RUSTikales Rust for beginners

Plan for today

Plan for today

1. Recap

Plan for today

1. Recap
2. Basic Types

Plan for today

1. Recap
2. Basic Types
3. Variables

1. Recap

1. Recap

- Setup of tools required to start writing Rust code
  - Rust toolchain
  - IDE
  - rust-analyzer

1. Recap

- Setup of tools required to start writing Rust code
- rustc → Rust Compiler
  - The heart of the language
  - Turns your source code (e.g. main.rs) into an executable

1. Recap

- Setup of tools required to start writing Rust code
- rustc → Rust Compiler
- cargo → Package Manager
  - Manages packages (crates), like third-party libraries
  - Many utility functions such as cargo run or cargo new
  - Overkill for 99% of the things we're doing, but extremely useful in general

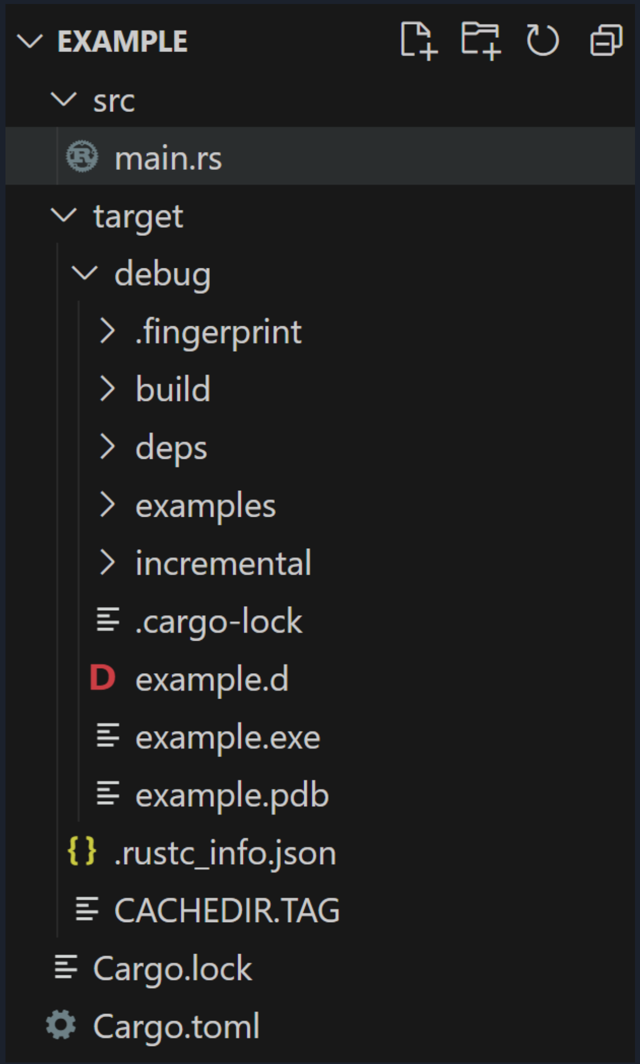
1. Recap

- Setup of tools required to start writing Rust code
- rustc → Rust Compiler
- cargo → Package Manager
- rustup → Toolchain Manager
  - Rust comes in different versions: stable, beta, nightly
  - Allows us to update the version of the Compiler, or switch to different toolchains (e.g. for cross-compilation)

1. Recap

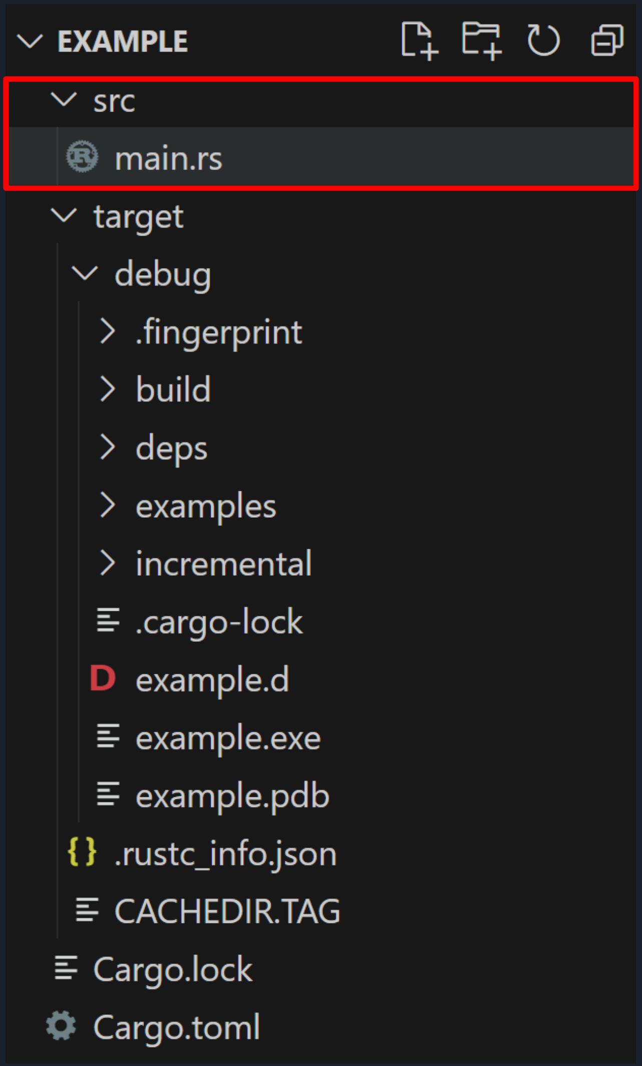
- Setup of tools required to start writing Rust code
- rustc → Rust Compiler
- cargo → Package Manager
- rustup → Toolchain Manager
- cargo new <name> creates a new package in the directory <name>
  - Unless specified otherwise, it's a binary (application) package → Executable
  - We can then open that directory in an IDE and start programming

