## Lesson 3 - Solana Command Line Tools / Rust

### Solana Command Line Tools

See the guide
You can

- Use Solana's Install Tool (Simplest option)
- Download Prebuilt Binaries
- Build from Source
- Use Homebrew

#### Wallets

You can use different types of wallets
In the homework we will create a file system wallet for you to use with the command line tools, and also show a browser extension wallet.

#### Faucets

There is a faucet that you can use to get SOL for the test networks, there are many others.

---

## 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

```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.

```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.

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

Passing arguments to functions transfers ownership to the function parameter

```
language-rust

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

```
language-rust
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

```rust
        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

```rust
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

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

---

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.

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

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

```rust
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

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

#### Implementing a trait

language-rust

```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

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

#### Using traits (polymorphism)

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

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

```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.

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

#### Adding to the vector

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

```rust
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

```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

```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

```rust
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()`

```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

```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]