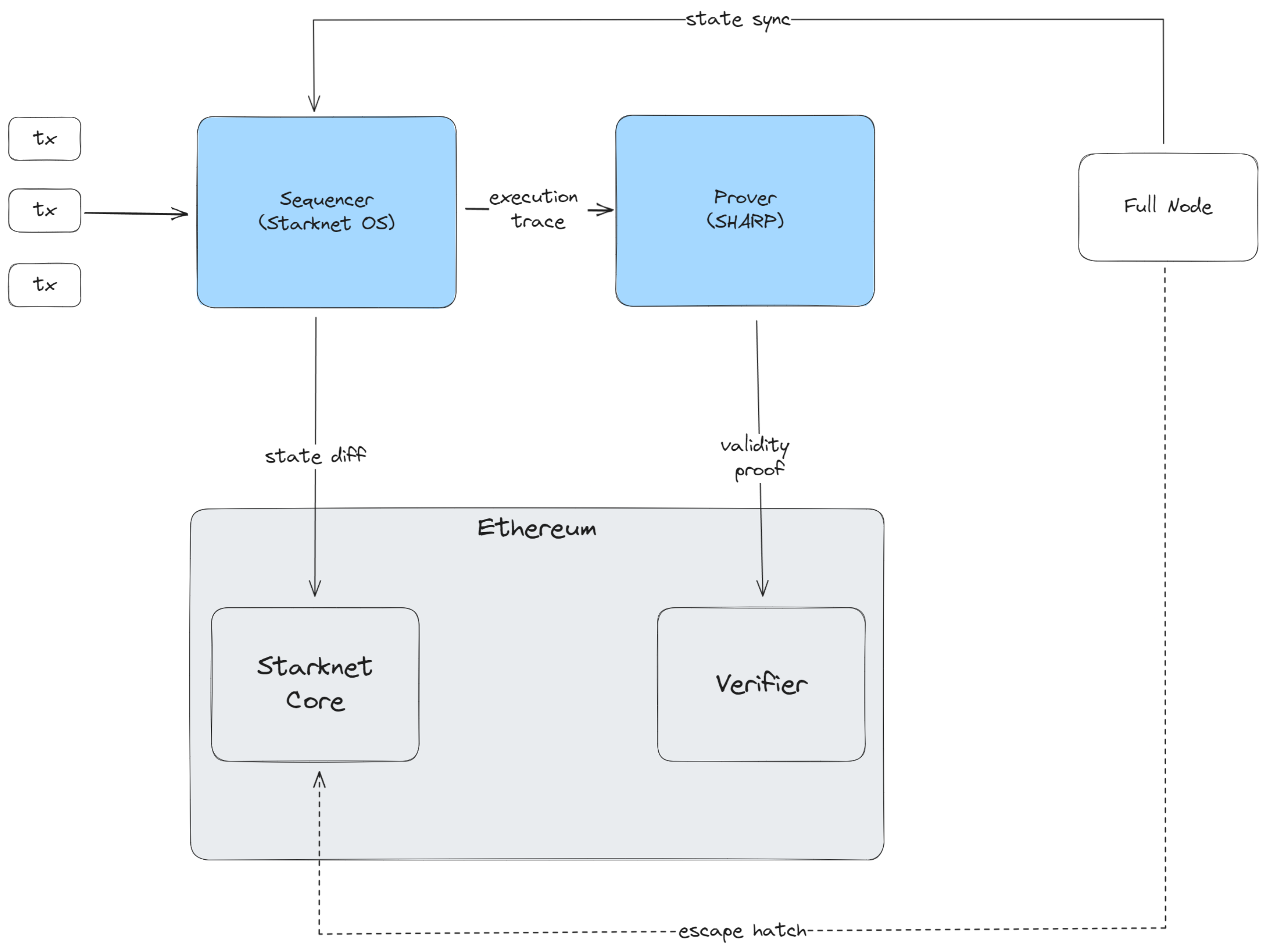
Lesson 5

Starknet architecture overview



See this article for a good overview

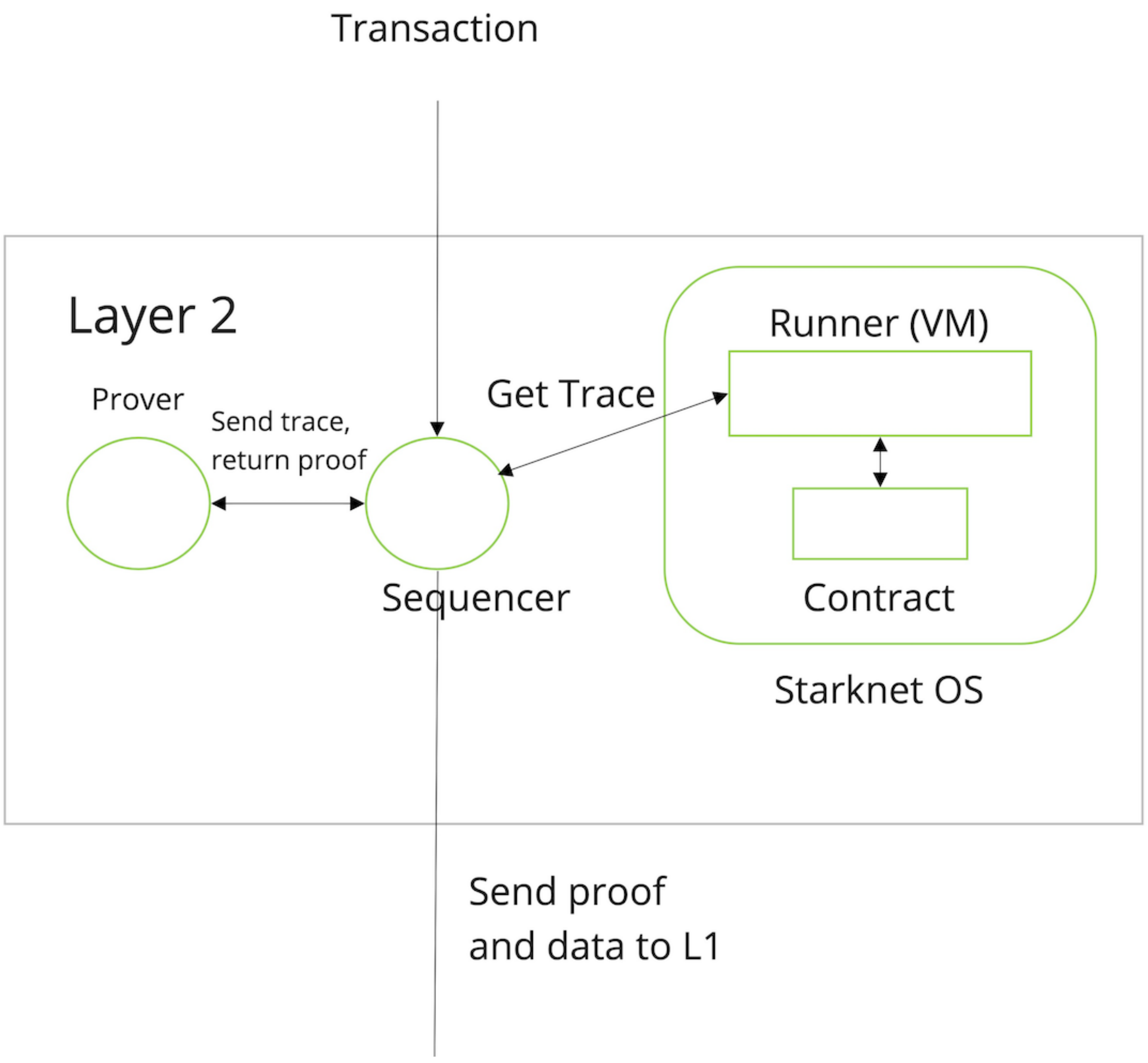
Starknet Components

1. **Prover**: A separate process (either an online service or internal to the node) that receives the output of Cairo programs and generates STARK proofs to be verified. The Prover submits the STARK proof to the verifier that registers the fact on L1.
2. **StarkNet OS**: Updates the L2 state of the system based on transactions that are received as inputs. Effectively facilitates the execution of the (Cairo-based) StarkNet contracts. The OS is Cairo-based and is essentially the program whose output is proven and verified using the STARK-proof system. Specific system operations and functionality available for StarkNet contracts are available as calls made to the OS.
3. **StarkNet State:** The state is composed of contracts' code and contracts' storage.
4. **StarkNet L1 Core Contract**: This L1 contract defines the state of the system by storing the commitment to the L2 state. The contract also stores the StarkNet OS program hash – effectively defining the version of StarkNet the network is running.
   The committed state on the L1 core contract acts as provides as the consensus mechanism of StarkNet, i.e., the system is secured by the L1 Ethereum consensus. In addition to

maintaining the state, the StarkNet L1 Core Contract is the main hub of operations for StarkNet on L1.
Specifically:

- It stores the list of allowed verifiers (contracts) that can verify state update transactions
