study_year***` has just been declared here.

There's an update to the variable `***age***` but ...

Solution

```
fn main() {
    let age = 32;

    {
        let year = 2024;
        println!("in {year} i'm {age} years old");
    }

    {
        println!("in {year} i'm studying in year {study_year}");
        let study_year = 1;
    }

    // 1 year later ...
    age = age + 1; // increase my age by 1

    println!("now i'm {age} years old");
}
```

Variable `***year***` has already been destroyed.

Variable `***study_year***` is used before the declaration.

There’s an update to the variable `***age***` but the variable is not declared with `***mut***` keyword

So far you might have seen us using integers here such as integer 12 and some addition.

But can we put anything more than integers?

```
fn main() {
    let variable = /*what can it be?*/;
}
```

Yes, you can put various **Expressions** here.

Clarification: Expression vs Value

```
fn main() {
    let variable = 2 + 3;
}
```

The whole `***2+3***` thing here is called ***Expression***, specifically ***Binary Expression***.

This expression will be evaluated to the ***Value*** of **5**

Continue ...

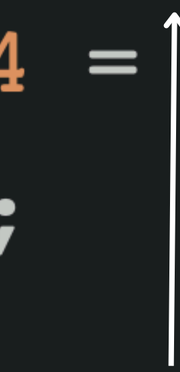
```
fn main() {
    let signed_integer: i32    = -52;
    let unsigned_integer: u32 = 32;
    let floating_point: f64    = 21.0;
    let boolean: bool          = true;
}
```

In this lecture, we will introduce you to 4 ***types*** of value, which are signed/unsigned integer, floating point, and boolean.

This is called type annotation. It's used for specifying the type of the variable. This is optional.

Signed Integer

```rust
fn main() {
    let first_signed_integer: i32 = -32;
    let second_signed_integer: i64 = 64;
    let third_signed_integer = 32;
}
```



Can be negative!

Signed Integers are integers (no decimal places) that **can be *negative***.

Rust supports multiple kinds of Signed Integer types:
`i8`, `i16`, `i32`, `i64`, and `i128`.

Signed Integer

```
fn main() {
    let first_signed_integer: i32 = -32;
    let second_signed_integer: i64 = 64;
    let third_signed_integer = 32;
}
```

Since the type annotation is optional and when it is not provided, the default ***type*** of ***integer*** (without decimal places) will be `i32`

Unsigned Integer

```rust
fn main() {
    let first_unsigned_integer: u32 = -2;
    let second_unsigned_integer: u16 = 3;
}
```

Unsigned Integers are similar to the ***Signed Integer*** except that they don't support negative numbers.

Rust supports multiple kinds of Unsigned Integer types: `u8`, `u16`, `u32`, `u64`, and `u128`.

This is **NOT ALLOWED!**

**This code will be rejected by the Rust compiler !*

Floating Point

This is **NOT ALLOWED.** This is an integer;
it must have decimal places followed

```
fn main() {
    let first_floating_point: f32 = 2;
    let second_floating_point: f64 = -1.5;
    let third_floating_point = 2.0;
}
```

Floating Point data types are numbers that can have decimal places and can be negative.

Can be a negative number!

Rust supports two kinds of floating point types: `f32` and `f64`.

**This code will be rejected by the Rust compiler !*

Floating Point

```rust
fn main() {
    let first_floating_point: f32 = 2;
    let second_floating_point: f64 = -1.5;
    let third_floating_point = 2.0;
}
```

The default type of the ***Floating Point*** is `f32`

Boolean

```
fn main() {
    let im_right = true;
    let im_wrong: bool = false;
}
```

Boolean data type has only two possible `true` and `false`.

Boolean might seem useless now, but we'll see the applications later in this lecture.

What’s up With the Number Following After the Type Name?

`u8`

`u32`

You might notice that after the type name of signed/unsigned integer and floating point type, there are a number with the power of two following after.

`i16`

`i64`

`f64`

`f32`

These numbers denote the ***size in bits*** of the type.

The larger the ***size*** the larger the range of value it can contain.

If you don’t know which size to choose, the sensible default is 32.

What's up With the Number Following After the Type Name?

u8 u32 i16 i64 f32 f64

For example, the type `u8` occupies ***8 bits*** in memory.
So it might look something like this:

Therefore, this unsigned integer type can only support the range of numbers
from 0 to 255 or (0 to 2^8 - 1), which is a total of 256 values.

This may seem too ***small*** for many applications but it also takes ***less space*** in memory

What’s up With the Number Following After the Type Name?

u8 u32

Next, the type `u32` occupies 32 ***bits*** in memory. i16

i64 So it might look something like this:

f32 `0110 1001 1101 0000 1101 0000 11011 0111` f64

Now, this type supports a wider range of values, starting from 0 to 4,294,967,295 or 0 to (2^32 - 1) -- a total of 4,294,967,296; which is significantly larger than the `u8`.