1. Recap



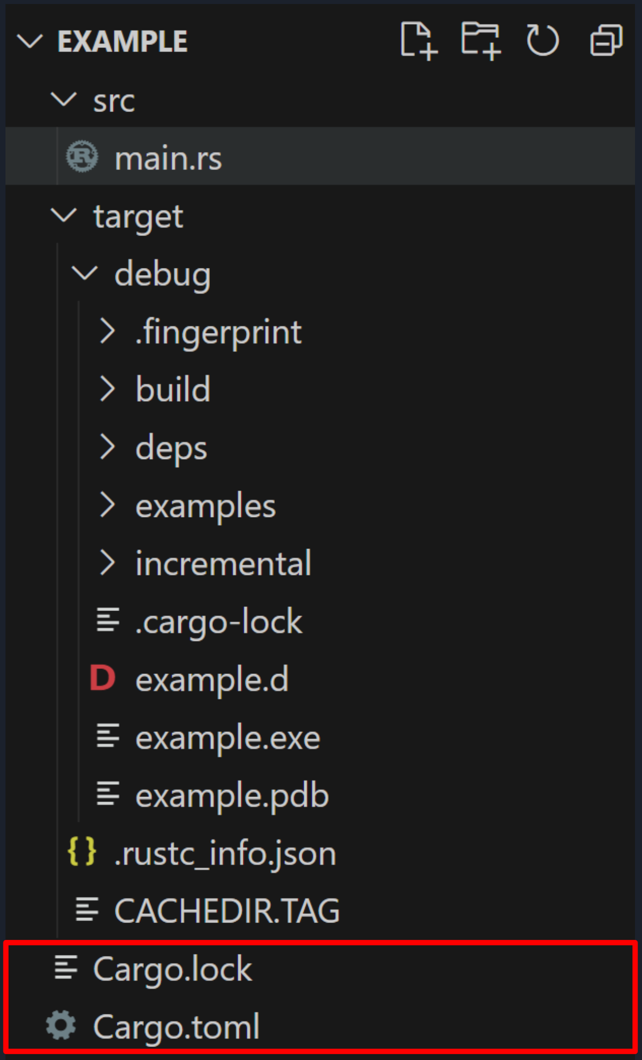
– cargo new example creates this folder structure

1. Recap



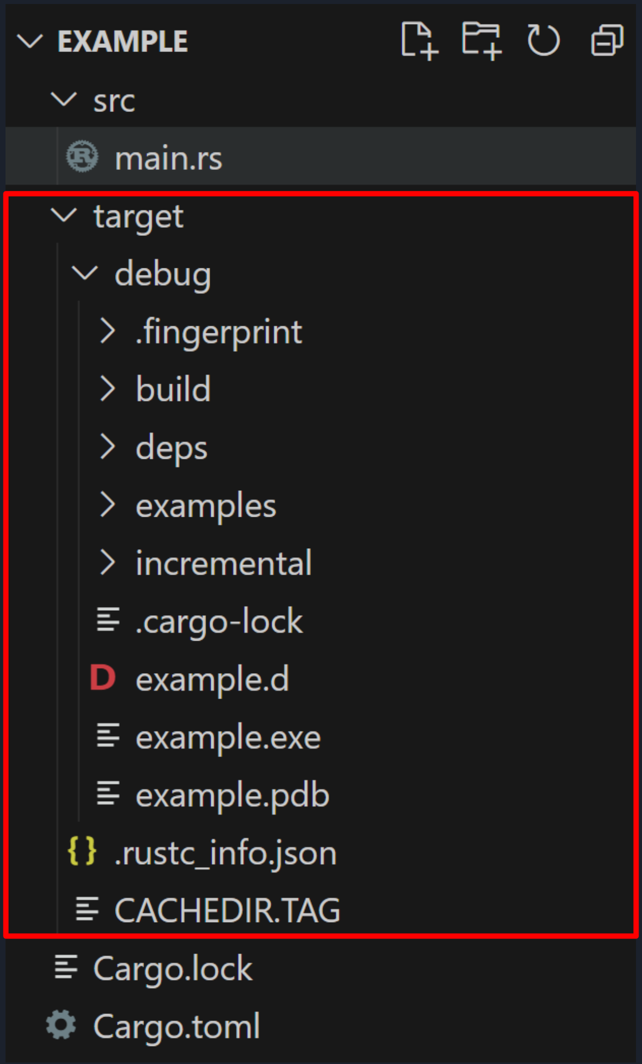
- cargo new example creates this folder structure
- We will spend most of our time in here
  - main.rs is where we write code