- It facilitates L1 ↔ L2 interaction

5. **Starknet Full Nodes:** Can get the current state of the network from the sequencer. If the connection between the Sequencer and the Full Node fails for some reason, you can recreate the L2 current state by indexing date from the **Starknet L1 Core Contract** independently



For details of Starknet blocks, see [Block structure]

Rust Continued

Idiomatic Rust

The way we design our Rust code and the patterns we will use differ from say what would be used in Python or JavaScript.
As you become more experienced with the language you will be able to follow the patterns

Memory - Heap and Stack

The simplest place to store data is on the stack, all data stored on the stack must have a known, fixed size.
Copying items on the stack is relatively cheap.
For more complex items, such as those that have a variable size, we store them on the heap, typically we want to avoid copying these if possible.

Clearing up memory

The compiler has a simple job keeping track of and removing items from the stack, the heap is much more of a challenge.
Older languages require you to keep track of the memory you have been allocated, and it is your responsibility to return it when it is no longer being used.
This can lead to memory leaks and corruption.
Newer languages use garbage collectors to automate the process of making available areas of memory that are no longer being used. This is done at runtime and can have an impact on performance.

Rust takes a novel approach of enforcing rules at compile time that allow it to know when variables are no longer needed.

Ownership

Problem

We want to be able to control the lifetime of objects efficiently , but without getting into some unsafe memory state such as having 'dangling pointers'

Rust places restrictions on our code to solve this problem, that gives us control over the lifetime of objects while guaranteeing memory safety.

In Rust when we speak of ownership, we mean that a variable owns a value, for example

```
let mut my_vec = vec![1,2,3];
```

here `my_vec` owns the vector.
(The ownership could be many layers deep, and become a tree structure.)

We want to avoid,

- the vector going out of scope, and `my_vec` ends up pointing to nothing, or (worse) to a different item on the heap
- `my_vec` going out of scope and the vector being left, and not cleaned up.

Rust ownership rules

- Each value in Rust has a variable that’s called its *owner*.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

```
fn main() {
    {
        let s = String::from("hello"); // s is valid from this point forward
        // do stuff with s
    }   // this scope is now over, and s is no longer valid
}
```

Copying

For simple datatypes that can exist on the stack, when we make an assignment we can just copy the value.

```rust
fn main() {
    let a = 2;
    let b = a;
}
```

The types that can be copied in this way are

- All the integer types, such as `u32`.
- The Boolean type, `bool`, with values `true` and `false`.
- All the floating point types, such as `f64`.
- The character type, `char`.
- Tuples, if they only contain these types. For example, `(i32, i32)`, but not `(i32, String)`.