The range is much larger than the previous one but takes ***four times more space***.

Choosing the Right Type

u8 u32

Choosing the right type of integer is important as well. For example, the sensible integer type for storing the age of a person would be some unsigned integer type; since the negative age wouldn’t make any sense in real life.

i16 i64 f32 f64

u8 u32

Or when performing the scientific calculation that involves decimal places, choosing floating point data types would be the correct choice.

i16 i64 f32 f64

Type Inference

```
fn main() {
    let first = -32;
    let second = 64;
    let third = 32.0;
    let fourth = true;
    let age = 18;
}
```

Type annotation is optional (the `: i32` or `: bool` syntax thing), however, every variable does have a ***type even without type annotation.***

If the type annotation is not provided, the Rust compiler will ***infer*** (guess) the type by looking at the value assigned to it.

Therefore, sometimes you might need to annotate the variable type if the Rust compiler doesn't infer the type you desired.

Type Inference

```
fn main() {
    let first = -32;
    let second = 64;
    let third = 32.0;
    let fourth = true;
    let age = 18;
}
```

age should be an unsigned integer but it has been inferred as `i32`. What should be done here? ...

Type Inference

```
fn main() {
    let first = -32;
    let second = 64;
    let third = 32.0;
    let fourth = true;
    let age: u32 = 18;
}
```

Problem Fixed! The variable `age` now has the type of `u32` as we ***explicitly*** specified it with type annotation.

Strong Typing

```rust
fn main() {
    let mut im_boolean = true;
    im_boolean = false;
    im_boolean = 32; // ??

    let mut im_floating: f32 = true; // ??

    let mut im_integer = 32;
    im_integer = 64.0; // ??
}
```

Rust programming language is **statically typed**. Meaning, the type of the value must match.

Any type mismatched that occurs will be considered as an error and the Rust compiler will reject the code.

Strong Typing

```
fn main() {
    let mut im_boolean = true;
    im_boolean = false;
    im_boolean = 32; // ??

    let mut im_floating: f32 = true; // ??

    let mut im_integer = 32;
    im_integer = 64.0; // ??
}
```

The variable `im_boolean` is declared here and is inferred to have the type of `bool`.

However, it's later assigned with `32`, which has the type of `i32`, and causes a type mismatched error.

The type of variable `im_floating` has been explicitly defined as `f32` but it's initialized with boolean `true`, which is a type mismatch.

The variable `im_integer` has been inferred as `i32` but `64.0` is assigned here, which is a floating point. Therefore, this produces an error.

Exercise

DATA TYPE

Try to spot an error found within this code and give the explanation of why.

Hint: there are 2 errors.

```
fn main() {
    let my_age: u32 = -18;

    let mut my_height = 155;
    my_height = 156;
    my_height = 157.5;
}
```

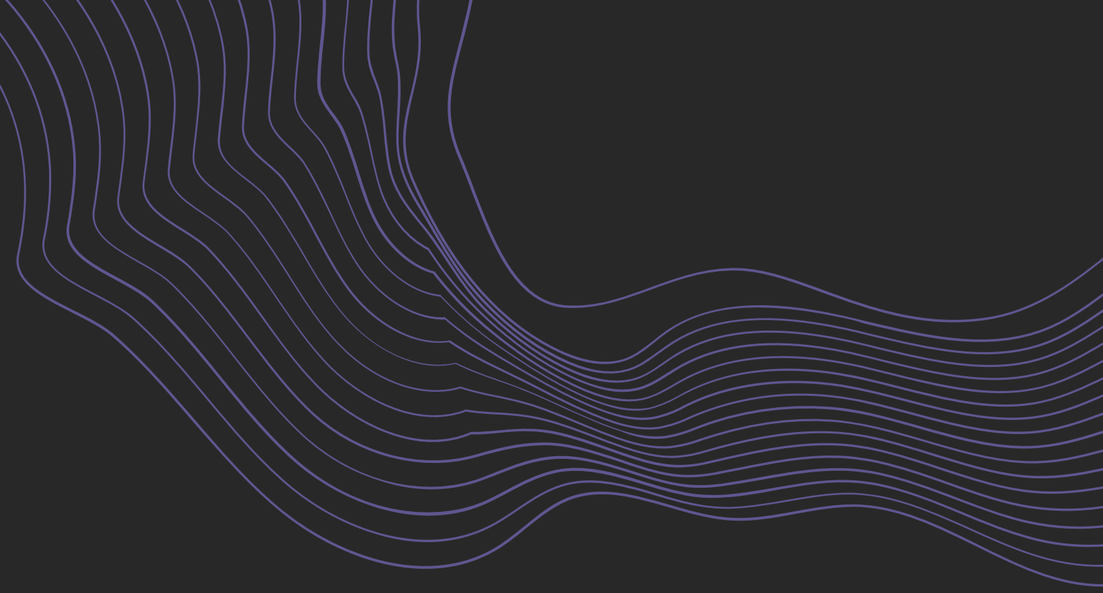
Solution

```
fn main() {
    let my_age: u32 = -18;

    let mut my_height = 155;
    my_height = 156;
    my_height = 157.5;
}
```

The unsigned integer can’t be negative!

