Lesson 4 - Rust continued / Solana Command Line

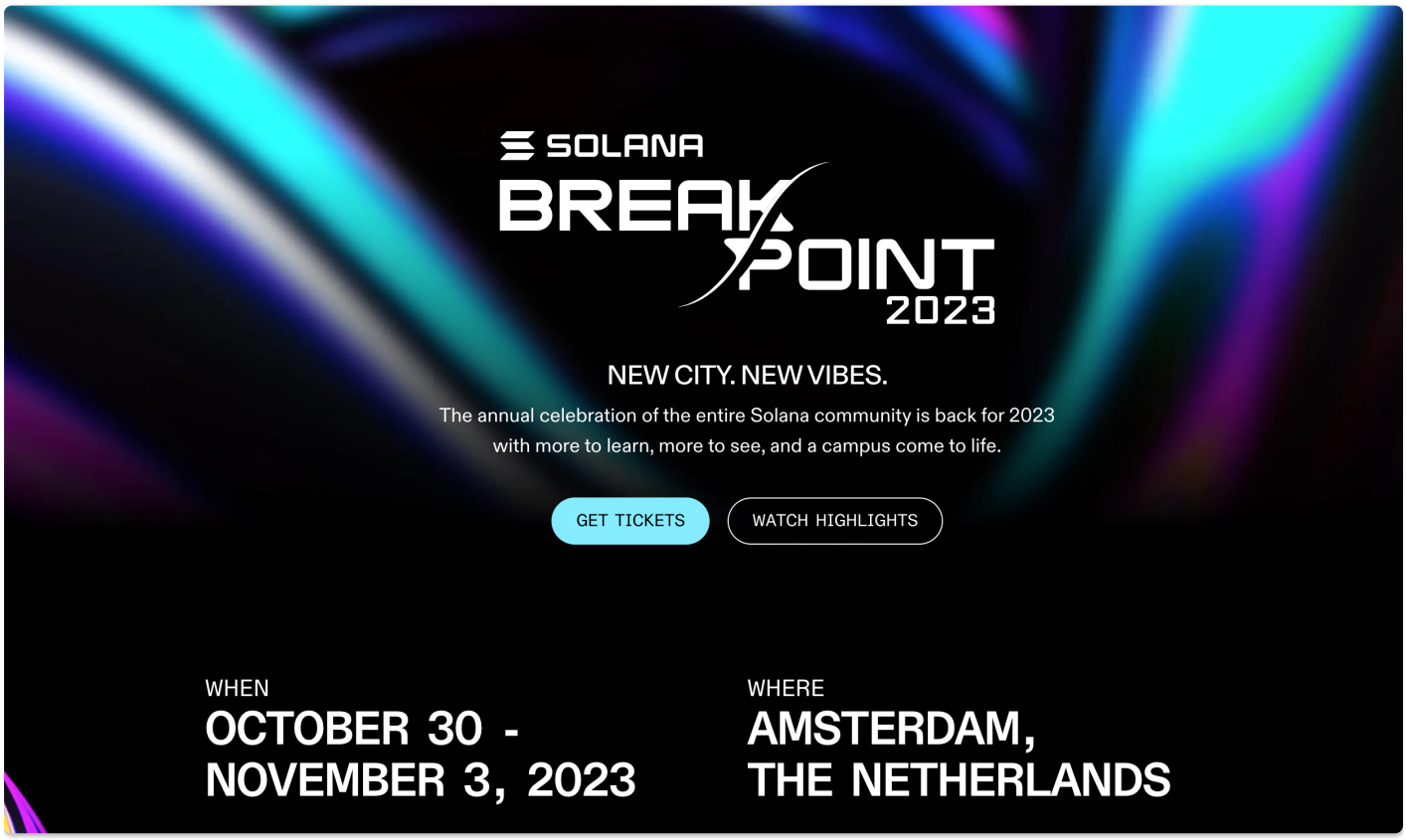
Solana Community

See resources page
This details their telegram / discord channels etc.
There are many meetup groups available worldwide

Hacker House

In 2022 there were hackathons hosted in many cities

Solana Breakpoint - Oct / Nov 2023



Solana Collective

See Docs
This is a program to help Solana supporters contribute to the ecosystem and work with core teams.

Solana Grants

Anyone can apply for a grant from the Solana Foundation.
That includes individuals, independent teams, governments, nonprofits, companies, universities, and academics.

Here is the list of initiatives Solana are currently looking to fund.
and categories they are interested in

- Censorship Resistance
- DAO Tooling
- Developer Tooling
- Education
- Payments / Solana Pay
- Financial Inclusion
- Climate Change
- Academic Research

Solana Command Line tools

Installing Solana Command Line Tools

You can either install the tools locally,
or use the gitpod environment

The instructions are also given in the documentation

Mac / Linux

```
sh -c "$(curl -sSfL https://release.solana.com/v1.14.6/install)"
```

You need to follow the instructions about updating the PATH variable, you can check the installation with
`solana --version`

Windows

Open a command prompt as an administrator
Run
`curl https://release.solana.com/v1.14.6/solana-install-init-x86_64-pc-windows-msvc.exe --output C:\solana-install-tmp\solana-install-init.exe --create-dirs`

`C:\solana-install-tmp\solana-install-init.exe v1.14.6`

Close and re open the command prompt, this time as a normal user
Check the installation with
`solana --version`

To get help run
`solana --help`

Create a keypair

```
mkdir ~/my-solana-wallet
solana-keygen new --outfile ~/my-solana-wallet/my-keypair.json
```

display the result with

```
solana-keygen pubkey ~/my-solana-wallet/my-keypair.json
```

verify your address

```
solana-keygen verify <PUBKEY> ~/my-solana-wallet/my-keypair.json
```

Connect to the dev network

`solana config set --url https://api.devnet.solana.com`
You can check the configuration with
`solana config get`

Get some tokens from dev net
`solana airdrop 1 <RECIPIENT_ACCOUNT_ADDRESS> --url https://api.devnet.solana.com`

you will receive the transaction ID, and can look for this on the dev net block explorer

You can also check your balance with
`solana balance <ACCOUNT_ADDRESS> --url https://api.devnet.solana.com`

See Docs

Note that you need to use the file system wallet we set up yesterday.
In the following `<KEYPAIR>` is the path to that wallet.