For more complex datatypes such as `String` we need memory allocated on the heap, and then the ownership rules apply

Move

For none copy types, assigning a variable or setting a parameter is a `move`
The source gives up ownership to the destination, and then the source is uninitialised.

```
let a = vec![1,2,3];
let b = a;
let c = a;    // <= PROBLEM HERE
```

Passing arguments to functions transfers ownership to the function parameter

```

let a = ...

loop {

g(a) // <= after the first iteratation, a has lost ownership

}



fn g(x : ...) {

// ...

}
```

Fortunately collections give us functions to help us deal with moves and iteration

```
let v = vec![1,2,3];
for mut a in v {
        v.push(4);
}
```

References

References are a flexible means to help us with ownership and variable lifetimes.
They are written
`&a`
If `a` has type `T` , then `&a` has type `&T`

These ampersands represent *references*, and they allow you to refer to some value without taking ownership of it.

Because it does not own it, the value it points to will not be dropped when the reference stops being used.

References have no effect on their referent's lifetime , so a referent will not get dropped as it would if it were owned by the reference.
Using a reference to a value is called 'borrowing' the value.
References must not outlive their referent.

There are 2 types of references

1. A *shared* reference
   You can read but not mutate the referent.
   You can have as many shared references to a value as you like.
   Shared references are copies
2. A *mutable* reference, mean you can read and modify the referent.
   If you have a mutable ref to a value, you can't have any other ref to that value active at the same time.
   This is following. the 'multiple reader or single writer principle'.
   Mutable references are denoted by `&mut`
   for example this works

   ```
           fn main() {
               let mut s = String::from("hello");

               let s1 = &mut s;
              // let s2 =  &mut s;

               s1.push_str(" bootcamp");

               println!("{}", s1);
              }
   ```

   but this fails

   ```
           fn main() {
               let mut s = String::from("hello");

               let s1 = &mut s;
   ```

```
                let s2 =  &mut s;
                s1.push_str(" bootcamp");

                println!("{}", s1);
               }
```

De referencing

Use the `*` operator

```
let a = 20;
let b = &a;
assert!(*b == 20);
```

String Data types

See Docs

Strings are stored on the heap
A `String` is made up of three components: a pointer to some bytes, a length, and a capacity. The pointer points to an internal buffer `String` uses to store its data. The length is the number of bytes currently stored in the buffer, and the capacity is the size of the buffer in bytes. As such, the length will always be less than or equal to the capacity.

This buffer is always stored on the heap.

```
let len = story.len();
let capacity = story.capacity();
```

We can create a `String` from a literal, and append to it

```
let mut title = String::from("Solana ");
title.push_str("Bootcamp"); // push_str() appends a literal to a String
println!("{}", title);
```

Traits

These bear some similarity to interfaces in other languages, they are a way to define the behaviour that a type has and can share with other types, where behaviour is the methods we can call on that type.
Trait definitions are a way to group method signatures together to define a set of behaviors necessary for a particular purpose.

Defining a trait

```
pub trait Summary {
        fn summarize(&self) -> String;
}
```

Implementing a trait

```
pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

Default Implementations

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

Using traits (polymorphism)

```
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

Introduction to Generics

Generics are a way to parameterise across datatypes, such as we do below with `Option<T>` where `T` is the parameter that can be replaced by a datatype such as `i32`.
The purpose of generics is to abstract away the datatype, and by doing that avoid duplication.

For example we could have a struct, in this example `T` could be an `i32`, or a `u8`, or .... depending how you create the Point `struct` in the main function.
In this case we are enforcing `x` and `y` to be of the same type.

```rust
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let my_point = Point { x: 5, y: 6 };
}
```

The handling of generics is done at compile time, so there is no run time cost for including generics in your code.

Introduction to Vectors

Vectors are one of the most used types of collections.

