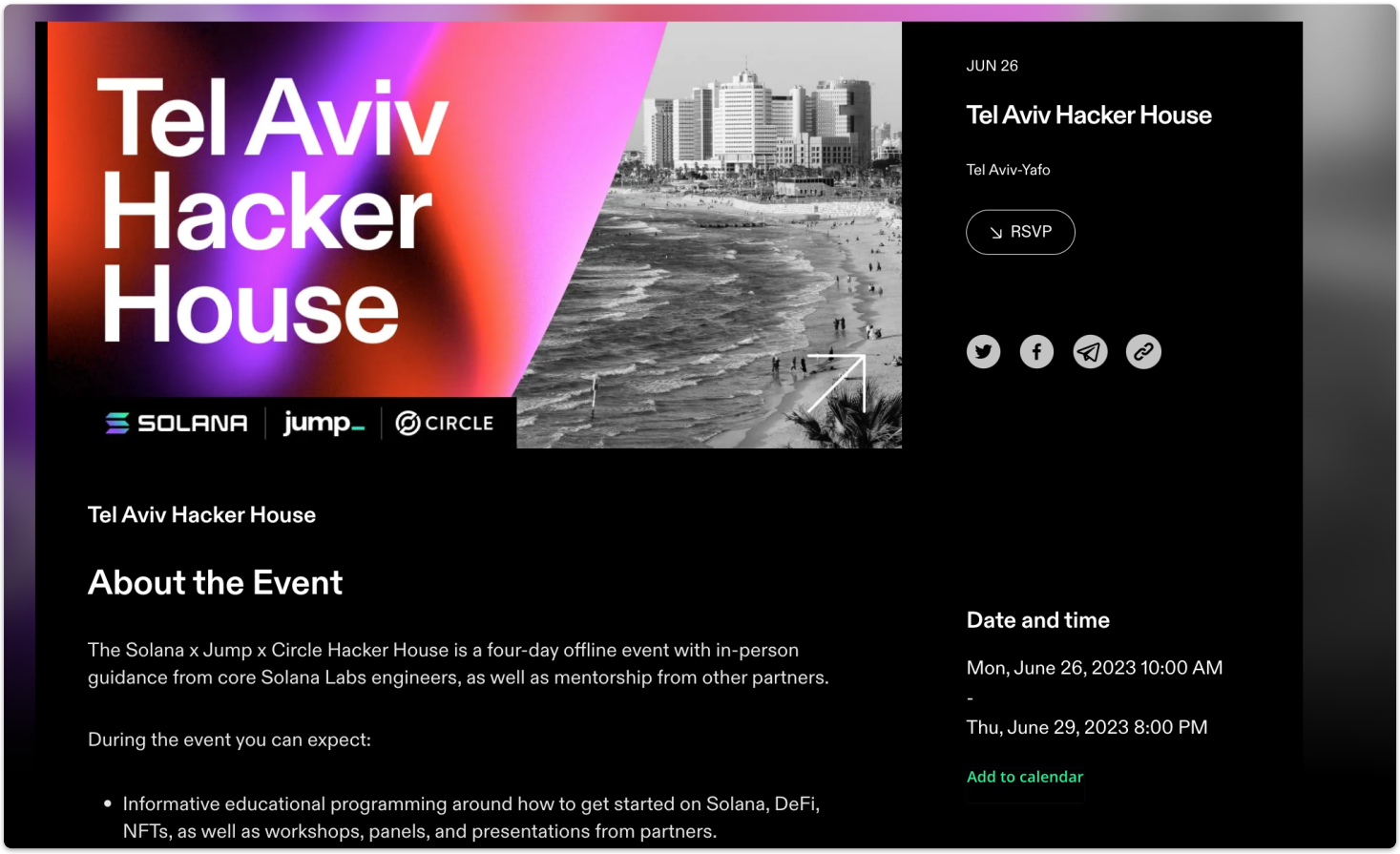
Lesson 5 - Rust - ownership

This week

- Rust
- Solana Development
- Solana Programs
- Token Program



Rust continued

Lifetime constraints on references

```
  let x;
{
        let a = 2;
        ...
        x = &a;
        ...
}
assert_eq!(*x,2);
```

This is a problem because

1. For a variable `a` , any reference to `a` must not outlive `a` itself.
   The variable's lifetime must enclose that of the reference.
   => how *large* a reference's lifetime can be
   The lifetime of &a must not be outside the dots
2. If you store a reference in a variable `x` the reference's type must be good for the lifetime of the variable.
   The reference's lifetime must enclose that of the variable.
   => how *small* a reference's lifetime can be.

We should have

```
  let a = 2;
{
        let x = &a;
        assert_eq!(*x,2);
}
```

Slices

Slices are similar to arrays, but their length is not known at compile time. Instead, a slice is a two-word object, the first word is a pointer to the data, and the second word is the length of the slice. The word size is the same as usize, determined by the processor architecture eg 64 bits on an x86-64.
Slices can be used to borrow a section of an array, and have the type signature `&[T]`."

```
  use std::mem;

// This function borrows a slice
fn analyze_slice(slice: &[i32]) {
    println!("first element of the slice: {}", slice[0]);
    println!("the slice has {} elements", slice.len());
}

fn main() {
    // Fixed-size array (type signature is superfluous)
    let xs: [i32; 5] = [1, 2, 3, 4, 5];

    // All elements can be initialized to the same value
    let ys: [i32; 500] = [0; 500];

    // Indexing starts at 0
    println!("first element of the array: {}", xs[0]);
    println!("second element of the array: {}", xs[1]);

    // `len` returns the count of elements in the array
    println!("number of elements in array: {}", xs.len());

    // Arrays are stack allocated
    println!("array occupies {} bytes", mem::size_of_val(&xs));

    // Arrays can be automatically borrowed as slices
    println!("borrow the whole array as a slice");
    analyze_slice(&xs);

    // Slices can point to a section of an array
    // They are of the form [starting_index..ending_index]
    // starting_index is the first position in the slice
    // ending_index is one more than the last position in the slice
    println!("borrow a section of the array as a slice");
    analyze_slice(&ys[1 .. 4]);

    // Out of bound indexing causes compile error
```

```
    println!("{}", xs[5]);
}
```