The variable `my_height` has been inferred to have a type of `i32` but a floating point is assigned here.

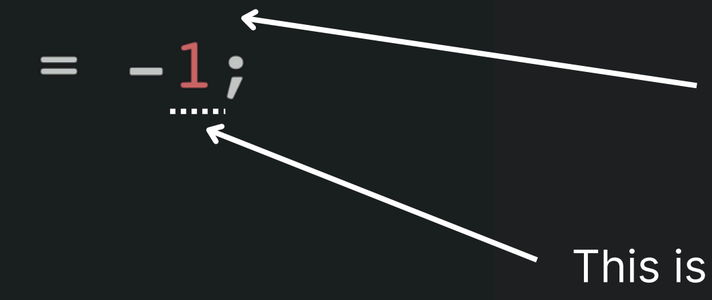
Operators



Operators

Operators are used to perform some
specific computations on the operand(s).

```
fn main() {
    let one_plus_one = 1 + 1;
    let negative_one = -1;
}
```



This is called an **operator.**

This is called an **operand.**

**The operators that operates on two operands such as addition and subtraction are called Binary Operator*

**Meanwhile, the operator that operates on one operand such as negation is called a Unary Operator.*

Arithmetic Operators

```
fn main() {
    let addition = 1 + 2;
    let subtraction = 1 - 3;
    let multiplication = 2 * 2;
    let division = 5 / 2;
    let another_division = 5.0 / 2.0;
    let modulo = 10 % 3;
}
```

10 divided by 3 equals 3 with the remainder of 1. Therefore, this expression evaluates to 1

Arithmetic Operators operate on number types (signed/unsigned integer and floating point).

The addition, subtraction, multiplication, and division are very straightforward and can work with any number types.

However, the modulo operator (% sign) here is different.

The modulo gives you the remainder of the division. This operator only works on signed/unsigned integer types.

Division on Integer vs Floating

```
fn main() {
    let addition = 1 + 2;
    let subtraction = 1 - 3;
    let multiplication = 2 * 2;
    let division = 5 / 2;
    let another_division = 5.0 / 2.0;
    let modulo = 10 % 3;
}
```

The division on an integer (here it is 5 and 2) will result in an integer that is floored.

For example, an expression `5 / 2` evaluates to integer `2`, ***NOT FLOATING POINT `2.5`***

The division here will work as normal, resulting to `2.5`.

Strong Typing is also Applied for Arithmetic Operator

```
fn main() {
    let im_i32: i32 = 1;
    let im_u64: u64 = 2;

    let addition = im_i32 + im_u64;
    let subtraction: f64 = im_i32 + im_i32;
}
```

Both operands for the operators must be the same ***type***. The ***type*** of these arithmetic operations will be the same as their operands.

This expression is invalid since both operands have different types.

This expression is valid, both operands have the same type.

However, the expression evaluates to a value of type `i32` but is assigned to a variable with type `f32`. Therefore, this statement is invalid.

Negation

```
fn main() {
    let signed_number: i64 = 20;
    let negated = -signed_number;
}
```

Negation is a Unary Operator that works on only signed numbers, which includes floating points.

It yields a negated value with the same type as its operand.

Comparison Operators

```
fn main() {
    let equal: bool = 2 == 2;
    let not_equal: bool = 1 != 2;
    let greater_than: bool = 3 > 2;
    let less_then  = 3 < 2;
    let greater_than_or_equal = 3 >= 3;
    let less_than_or_equal = 2 <= 4;
}
```

**Double equal sign `==` is an equality comparison wheras single equal sign `=` is an assignment.*

Comparison Operators operate on number types (signed/unsigned integer and floating point) and boolean; they compare the value between operands and yield a ***boolean type*** (true or false).

Strong typing is also applied here, the type between two operands must be the same.

Every comparison here evaluates to a boolean `true`.

Comparison Operators

```
fn main() {
    let equal = true == true;
    let not_equal = false != true;
    let greater = true > false;
    let less_than = false < true;
}
```

Comparison Operators do work on boolean too! However, you won't see it being used quite often especially with greater than and less than.

Logical Operators

```
fn main() {
    let and = true && true;
    let or  = true || false;
    let not = !false;
}
```

Logic Operators work on the boolean values and also yield a boolean.

This is called the `logical and` operator. In short, this operator yields the value `true` if ***both*** operands are `true`.

This is called the `logical or` operator. This operator yields the value `true` if ***any*** one of the operands is `true`.