1. Recap



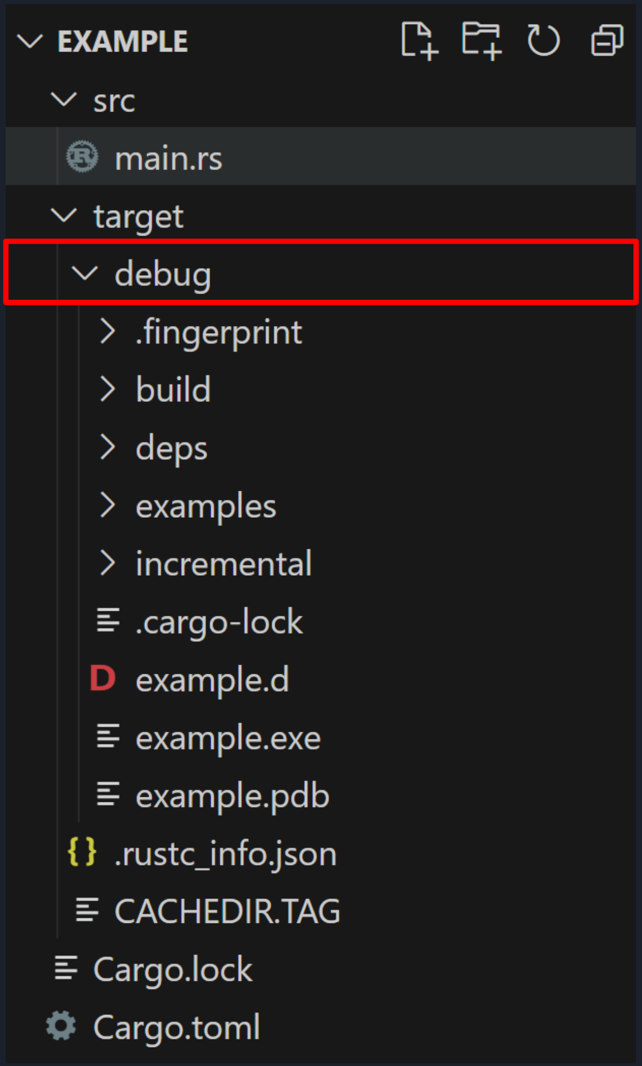
- cargo new example creates this folder structure
- main.rs is where we write code
- We can ignore Cargo.lock and Cargo.toml for now
  - Needed for package dependencies

1. Recap



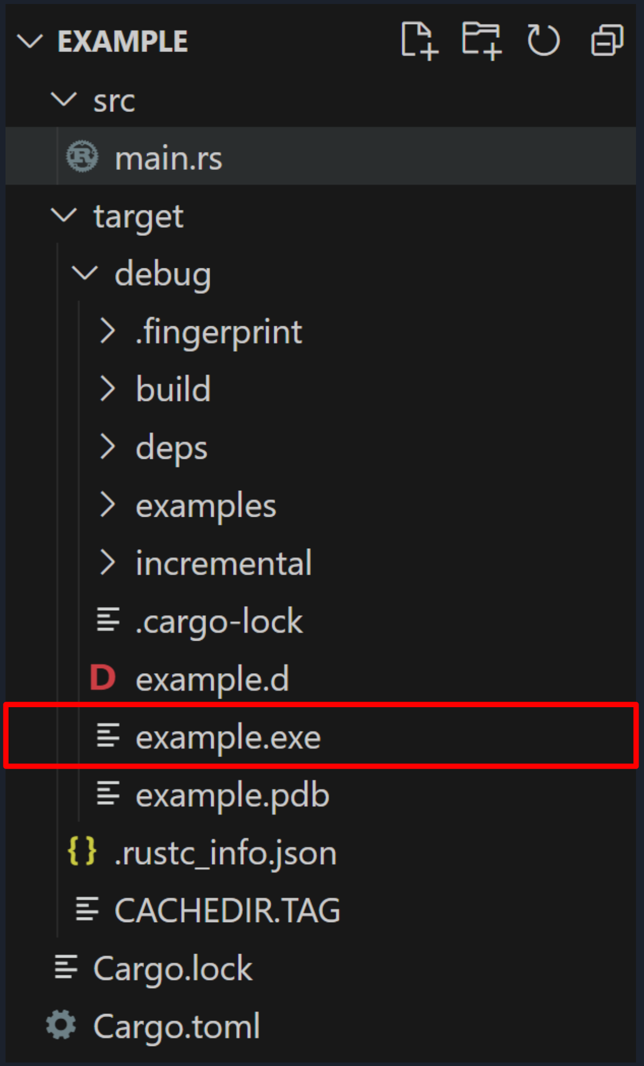
- cargo new example creates this folder structure
- main.rs is where we write code
- We can ignore Cargo.lock and Cargo.toml for now
- cargo moves the output to the target directory

