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
Lisbon Hacker House starts on 1st Nov

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 Tool

Commands continued
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.
I will introduce more patterns as we go through the course.

Introduction to Generics

Generics are a way to parameterise across datatypes, such as we do above 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.

```
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.

Further Datatypes

Enums

See docs
Use the keyword `enum`

```
enum Fruit {
    Apple,
    Orange,
    Grape,
}
```

You can then reference the enum with for example

```
Food::Orange
```

Control Flow

If expressions

See Docs

The `if` keyword is followed by a condition, which *must evaluate to bool* , note that Rust does not automatically convert numerics to bool.

```
    if x < 4 {
        println!("lower");
    } else {
        println!("higher");
    }
```

Note that 'if' is an expression rather than a statement, and as such can return a value to a 'let' statement, such as

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {}", number);
}
```

Note that the possible values of `number` here need to be of the same type.

We also have `else if` and `else` as we do in other languages.

Looping

We have already see for loops to loop over a range, other ways to loop include
`loop` - to loop until we hit a `break`
`while` which allows an ending condition to be specified
See Rust book for examples.

Option

We may need to handle situations where a statement or function doesn't return us the value we are expecting, for this we can use Option.

Option is an enum defined in the standard library.
The `Option<T>` enum has two variants:

- `None`, to indicate failure or lack of value, and
- `Some(value)`, a tuple struct that wraps a `value` with type `T`.

It is useful in avoiding inadvertently handling null values.

Another useful enum is `Result`

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Matching

A powerful and flexible way to handle different conditions is via the `match` keyword
This is more flexible than an `if` expression in that the condition does not have to be a boolean, and pattern matching is possible.

Match Syntax

```
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
}
```

Match Example

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

The keyword `match` is followed by an expression, in this case `coin`
The value of this is matched against the 'arms' in the expression.
Each `arm` is made of a pattern and some code
If the value matches the pattern, then the code is executed, each arm is an expression, so the return value of the whole match expression, is the value of the code in the arm that matched.

Matching with `Option`

```
fn main() {
    fn plus_one(x: Option<i32>) -> Option<i32> {
        match x {
            None => None,
            Some(i) => Some(i + 1),
        }
    }

    let five = Some(5);
```

```
    let six = plus_one(five);
    let none = plus_one(None);
}
```

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);
    }
```

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

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

---

Macros

see Docs

Macros allow us to avoid code duplication, or define syntax for DSLs.
Examples of Macros we have seen are

`vec!` to create a Vector

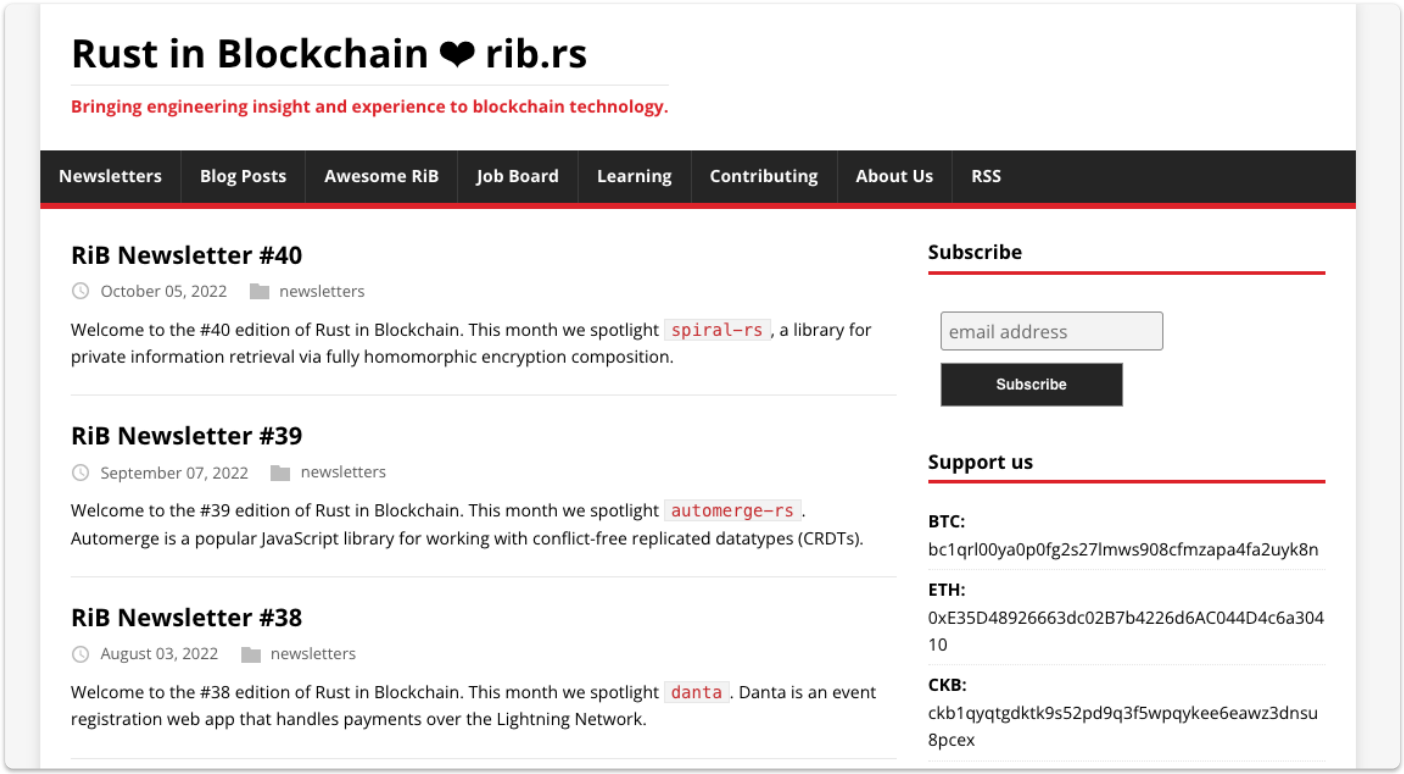
```
let names = vec!["Bob", "Frank", "Ferris"];
```

`println!` to output a line

```
println!("The value of number is: {}", number);
```

Resources

- Rust in Blockchain



Next Week

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