This is called the `logical not` operator. It flips the value of the value (true becomes false and false becomes true).

Logical Operators

LHS	RHS	Value
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

Logical AND

LHS	RHS	Value
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

Logical OR

Logical Operators

Operand	Value
FALSE	TRUE
TRUE	FALSE

Logical NOT

Operator	Symbol	Kind	Operand Type	Result Type
Addition	+	Binary	Any Numeric Type	Same as Operand’s Type
Subtraction	-	Binary	Any Numeric Type	Same as Operand’s Type
Multiplication	*	Binary	Any Numeric Type	Same as Operand’s Type
Division	/	Binary	Any Numeric Type	Same as Operand’s Type
Modulo	%	Binary	Any Integer Type	Same as Operand’s Type
Negation	-	Unary	Any Signed Number Type	Same as Operand’s Type
Equal	==	Binary	Any Type	`bool`
Not Equal	!=	Binary	Any Type	`bool`
Greater	>	Binary	Any Type	`bool`
Less	<	Binary	Any Type	`bool`
Greater or Equal	>=	Binary	Any Type	`bool`
Less or Equal	<=	Binary	Any Type	`bool`

Operator	Symbol	Kind	Operand Type	Result Type
Logical And	&&	Binary	`bool`	`bool`
Logicl Or	||	Binary	`bool`	`bool`
Logic Not	!	Unary	`bool`	`bool`

Operator Precedence and Evaluation Order

In an expression with multiple operators chained together, the operator precedence will determine which operator will be evaluated first

The precedence is shown below. The operators within the same column have the same precedence.

lower → higher

- Logical Or `||`

- Logical And `&&`

- Equal `==`
- Not Equal `!=`
- Greater than `>`
- Less than `<`
- Greater than or Equal `>=`
- Less than or Equal `<=`

- Addition `+`
- Subtraction `-`

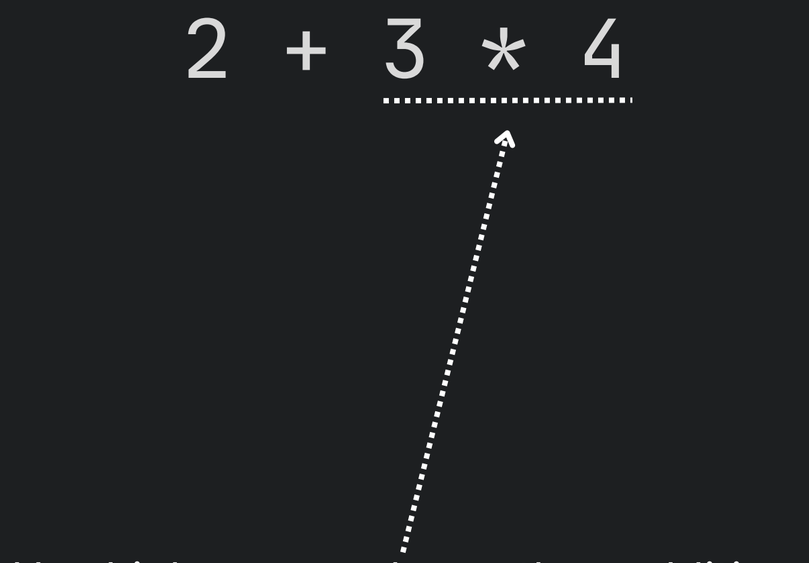
- Multiplication `*`
- Division `/`
- Modulo `%`