1. Recap



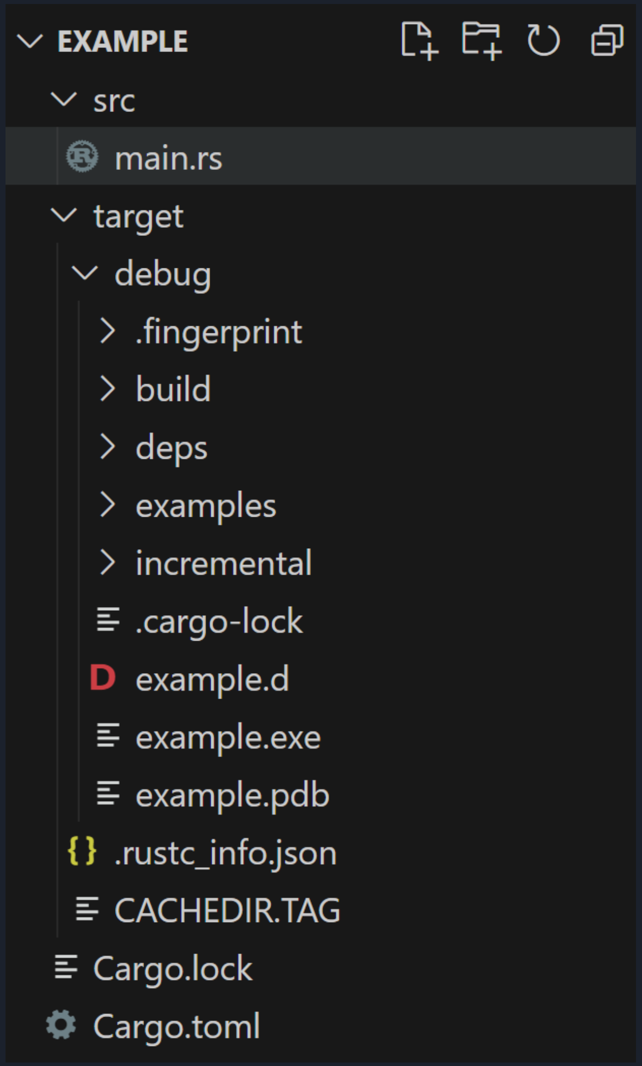
- cargo new example creates this folder structure
- main.rs is where we write code
- We can ignore Cargo.lock and Cargo.toml for now
- cargo moves the output to the target directory
- cargo build creates a debug build
- cargo build --release creates a release build
  - all optimizations enabled

1. Recap



- cargo new example creates this folder structure
- main.rs is where we write code
- We can ignore Cargo.lock and Cargo.toml for now
- cargo moves the output to the target directory
- cargo build creates a debug build
- cargo build --release creates a release build
- you can find the executable here

1. Recap



- cargo new example creates this folder structure
- main.rs is where we write code
- We can ignore Cargo.lock and Cargo.toml for now
- cargo moves the output to the target directory
- cargo build creates a debug build
- cargo build --release creates a release build
- cargo run [--release] builds the project and runs it

2. Basic Types

2. Basic Types

- Rust is a statically typed language
  - Every variable and every value has exactly one type
  - Type must be known at compile time
  - You can’t change that type

2. Basic Types

- Rust is a statically typed language
- The compiler statically type checks your code

2. Basic Types

- Rust is a statically typed language
- The compiler statically type checks your code
  - Finds type errors such as
    - Assigning an i32 to an u64

```rust
let a: i32 = 0;
let b: u64 = a;
```

```
error[E0308]: mismatched types
 --> src\main.rs:5:18
  |
5 |     let b: u64 = a;
  |            ---   ^ expected `u64`, found `i32`
  |            |
  |            expected due to this
```

