

# Lesson 5 - Rust - ownership

## This week

---

Mon : Rust

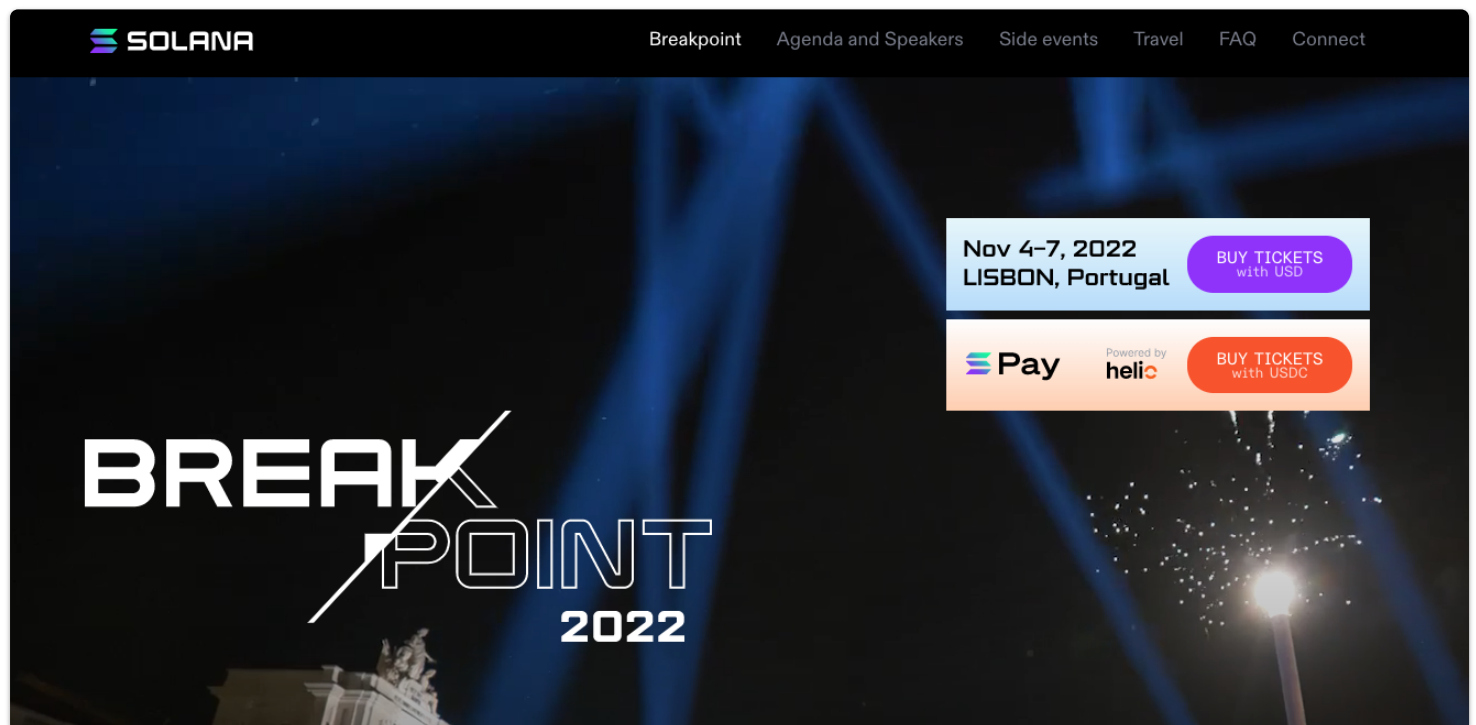
Tues : Solana Accounts / Solana development

Weds : Rust

Thurs : Token Program

## Solana News

---



The banner features a dark background with blue light beams and a cityscape at night. A large, stylized 'BREAKPOINT 2022' logo is on the left. On the right, there are two ticket purchase buttons: one for USD and one for USDC.

**SOLANA** Breakpoint Agenda and Speakers Side events Travel FAQ Connect

**BREAKPOINT 2022**

Nov 4-7, 2022  
LISBON, Portugal

BUY TICKETS with USD

Pay Powered by helio

BUY TICKETS with USDC

## RELATED EVENTS

[SUBMIT SIDE EVENT](#)[SEE ALL SIDE EVENTS](#)

### Solana x Jump Hacker House

Vibe. Mint. Build. Get ready for Breakpoint by meeting developers and founders, coding, and learning from core Solana engineers.