- Negation `-`
- Logical Not `!`

First Example: Evaluate the Following Expression

```
2 + 3 * 4
```

First Example: Evaluate the Following Expression



First Example: Evaluate the Following Expression

$$2 + 12$$

Then, do the addition

First Example: Evaluate the Following Expression

14

= 2 + *3 * 4*

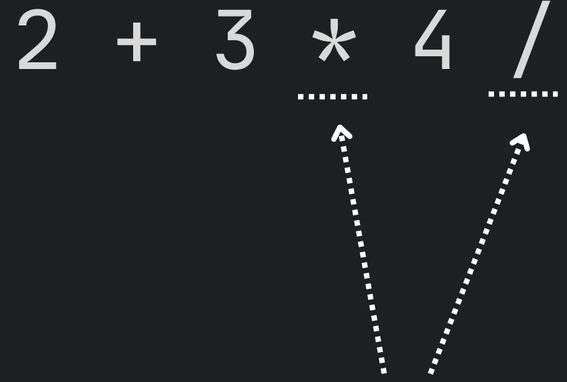
= *2 + 12*

= *14*

Second Example: Evaluate the Following Expression

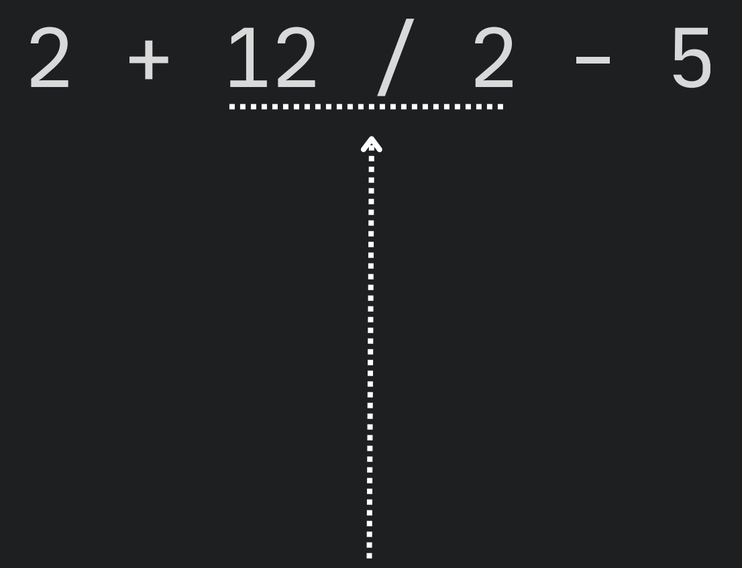
```
2 + 3 * 4 / 2 - 5
```

Second Example: Evaluate the Following Expression



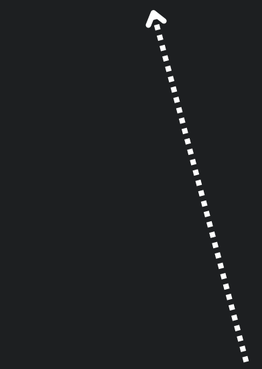
Both multiplication `*` and division `/` have the highest precedence. In this case, we proceed from the left to right, which is multiplication.

Second Example: Evaluate the Following Expression



Second Example: Evaluate the Following Expression

$$2 + 6 - 5$$



Again, both addition and subtraction have the same precedence, proceed from left to right.

Second Example: Evaluate the Following Expression

$$8 - 5$$

Do the subtraction

Second Example: Evaluate the Following Expression

```
2 + 3 * 4 / 2 - 5
```

```
= 2 + 3 * 4 / 2 - 5
= 2 + 12 / 2 - 5
= 2 + 6 - 5
= 8 - 5
= 3
```

Parentheses

```
(2 + 3) * 4 / 2 - 5
```

```
= (2 + 3) * 4 / 2 - 5
= 5 * 4 / 2 - 5
= 20 / 2 - 5
= 10 - 5
= 5
```

Exercise

OPERATORS

Write a program that calculates the area and circumference of a circle.

The program must read the value stored in the variable `radius`, calculate the area and circumference, and store it to the corresponding variables.

Finally print it out to the console using `println!`

Try changing the value of the variable `radius` and see how the program outputs differently

```
fn main() {
    // use `f64` for calculation with decimal
    let radius: f64 = 10.0 /*try changing this value*/;

    let area = /*what goes here?*/;
    let circumference = /*what goes here?*/

    println!(/*what goes here*/);
    println!(/*what goes here*/);
}
```

If-Else

If

```
fn main() {
1    let condition = true;

     if condition {
     2
          println!("the condition was true");
          println!("this is if construct");
     }

     println!("program is ending");
}
```

If-Else constructs are used to conditionally change the flow of statement execution depending on the boolean expression.

This can be replaced with any ***expression*** that yields a boolean. This determines whether the statements in the scope **2** will be executed or not.

```
the condition was true
this is if construct
program is ending
```

If

```
fn main() {
1    let condition = false;

     if condition {
     2   println!("the condition was true");
         println!("this is if construct");
     }

     println!("program is ending");
}
```

This ***expression*** evaluates to `false`. Therefore, the statements inside the scope ***2*** will be skipped.

If you execute this program, you will only see the text `***program is ending***` on the console.

```
program is ending
```

If

```
fn main() {
    if 3 > 2 {
        println!("three is greater than two");
    }

    println!("program is ending");
}
```

This also works since the expression `***3 > 2***` yields a boolean expression.

```
three is greater than two
program is ending
```

If with Else

```
fn main() {
1    let age = 20;
     let mut can_vote = true;

     if age >= 18 {
2        println!("you are an adult");
     } else {
3        can_vote = false;
         println!("you are not an adult");
     }

     println!("can you vote: {can_vote}");
     println!("program is ending");
}
```

The `else` keyword followed by a scope of statements can be inserted after the `if` construct.

The statements in scope ***3*** will be executed if the expression `*age >= 18*` evaluates to false, which doesn't; therefore, the statements will not be executed.