2. Basic Types

- Rust is a statically typed language
- The compiler statically type checks your code
  - Finds type errors such as
    - Assigning an i32 to an u64
    - Passing an i32 where a u32 was expected

```rust
fn f(x: u32) {}
▶ Run | Debug
fn main() {
    let a: i32 = 0;
    f(a);
}
```

```
error[E0308]: mismatched types
 --> src\main.rs:6:7
  |
6 |     f(a);
  |     - ^ expected `u32`, found `i32`
  |     |
  |     arguments to this function are incorrect
  |
note: function defined here
 --> src\main.rs:3:4
  |
3 | fn f(x: u32) {}
  |    ^ -------
```

2. Basic Types

- Rust is a statically typed language
- The compiler statically type checks your code
  - Finds type errors such as
    - Assigning an i32 to an u64
    - Passing an i32 where a u32 was expected
    - Inserting an u8 into an array of f32

```rust
let mut a: [f32; 2] = [0.0; 2];
a[0] = 5u8;
```

```
error[E0308]: mismatched types
 --> src\main.rs:5:12
  |
5 |     a[0] = 5u8;
  |     ----   ^^^ expected `f32`, found `u8`
  |     |
  |     expected due to the type of this binding
```

2. Basic Types

- Rust is a statically typed language
- The compiler statically type checks your code
- Very useful, because we can easily reason about our code, and prevent many bugs
  - Types are always known, and not changeable
  - Contrast to variables in dynamic languages
    - Assign any value to any variable at any point
    - Hope it doesn't crash at runtime

2. Basic Types

- Rust has many different types

2. Basic Types

- Rust has many different types
  - Scalar types → Represent single values

2. Basic Types

- Rust has many different types
  - Scalar types → Represent single values
    - Integer → whole numbers, signed or unsigned
    - Floating Point → fractions, big numbers
    - boolean → either true or false
    - character → Unicode → abcäøóáßð 😊 😍

2. Basic Types

- Rust has many different types
  - Scalar types → Represent single values
  - Compound types → Combinations of types

2. Basic Types

- Rust has many different types
  - Scalar types → Represent single values
  - Compound types → Combinations of types
    - array → Fixed length collection of values of the same type
    - tuple → Fixed length collection of values of (possibly) different types
    - struct → User-defined collections of values

2. Basic Types

- Rust has many different types
- Today we will only look at integers, other types will be introduced later

2. Basic Types

- Rust has many different types
- Rust has clear names for integer types

2. Basic Types

- Rust has many different types
- Rust has clear names for integer types
  - Letter indicates signed or unsigned
  - Number indicates size in bits

2. Basic Types

- Rust has many different types
- Rust has clear names for integer types
  - Letter indicates signed or unsigned
  - Number indicates size in bits
  - u8 is an unsigned 8bit integer
  - i32 is a signed 32bit integer
  - u16 is an unsigned 16bit integer

2. Basic Types

- Rust has many different types
- Rust has clear names for integer types
- The bitsize shows how big a number can be
  - More bits, bigger numbers
  - 32bits can store numbers as big as 2^32-1 = 4.294.967.295
  - Bigger numbers require more space in memory

2. Basic Types

- Rust has many different types
- Rust has clear names for integer types
- The bitsize shows how big a number can be
- signed means the number can be negative
  - one bit needed to specify the sign, so the number range is smaller
  - i32 can not represent 4.294.967.295
  - i32 can represent -420

2. Basic Types

- Rust has many different types
- Rust has clear names for integer types
- The bitsize shows how big a number can be
- signed means the number can be negative

Type	Meaning	Min Value	Max Value
i8	signed 8bit	-(2^7) = -128	2^7-1 = 127
u8	unsigned 8bit	0	2^8-1 = 255
i16	signed 16bit	-(2^15) = -32768	2^15-1 = 32767
u16	unsigned 16bit	0	2^16-1 = 65535

The pattern repeats up to i128 and u128, doubling the amount of bits every time

2. Basic Types

- Rust also has types isize and usize
  - Target machine dependent size
  - 64bit machine → 64bit wide
  - 32bit machine → 32bit wide

2. Basic Types

- Rust also has types isize and usize
  - Target machine dependent size
  - 64bit machine → 64bit wide
  - 32bit machine → 32bit wide
  - Does anyone know why? 3/3

2. Basic Types

- Rust also has types isize and usize
  - Target machine dependent size
  - 64bit machine → 64bit wide
  - 32bit machine → 32bit wide
  - Does anyone know why? 3/3
    - Used for indexing, sizes, offsets
    - Everything involving memory and pointer arithmetics
    - 32bit systems can't make use of 64bit memory addresses → Must be flexible

2. Basic Types

- Rust also has types isize and usize
  - Target machine dependent size
  - 64bit machine → 64bit wide
  - 32bit machine → 32bit wide
- If you want to index into an array or vector, your index needs to be of type usize