Creating the Vector

We can use `Vec::new()` To create a new empty vector

```
let v: Vec<i32> = Vec::new();
```

We can also use a macro to create a vector from literals, in which case the compiler can determine the type.

```
let v = vec![41, 42, 7];
```

Adding to the vector

We use `push` to add to the vector, for example

```
v.push(19);
```

Retrieving items from the vector

2 ways to get say the 5th item

- using `get`
  e.g. `v.get(4);`
- using an index
  e.g. `v[4];`

We can also iterate over a vector

```
let v = vec![41, 42, 7];
for ii in &v {
        println!("{}", ii);
}
```

You can get an iterator over the vector with the `iter` method

```
let x = &[41, 42, 7];
let mut iterator = x.iter();
```

There are also methods to `insert` and `remove`
For further details see Docs

Iterators

The iterator in Rust is optimised in that it has no effect until it is needed

```
let names = vec!["Bob", "Frank", "Ferris"];
let names_iter = names.iter();
```

This creates an iterator for us that we can then use to iterate through the collection using `.next()`

```
    fn iterator_demonstration() {
        let v1 = vec![1, 2, 3];
        let mut v1_iter = v1.iter();

        assert_eq!(v1_iter.next(), Some(&1));
        assert_eq!(v1_iter.next(), Some(&2));
        assert_eq!(v1_iter.next(), Some(&3));
        assert_eq!(v1_iter.next(), None);
    }
```

Shadowing

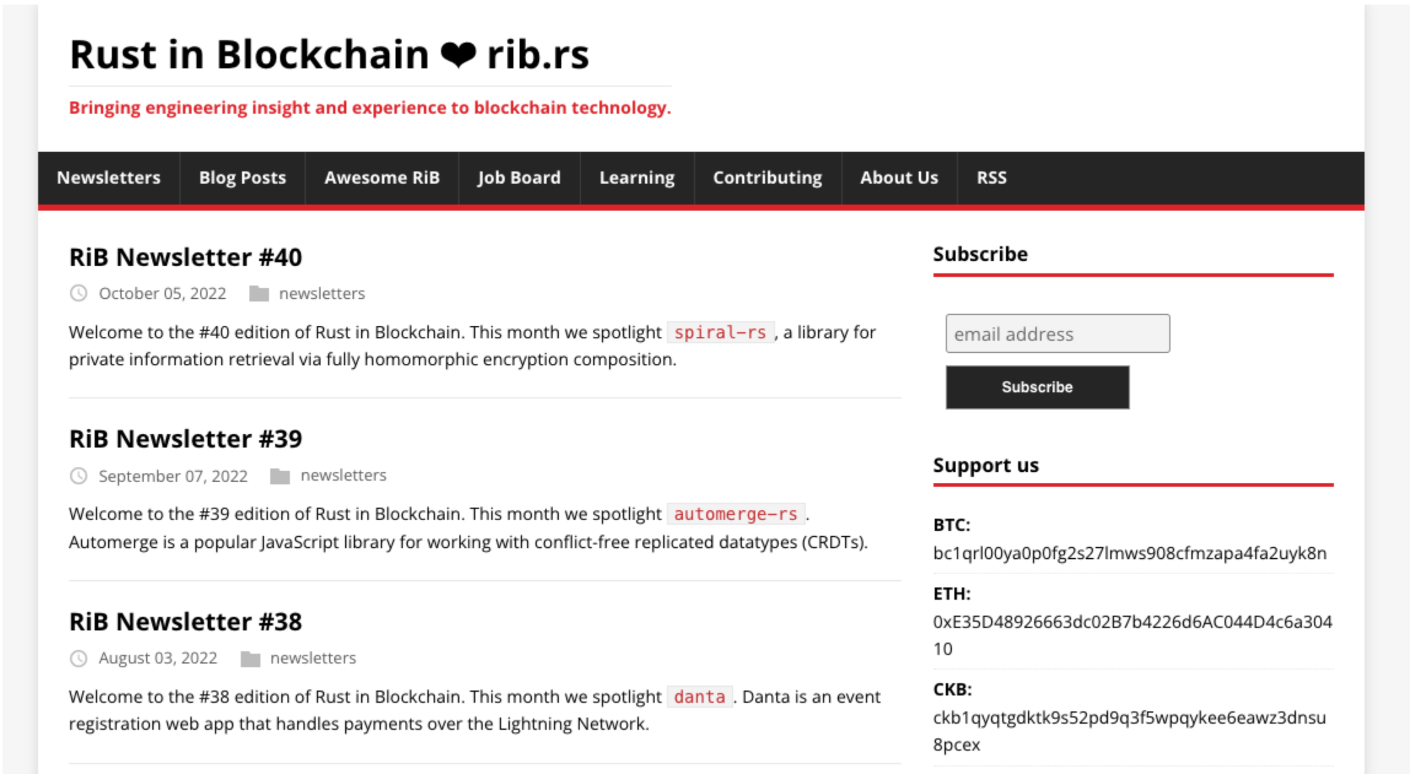
It is possible to declare a new variable with the same name as a previous variable.
The first variable is said to be shadowed by the second,
For example

```rust
fn main() {
    let y = 2;
    let y = y * 3;

    {
        let y = y * 2;
        println!("Here y is: {y}");
    }

    println!("Here y is: {y}");
}
```