**The prior `if` construct will behave the same.*

```
you are an adult
can you vote: true
program is ending
```

If with Else

```
fn main() {
    let age = 17;
    let mut can_vote = true;

    if age >= 18 {
        println!("you are an adult");
    } else {
        can_vote = false;
        println!("you are not an adult");
    }

    println!("can you vote: {can_vote}");
    println!("program is ending");
}
```

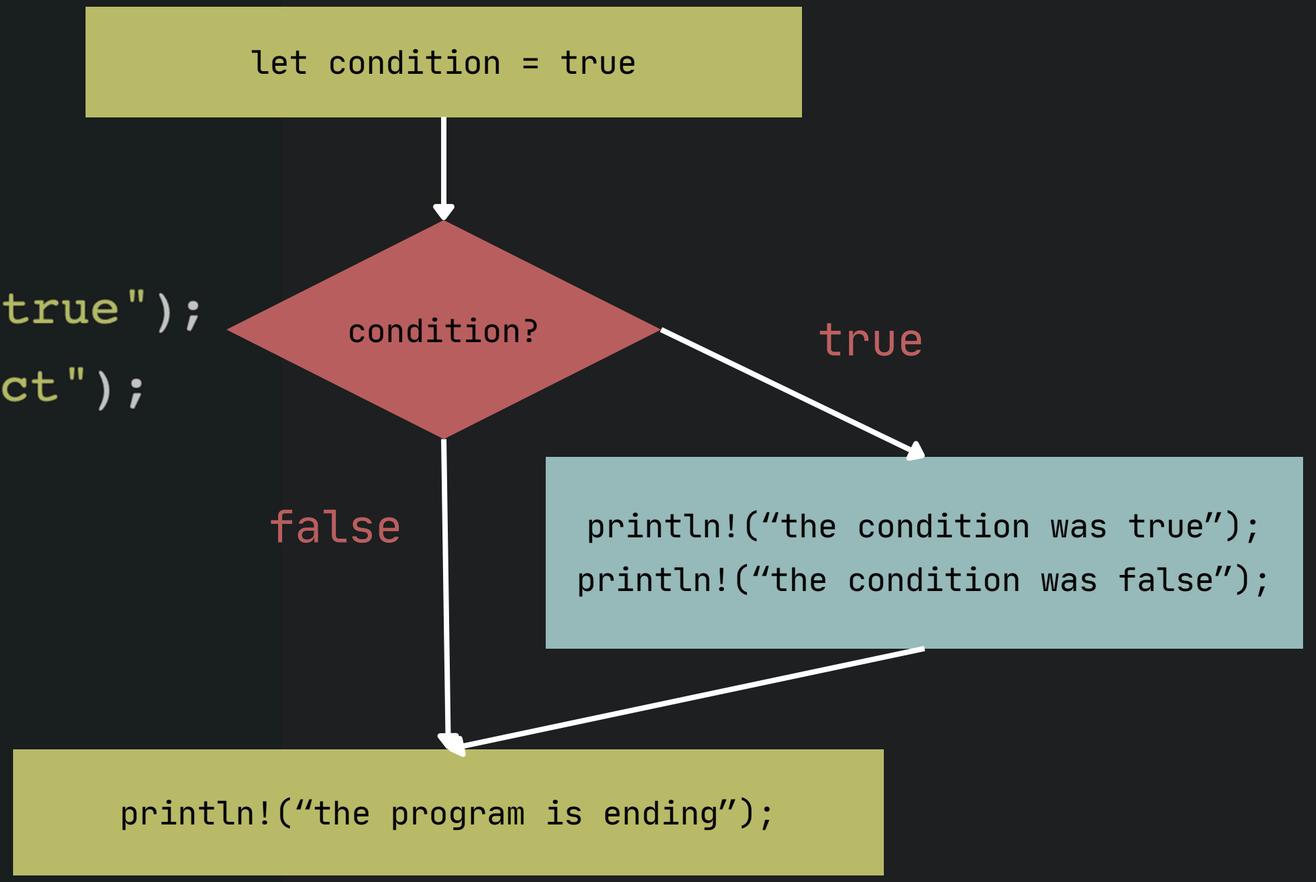
The expression now evaluates to `false`.

These two statements will be executed.

```
you are not an adult
can you vote: false
program is ending
```