3. Variables

3. Variables

- Variables are a very fundamental part in programming
  - They allow us to store values in memory for later use

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let

```rust
fn main() {
    let a: i8 = -128;
    let b: u8 = 14;
    let c = 20;
    let mut d = 129;
}
```

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let

```rust
fn main() {
    let a: i8 = -128;
    let b: u8 = 14;
    let c = 20;
    let mut d = 129;
}
```

Declarations always follow the same rule:

3. Variables

– Variables are a very fundamental part in programming
– Variables are declared using the keyword let

```rust
fn main() {
    let a: i8 = -128;
    let b: u8 = 14;
    let c = 20;
    let mut d = 129;
}
```

Declarations always follow the same rule:
let

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let

```rust
fn main() {
    let a: i8 = -128;
    let b: u8 = 14;
    let c = 20;
    let mut d = 129;
}
```

Declarations always follow the same rule:
let [mut]

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let

```rust
fn main() {
    let a: i8 = -128;
    let b: u8 = 14;
    let c = 20;
    let mut d = 129;
}
```

Declarations always follow the same rule:
let [mut] name

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let

```rust
fn main() {
    let a: i8 = -128;
    let b: u8 = 14;
    let c = 20;
    let mut d = 129;
}
```

Declarations always follow the same rule:
let [mut] name [:Type]

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let

```rust
fn main() {
    let a: i8 = -128;
    let b: u8 = 14;
    let c = 20;
    let mut d = 129;
}
```

Declarations always follow the same rule:
let [mut] name [:Type] = Expression;

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let

```
fn main() {
    let a: i8 = -128;
    let b: u8 = 14;
    let c = 20;
    let mut d = 129;
}
```

Declarations always follow the same rule:
let [mut] name [:Type] = Expression;
→ mut indicates the mutability of a variable

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let

```
fn main() {
    let a: i8 = -128;
    let b: u8 = 14;
    let c = 20;
    let mut d = 129;
}
```

Declarations always follow the same rule:
let [mut] name [:Type] = Expression;
→ mut indicates the mutability of a variable
→ Type Inference infers the type based on the context in which the variable is used

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let

```
fn main() {
    let a: i8 = -128;
    let b: u8 = 14;
    let c = 20;
    let mut d = 129;
}
```

This code snippet creates four variables

Name	Mutable	Type	Value
a	no	i8	-128
b	no	u8	14
c	no	i32 (inferred)	20
d	yes	i32 (inferred)	129

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let
- Using the keyword mut, we can make our variables mutable

```
fn main() {
    let a: i8 = -128;
    a = 20;
    let mut d: i32 = 129;
    d = 50;
}
```

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let
- Using the keyword mut, we can make our variables mutable

```rust
fn main() {
    let a: i8 = -128;
    a = 20;
    let mut d: i32 = 129;
    d = 50;
}
```

Immutable, we can't re-assign to the variable

```
error[E0384]: cannot assign twice to immutable variable `a`
 --> src\main.rs:5:5
  |
4 |     let a: i8 = -128;
  |         -
  |         |
  |         first assignment to `a`
  |         help: consider making this binding mutable: `mut a`
5 |     a = 20;
  |     ^^^^^^ cannot assign twice to immutable variable
```

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let
- Using the keyword mut, we can make our variables mutable

```rust
fn main() {
    let a: i8 = -128;
    a = 20;
    let mut d: i32 = 129;
    d = 50;
}
```

Mutable, we can re-assign to the variable

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let
- Using the keyword mut, we can make our variables mutable

```
fn main() {
    let a: i8 = -128;
    a = 20;
    let mut d: i32 = 129;
    d = 50;
}
```

rust-analyzer shows us inferred types

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let
- Using the keyword mut, we can make our variables mutable
  - Why does Rust have that system? What do we gain from making variables [im]mutable? 3/3

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let
- Using the keyword mut, we can make our variables mutable
  - Why does Rust have that system? What do we gain from making variables [im]mutable? 3/3
    - Easier to reason your code, make it explicit what you’re expecting to change
    - Very useful when we get to references and the borrow checker
    - Prevents a lot of bugs and oversights
    - TLDR: More control over what is happening

3. Variables

- Variables are a very fundamental part in programming
- Variables are declared using the keyword let
- Using the keyword mut, we can make our variables mutable
- When we assign values to variables, two things can happen:
  - The value is copied
  - The value is moved
  - Will be covered when we talk about Ownership

Intermission - How a program is executed

Intermission - How a program is executed

- Understanding how a computer executes code helps with writing better programs
  - We’ll skip over all technical details, and how the source code turns into machine code

Intermission - How a program is executed

- Understanding how a computer executes code helps with writing better programs
- Computers process instructions sequentially, one after the other
  - Fetch → Decode → Execute