Transferring SOL to another account

`solana transfer --from <KEYPAIR> <RECIPIENT_ACCOUNT_ADDRESS> <AMOUNT> --fee-payer <KEYPAIR>`

Checking a balance

`solana balance <ACCOUNT_ADDRESS> --url http://api.devnet.solana.com`

Rust Continued

Ideomatic 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

```language-rust
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.

```language-rust
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

Control Flow

We need to be careful if the control flow in our code could mean a variable is uninitialised

```
  let a = vec![1,2,3];
while f() {
        g(a) <= after the first iteratation, a has lost ownership
}

fn g(x : u8)...
```

Fortunately collections give us functions to help us deal with moves

```
  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

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

Implementing a trait

```language-rust
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

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

Using traits (polymorphism)

```language-rust
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.

```language-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

```language-rust
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.

```language-rust
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

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

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

```language-rust
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()`

```language-rust
    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

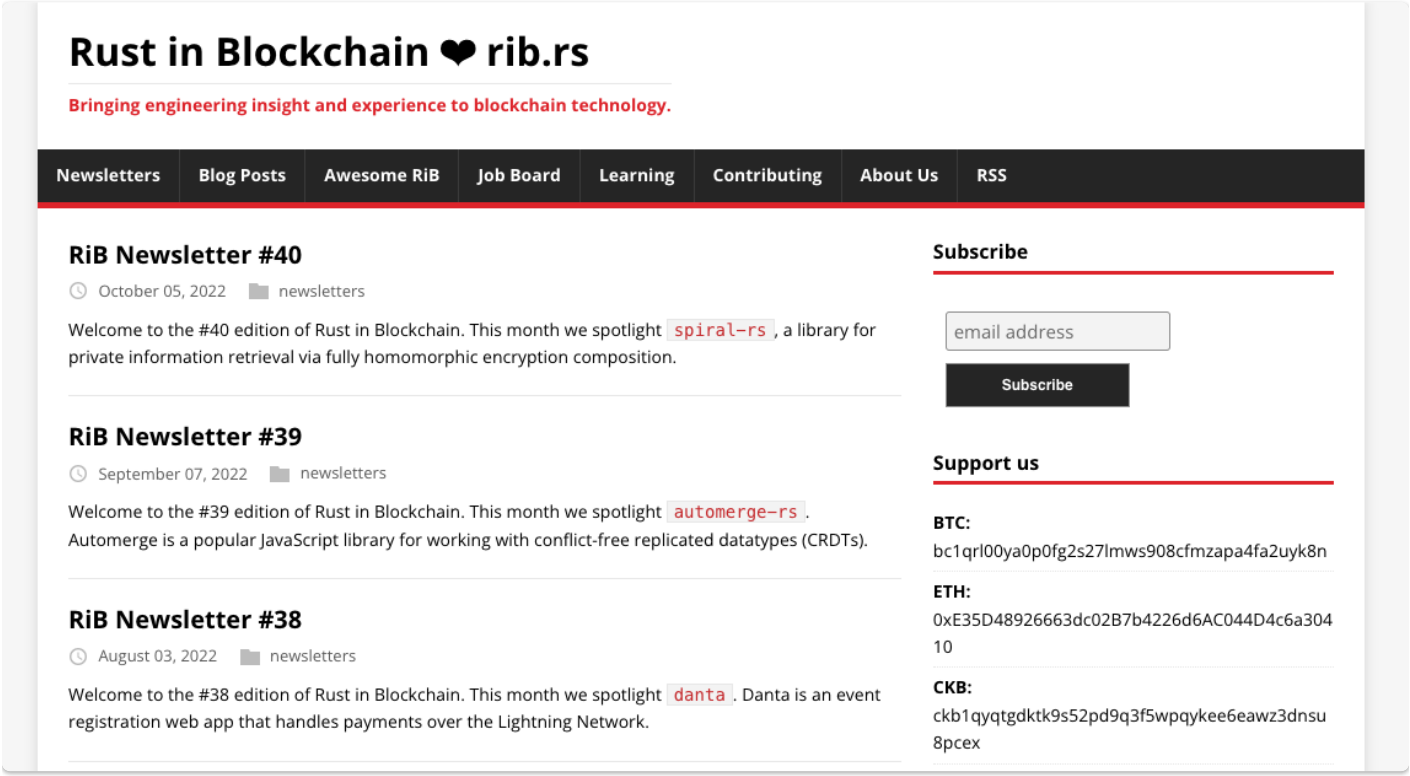
It is possible to declare a new variable with the same name as a previous variable.
The first variable is said to be shadowed by the second,
For example

```
  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



Next Week

- Solana Development tools and environment
- Token Program
- Further Rust
- Solana Web3.js