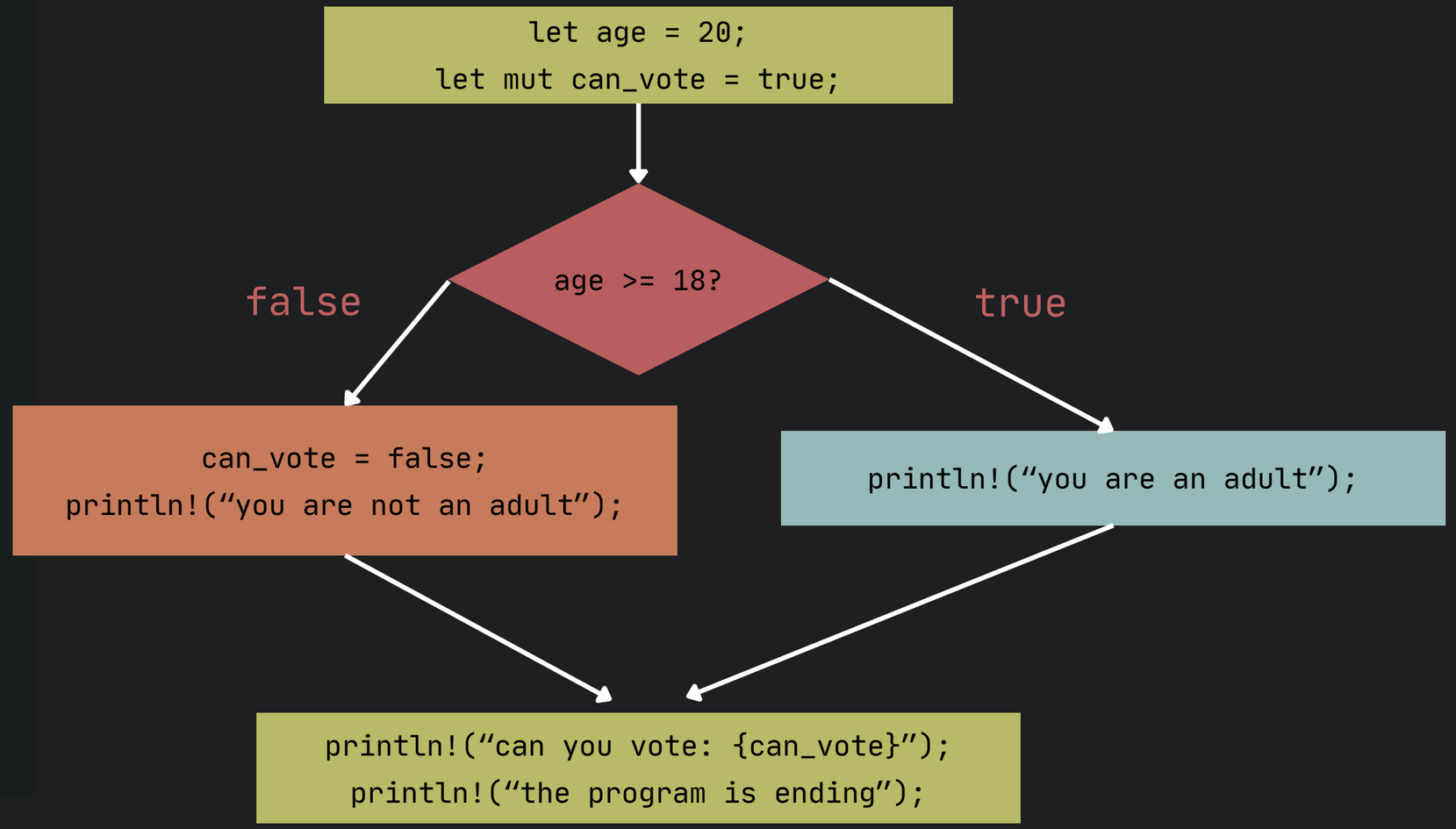
Describing If-Else with Flowchart

```
fn main() {
1    let condition = true;

     if condition {
     2    println!("the condition was true");
          println!("this is if construct");
     }

     println!("program is ending");
}
```



Describing If-Else with Flowchart

```
fn main() {
1   let age = 20;
    let mut can_vote = true;

    if age >= 18 {
2       println!("you are an adult");
    } else {
3       can_vote = false;
        println!("you are not an adult");
    }

    println!("can you vote: {can_vote}");
    println!("program is ending");
}
```



Exercise

IF-ELSE

Write a program that determines the state of the water based on its temperature in Celsius.

There is freezing, normal, and boiling.

Incorporate these conditions into the program:

- Temperature less than 0, the water is ***ice.***
- Temperature greater than 100, the water is ***boiling.***
- Otherwise, the water is ***freezing.***

```
fn main() {
    let water_temperature = 24 /*can be changed*/;

    // complete the rest of the program
}
```

Hint

```
fn main() {
    let water_temperature = 24 /*can be changed*/;

    if water_temperature > 100 {
        println!("the water is boiling");
    } else {
        // what to put here?
    }
}
```

More Hint

```
fn main() {
    let water_temperature = 24 /*can be changed*/;

    if water_temperature > 100 {
        println!("the water is boiling");
    } else {
        // HINT: it would be some if-else here
    }
}
```

Possible Solution

```
fn main() {
    let water_temperature = 24 /*can be changed*/;

    if water_temperature > 100 {
        println!("the water is boiling");
    } else {
        if water_temperature < 0 {
            println!("the water is freezing");
        } else {
            println!("the water is normal");
        }
    }
}
```