Intermission - How a program is executed

- Understanding how a computer executes code helps with writing better programs
- Computers process instructions sequentially, one after the other
- On a high level, we specify what these instructions should be
  - All the fancy steps in the middle just turn human-readable into machine-executable

Intermission - How a program is executed

```rust
fn main() {
    let a: i32 = 69;
    let b: i32 = 420;
    let c: i32 = a * 1000 + b;
    println!("{}", c);
}
```

Intermission - How a program is executed

```rust
fn main() {
    let a: i32 = 69;
    let b: i32 = 420;
    let c: i32 = a * 1000 + b;
    println!("{}", c);
}
```

1. First, store 69 in a

Intermission - How a program is executed

```rust
fn main() {
    let a: i32 = 69;
    let b: i32 = 420;
    let c: i32 = a * 1000 + b;
    println!("{}", c);
}
```

1. First, store 69 in a
2. Then, store 420 in b

Intermission - How a program is executed

```rust
fn main() {
    let a: i32 = 69;
    let b: i32 = 420;
    let c: i32 = a * 1000 + b;
    println!("{}", c);
}
```

1. First, store 69 in a
2. Then, store 420 in b
3. Read a, multiply its value by 1000

Intermission - How a program is executed

```rust
fn main() {
    let a: i32 = 69;
    let b: i32 = 420;
    let c: i32 = a * 1000 + b;
    println!("{}", c);
}
```

1. First, store 69 in a
2. Then, store 420 in b
3. Read a, multiply its value by 1000
4. Read b, add its value to the result of step 3

Intermission - How a program is executed

```
fn main() {
    let a: i32 = 69;
    let b: i32 = 420;
    let c: i32 = a * 1000 + b;
    println!("{}", c);
}
```

1. First, store 69 in a
2. Then, store 420 in b
3. Read a, multiply its value by 1000
4. Read b, add its value to the result of step 3
5. Store the result of step 4 in c

Intermission - How a program is executed

```rust
fn main() {
    let a: i32 = 69;
    let b: i32 = 420;
    let c: i32 = a * 1000 + b;
    println!("{}", c);
}
```

1. First, store 69 in a
2. Then, store 420 in b
3. Read a, multiply its value by 1000
4. Read b, add its value to the result of step 3
5. Store the result of step 4 in c
6. Read c, print its value to the console

Intermission - How a program is executed

- Understanding how a computer executes code helps with writing better programs
- Computers process instructions sequentially, one after the other
- On a high level, we specify what these instructions should be
- A memory table can help understand what the computer does

Intermission - How a program is executed

```rust
fn main() {
    let a: i32 = 69;
    let b: i32 = 420;
    let c: i32 = a * 1000 + b;
    println!("{}", c);
}
```

Variable	Type	Value
a	i32	???
b	i32	???
c	i32	???
temp*	i32	???

Intermission - How a program is executed

```rust
fn main() {
    let a: i32 = 69;
    let b: i32 = 420;
    let c: i32 = a * 1000 + b;
    println!("{}", c);
}
```

Variable	Type	Value
a	i32	69
b	i32	???
c	i32	???
temp*	i32	???

Intermission - How a program is executed

```
fn main() {
    let a: i32 = 69;
    let b: i32 = 420;
    let c: i32 = a * 1000 + b;
    println!("{}", c);
}
```

Variable	Type	Value
a	i32	69
b	i32	420
c	i32	???
temp*	i32	???

Intermission - How a program is executed

```rust
fn main() {
    let a: i32 = 69;
    let b: i32 = 420;
    let c: i32 = a * 1000 + b;
    println!("{}", c);
}
```

Variable	Type	Value
a	i32	69
b	i32	420
c	i32	???
temp*	i32	69*1000=69.000

Intermission - How a program is executed

```rust
fn main() {
    let a: i32 = 69;
    let b: i32 = 420;
    let c: i32 = a * 1000 + b;
    println!("{}", c);
}
```

Variable	Type	Value
a	i32	69
b	i32	420
c	i32	69.000+420=69.420
temp*	i32	69.000

Intermission - How a program is executed

```rust
fn main() {
    let a: i32 = 69;
    let b: i32 = 420;
    let c: i32 = a * 1000 + b;
    println!("{}", c);
}
```

Variable	Type	Value
a	i32	69
b	i32	420
c	i32	69.420
temp*	i32	69.000

```
Running
69420
```

Intermission - How a program is executed

- Understanding how a computer executes code helps with writing better programs
- Computers process instructions sequentially, one after the other
- On a high level, we specify what these instructions should be
- A memory table can help understand what the computer does
  - very helpful to create one for exercises that compile
  - go step by step through the program, do you get the same output as the computer?