[REGISTER HERE](#)

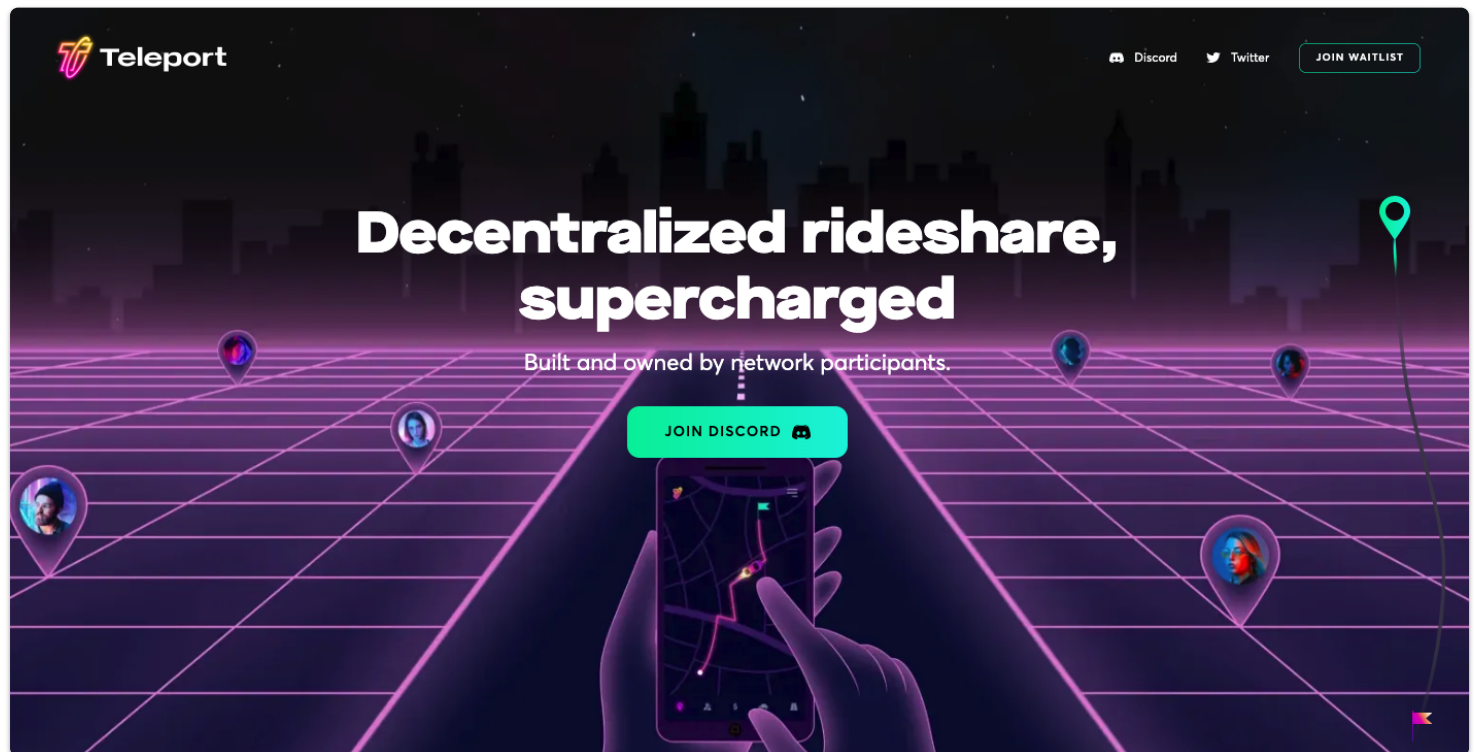
### Solana Foundation Games Day

Let's go! See how game companies have been building on Solana — and you can play a few yourself at this limited-capacity, community event.

[RSVP](#)

## Teleport

Teleport, a decentralized ride-sharing application planned to be launched in December. Payments will be allowed in stablecoins.



## 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.

---

## 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);
```

## 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.

```
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 iteration, 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);
```

---



## 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!("{}", 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");
```