We can use this approach to change the value of otherwise immutable variables

Resources

- Rust cheat sheet
- Rustlings
- Rust by example
- Rust Lang Docs
- Rust Playground
- Rust Forum
- Rust Discord
- Rust in Blockchain


Rust in Blockchain ❤ rib.rs

Bringing engineering insight and experience to blockchain technology.

RiB Newsletter #40

October 05, 2022 newsletters

Welcome to the #40 edition of Rust in Blockchain. This month we spotlight `spiral-rs`, a library for private information retrieval via fully homomorphic encryption composition.

RiB Newsletter #39

September 07, 2022 newsletters

Welcome to the #39 edition of Rust in Blockchain. This month we spotlight `automerge-rs`. Automerge is a popular JavaScript library for working with conflict-free replicated datatypes (CRDTs).

RiB Newsletter #38

August 03, 2022 newsletters

Welcome to the #38 edition of Rust in Blockchain. This month we spotlight `danta`. Danta is an event registration web app that handles payments over the Lightning Network.

Subscribe

Support us

BTC:
bc1qrl00ya0p0fg2s27lmws908cfmzapa4fa2uyk8n

ETH:
0xE35D48926663dc02B7b4226d6AC044D4c6a304 10

CKB:
ckb1qyqtgdktk9s52pd9q3f5wpqykee6eawz3dnsu 8pcex

Anatomy of a contract

- `#[starknet::contract]` - declaring a contract

```
#[starknet::contract]
mod math_contract {
        // CODE HERE
}
```

- `#[starknet::interface]` - declaring an interface of a contract

```
#[starknet::interface]
trait iMathContract<TContractState> {
        // CODE HERE
}
```

- `#[starknet::components]` - declaration of components

```
#[starknet::component]
mod ownable_component{
        // CODE HERE
}
```

- `#[storage]` - declaring the storage of a contract

```
    #[storage]
    struct Storage {
        // CODE HERE
    }
```

- `#[constructor]` - declaration of the constructor function for the contract

```
    #[constructor]
    fn constructor(ref self: ContractState) {
        // CODE HERE
    }
```

- `#[external(v0)]` - declaration of public functions

```
    #[external(v0)]
    impl MathContractImpl of super::iMathContract<ContractState> {
                // CODE HERE
        }
```

- `#[event]` - declaration of events

```
#[event]
#[derive(Drop, starknet::Event)]
enum Event {
            // CODE HERE
}
```

- `#[generate_trait]` - generation of the trait of an implementation

```
#[generate_trait]
impl InternalImpl of InternalContract {
            // CODE HERE
    }
```

- Read functions

```
    fn get_total(self: @TContractState) -> u128;
```

- Read/write functions

```
    fn addition(ref self: TContractState, amount: u128);
```

Contract example code here.