Intermission - How a program is executed

- Understanding how a computer executes code helps with writing better programs
- Computers process instructions sequentially, one after the other
- On a high level, we specify what these instructions should be
- A memory table can help understand what the computer does
- Best debugging advice:
  - Go through your program, step by step, and pretend you're the computer
  - if something doesn't make sense to you, that's usually where the bug is

Intermission - How a program is executed

- Understanding how a computer executes code helps with writing better programs
- Computers process instructions sequentially, one after the other
- On a high level, we specify what these instructions should be
- A memory table can help understand what the computer does
- Best debugging advice:
  - Go through your program, step by step, and pretend you're the computer
  - if something doesn't make sense to you, that's usually where the bug is
  - 🦆: https://en.wikipedia.org/wiki/Rubber_duck_debugging

Intermission - How a program is executed

- Understanding how a computer executes code helps with writing better programs
- Computers process instructions sequentially, one after the other
- On a high level, we specify what these instructions should be
- A memory table can help understand what the computer does
- Best debugging advice:
  - Go through your program, step by step, and pretend you're the computer
  - if something doesn't make sense to you, that's usually where the bug is
  - 🦆: https://en.wikipedia.org/wiki/Rubber_duck_debugging
- Programming involves a lot of logic and reasoning
  - in theory → How computers work, how programming languages work
  - in practice → What I want my computer to do, and when, and how

Intermission - How a program is executed

- Understanding how a computer executes code helps with writing better programs
- Computers process instructions sequentially, one after the other
- On a high level, we specify what these instructions should be
- A memory table can help understand what the computer does
- Best debugging advice:
  - Go through your program, step by step, and pretend you're the computer
  - if something doesn't make sense to you, that's usually where the bug is
  - 🦆: https://en.wikipedia.org/wiki/Rubber_duck_debugging
- Programming involves a lot of logic and reasoning
  - in theory → How computers work, how programming languages work
  - in practice → What I want my computer to do, and when, and how
- Better programming skills → Better logical thinking and reasoning skills, and vice versa

Intermission - Exercise

- Time for exercises!

Intermission - Exercise

1/3

```rust
fn main() {
    let a: i32 = 0;
    let b: i32 = 0;
    let c: u32 = 0;
    let d: i32 = a + b;
    let e: i32 = b + c;
    let f: u32 = (b as u32) + c;
    println!("{}", f);
}
```

Intermission - Exercise

1/3

```rust
fn main() {
    let a: i32 = 0;
    let b: i32 = 0;
    let c: u32 = 0;
    let d: i32 = a + b;
    let e: i32 = b + c;
    let f: u32 = (b as u32) + c;
    println!("{}", f);
}
```

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

Intermission - Exercise

1/3

```rust
fn main() {
    let a: i32 = 0;
    let b: i32 = 0;
    let c: u32 = 0;
    let d: i32 = a + b;
    let e: i32 = b + c;
    let f: u32 = (b as u32) + c;
    println!("{}", f);
}
```

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

Nope, you can‘t add i32 and u32 :^)

```
error[E0308]: mismatched types
 --> src\main.rs:6:22
  |
6 |     let e: i32 = b + c;
  |                      ^ expected `i32`, found `u32`

error[E0277]: cannot add `u32` to `i32`
 --> src\main.rs:6:20
  |
6 |     let e: i32 = b + c;
  |                    ^ no implementation for `i32 + u32`
  |
```

Intermission - Exercise

2/3

```rust
fn main() {
    let a = 0;
    let b: i32 = a;
    let arr: [i32; 2] = [a, b];
    let d = a as usize;
    let e = arr[d];
    println!("{}", e);
}
```

Intermission - Exercise

```rust
fn main() {
    let a = 0;
    let b: i32 = a;
    let arr: [i32; 2] = [a, b];
    let d = a as usize;
    let e = arr[d];
    println!("{}", e);
}
```

Does this code compile?
What type does a have?
What type does d have?
What type does e have?

Intermission - Exercise

2/3

```rust
fn main() {
    let a = 0;
    let b: i32 = a;
    let arr: [i32; 2] = [a, b];
    let d = a as usize;
    let e = arr[d];
    println!("{}", e);
}
```

Does this code compile?
What type does a have?
What type does d have?
What type does e have?

It does compile!
The compiler was able to figure out the types for all variables!

Intermission - Exercise

2/3

```
fn main() {
    let a: i32 = 0;
    let b: i32 = a;
    let arr: [i32; 2] = [a, b];
    let d: usize = a as usize;
    let e: i32 = arr[d];
    println!("{}", e);
}
```

Does this code compile?
What type does a have?
What type does d have?
What type does e have?

It does compile!
The compiler was able to figure out the types for all variables!
Variable a is of type i32
→ used in the context of b
Variable d is of type usize
→ Read a, take its value, interpret as usize
Variable e is of type i32
→ Arrays will be covered next time, but all elements of arr are of type i32

4. Next time

- Arrays
- Vectors