See
[Arrays - TutorialsPoint]
[Arrays and Slices - RustBook]

Printing / Outputting

See Docs

There are many options

- `format!`: write formatted text to `String`
- `print!`: same as `format!` but the text is printed to the console (io::stdout).
- `println!`: same as `print!` but a newline is appended.
- `eprint!`: same as `print!` but the text is printed to the standard error (io::stderr).
- `eprintln!`: same as `eprint!` but a newline is appended.

We commonly use `println!`

The braces `{}` are used to format the output and will be replaced by the arguments
For example

```
println!("{} days", 31);
```

You can have positional or named arguments
For *positional* specify an index
For example

```
println!("{0}, this is {1}. {1}, this is {0}", "Alice", "Bob");
```

For *named* add the name of the argument within the braces

```
println!("{subject} {verb} {object}",
              object="the lazy dog",
              subject="the quick brown fox",
              verb="jumps over");
```

Packages, crates and modules

A `crate` is a binary or library, a. number of these (plus other resources) can form. a `package` , which will contain a *Cargo.toml* file that describes how to build those crates.
A crate is meant to group together some functionality in a way that can be easily shared with other projects.
A package can contain at most one library crate, but. any number of binary crates.
There can be further refinement with the use of `modules` which organise code within a crate and can specify the privacy (public or private) of the code.
Module code is private by default, but you can make definitions public by adding the `pub` keyword.

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);`

Hashmaps

See Docs
and examples

Where vectors store values by an integer index, `HashMap`s store values by key. `HashMap` keys can be booleans, integers, strings, or any other type that implements the `Eq` and `Hash` traits.

Like vectors, `HashMap`s are growable, but HashMaps can also shrink themselves when they have excess space.
You can create a HashMap with a certain starting capacity
using `HashMap::with_capacity(uint)`, or use `HashMap::new()` to get a HashMap with a default initial capacity (recommended).

An example inserting values

```
  use std::collections::HashMap;
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

You can use an iterator to get values from the hashmap

```
  let mut balances = HashMap::new();

balances.insert("132681", 12);
balances.insert("234987", 9);


for (address, balance) in balances.iter() {
        ...
}
```

Using the get method

The get method on a hashmap returns an `Option<&V>` where V is the type of the value

```
  fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    let team_name = String::from("Blue");
    let score = scores.get(&team_name).copied().unwrap_or(0);
}
```

This program handles the `Option` by calling `copied` to get an `Option<i32>` rather than an `Option<&i32>`, then `unwrap_or` to set `score` to zero if `scores` doesn't have an entry for the key.

Rust - (more) pattern matching

See Docs

Ignoring values and matching literals

We can use `_` to show we are ignoring a value, or to show a default match, where we don't care about the actual value.

```language-rust
      let x = ...;

      match x {
          1 => println!("one"),
          3 => println!("three"),
          5 => println!("five"),
          _ => println!("some other value"),
      }
```

Ranges

Use `..=`
For example

```
  let x = 5;
match x {
        1..=5 => println!("one through five"),
        _ => println!("something else"),
}
```

Variables

Be aware the match starts a new scope, so if you use a variable it may shadow an existing variable.

```
  let x = Some(5);
let y = 10;
match x {
        Some(50) => println!("Got 50"),
        Some(y) => println!("Matched, y = {y}"),
        _ => println!("Default case, x = {:?}", x),
}
println!("at the end: x = {:?}, y = {y}", x);
```

What we get printed out is
`Matched, y = 5`.
`x = Some(5), y = 10`.

Destructuring

We can destructure structs and match on their constituent parts

For example

```
  fn main() {
let p = Point { x: 0, y: 7 };
match p {
        Point { x, y: 0 } => println!("On the x axis at {}", x),
        Point { x: 0, y } => println!("On the y axis at {}", y),
        Point { x, y } => println!("On neither axis: ({}, {})", x, y), }
}
```

Adding further expressions

```language-rust
  let num = Some(4);
  match num {
          Some(x) if x % 2 == 0 => println!("The number {} is even", x),
          Some(x) => println!("The number {} is odd", x),
          None => (),
  }
```

Writing tests in Rust

See Docs
See examples

A test in Rust is a function that's annotated with the `test` attribute

To change a function into a test function, add `#[test]` on the line before `fn`.
When you run your tests with the `cargo test` command, Rust builds a test runner binary that runs the annotated functions and reports on whether each test function passes or fails.

```
  #[cfg(test)] mod tests {
#[test]
fn simple_example() {
        let result = 3 + 5;
        assert_eq!(result, 8);
        }
}
```

The `#[cfg(test)]` annotation on the tests module tells Rust to compile and run the test code only when you run `cargo test`, not when you run `cargo build`.