Creating your first project

Now that we are all set up, let's create your first project with `scarb`.

```
scarb new hello_world
cd hello_world
```

Once created, let's delve into the project structure.

```
.
├── Scarb.toml
└── src
└── lib.cairo
```

By default, `scarb new` will generate a `Scarb.toml` file, known as the *manifest*.

This file contains metadata necessary for compiling the package. Additionally, it creates a `src` folder that includes the `lib.cairo` file with an example Cairo code.

Now that we have the initial project structure, let's make some modifications to the `Scarb.toml`. To enable the compilation of Starknet contracts, we need to add the following line to our `Scarb.toml`:

`[[target.starknet-contract]]`

In addition, we will have to declare the dependency of the Starknet version. We do this by adding the following line:

```
[dependencies]
starknet = ">=2.2.1"
```

In the end your `Scarb.toml` should look like this:

```
[package]

name = "hello_world"
version = "0.1.0"

[dependencies]
starknet = ">=2.2.0"

[[target.starknet-contract]]
sierra = true
```

There are other exciting things you can do here, such as adding external dependencies, defining custom scripts or profiles, and much more.
If you are interested in learning more, you can refer to the Scarb Cheatsheet for additional information.

Creating our contract

Let's create a new file called `contract.cairo` in our `src` folder.

Once created, we can delete all the code from the `lib.cairo` file and import the module that we will create in `contract.cairo`.

Your `lib.cairo` file should look like this:

```
mod contract;
```

Let's implement our first contract.

Here's an example of how your `contract.cairo` file should look:

```
#[starknet::interface]
trait IHelloStarknet<TContractState> {
    fn increase_balance(ref self: TContractState, amount: felt252);
    fn get_balance(self: @TContractState) -> felt252;
}

#[starknet::contract]
mod HelloStarknet {
    #[storage]
    struct Storage {
        balance: felt252,
    }

    #[abi(embed_v0)]
    impl HelloStarknetImpl of super::IHelloStarknet<ContractState> {
        fn increase_balance(ref self: ContractState, amount: felt252) {
            assert(amount != 0, 'Amount cannot be 0');
            self.balance.write(self.balance.read() + amount);
        }

        fn get_balance(self: @ContractState) -> felt252 {
            self.balance.read()
        }
    }
}
```

Code is here

Now that we have our smart contract, let's try to build it with `scarb` and generate the Sierra JSON output.

Run the following command in your terminal:

```
scarb build
```

If everything worked correctly, you should notice a new folder that was created by `scarb` called `target`.

There we will have our sierra file that was generated by the compiler.
Currently, by default, `scarb` generated 2 different files within the `target/dev` folder.

The one that we will declare and deploy is the `hello_world_contract.contract_class.json` file.

Declaring and deploying your contract

Now, for the final step in our process.

We will deploy the smart contract using `starkli`.

If you haven't set up your account with `starkli`, please follow the steps below:

```
starkli signer keystore new demo-key.json
starkli account oz init demo-account.json --keystore ./demo-key.json
starkli account deploy demo-account.json --keystore ./demo-key.json
```

Now, you should receive an address where you need to send some test funds to deploy your account.

If you need test funds, you can request them from here. Once you have sent the test funds, you can proceed with deploying your account.

Now that we are prepared, it's time to declare your contract.
Run the following command:

```
starkli declare target/dev/hello_world_contract.contract_class.json --account
demo-account.json --keystore ./demo-key.json --compiler-version 2.4.0 --network
goerli-1 --watch
```

Now that you finished declaring your contract, you should receive a hash address. Use this and replace the `HASH` in the following line and run the command.

```
starkli deploy HASH --account demo-account.json --keystore ./demo-key.json
```

And that's it! We have successfully deploy our contract on Testnet.