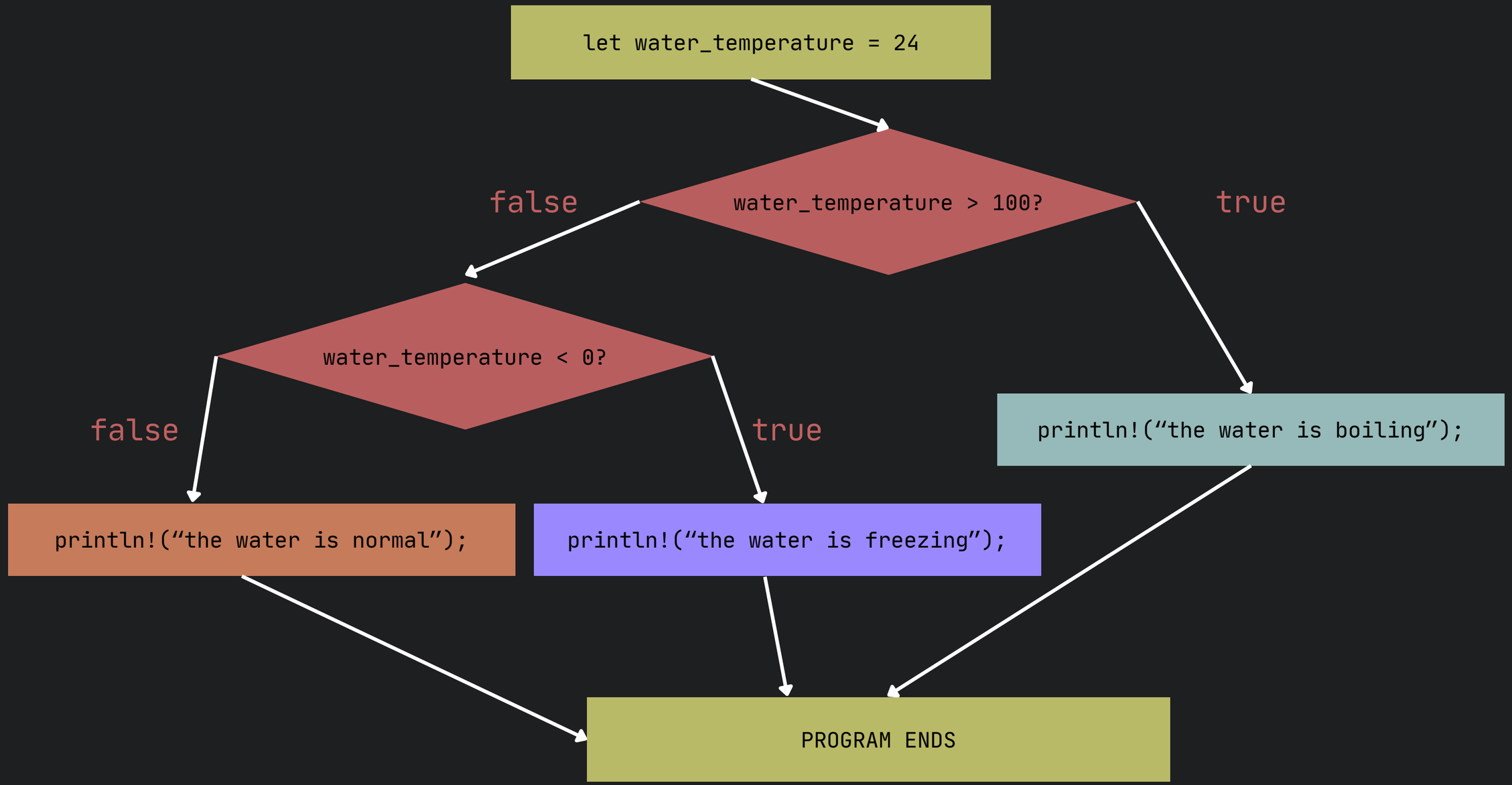
Else-If Trick

```rust
fn main() {
    let water_temperature = 24 /*can be changed*/;

    if water_temperature > 100 {
        println!("the water is boiling");
    } else {
        if water_temperature < 0 {
            println!("the water is freezing");
        } else {
            println!("the water is normal");
        }
    }
}
```

You can remove those two curly braces if the `else` keyword will be immediately followed by another `if` construct.

```rust
fn main() {
    let water_temperature = 24 /*can be changed*/;

    if water_temperature > 100 {
        println!("the water is boiling");
    } else if water_temperature < 0 {
        println!("the water is freezing");
    } else {
        println!("the water is normal");
    }
}
```

These two codes are semantically equivalent (behave the same).


let water_temperature = 24
water_temperature > 100?
false
true
water_temperature < 0?
false
true
println!(“the water is boiling”);
println!(“the water is normal”);
println!(“the water is freezing”);
PROGRAM ENDS