Borrowing

We have already discussed borrowing in chapter 4 while discussing operators. Let's discuss it again in terms of ownership.

We'll cover the following

What is Borrowing?

Borrowing, in simple terms, means to share.

Let’s look at a real-life analogy to explain this concept. You are the owner of a book, but you allow other people to use it.

Types of Borrowing

In Rust, borrowing can be of two types:

Shared Borrows

The ownership belongs to the assignee variable but the assigned variable can only read the value.

> Multiple variables can borrow the value of the variable at the same time.

```
let a = 1;
let b = &a;
```

1 of 2

```
let a = 1;
let b = &a;
```

2 of 2

Mutable Borrows

The ownership belongs to the assignee variable but the assigned variable can share as well as mutate the value of owner variable.

> Only one variable in the scope can borrow mutably.

After the mutable borrow operation, the value of the mutably borrowed variable is moved to become inaccessible.

```
let a = 1;
let b = &mut a;
```

```
let a = 1;

let b = &mut a;
```


Rules of Borrowing

There are two rules to referencing or borrowing variables.

Rule # 1

There can be either one mutable borrow or any number of immutable borrows within the same scope.

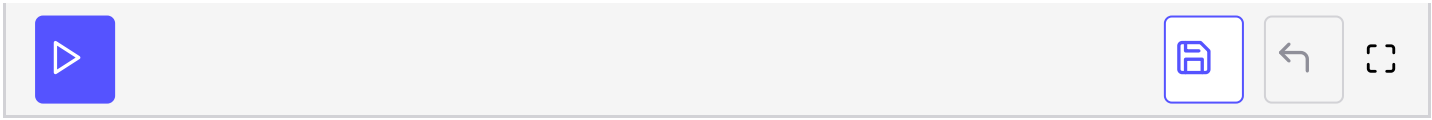
> It is not possible to do a shared borrow as well as a mutable borrow operation simultaneously in the same scope. If you want to do this in the same program code, then enclose a block of code within `{}`. In the inner block perform the shared borrow, and in the outer block perform the mutable borrow. Vice versa, outer block perform the shared borrow, and in the inner block perform the mutable borrow.

Example

The following example explains rule # 1.

```
/// cannot mutable borrow b since its already a shared borrow
/// mutable borrow a in outer scope and shared borrow in inner scope
fn main() {

  let mut a = 1; // mutable variable a is defined
  println!("variable `a` :{}", a);
  let b = 1;
  println!("variable `b` :{}", b);
  {

      let r1 = &a; // no problem
```

```
        println!("variable `r1` references `a` in inner scope(SHARED BORROW(a)) :{}",r1);
        let r2 = &a; // no problem
        println!("variable `r1` references `a` in inner scope(SHARED BORROW(a) :{}",r2);

        println!("r1:{}\nr2:{}", r1, r2);
        // r1 and r2 scope end here
    }

    let r3 = &mut a; // no problem
    *r3 = 3;
    println!("variable `r1` references `a` in outer scope(MUTABLE BORROW(a) and derefernced it and cl
    let r4 = &b;
    println!("variable `r3` references `b` in outer scope(SHARED BORROW(b)) :{}",r4);
    let r5 = &b;
    println!("variable `r3` references `b` in outer scope(SHARED BORROW(b)) :{}",r5);
    println!("r3:{}\nr4:{}\nr5:{}", r3 , r4 , r5);
}
```



Within the `main` function there is another block of code.

- On **line 5 and line** 7, mutable variables `a` and `b` are defined.
- On **line 11**, a variable `r1` makes a **shared borrow** operation with variable `a` within the inner block.
- On **line 13**, a variable `r2` makes a **shared borrow** operation with variable `a` within the inner block.
- On **line 19**, variable `r3` makes a **mutable borrow** operation with variable `a` outside the inner block.

> Note that this does not violate rule 1 since variable r1 and r2 are out of scope.

- On **line 22**, a variable `r4` makes a **shared borrow** operation with variable `b` within the outer block.
- On **line 24** a variable `r5` makes a **shared borrow** operation with variable `b` within the outer block.

Rule # 2

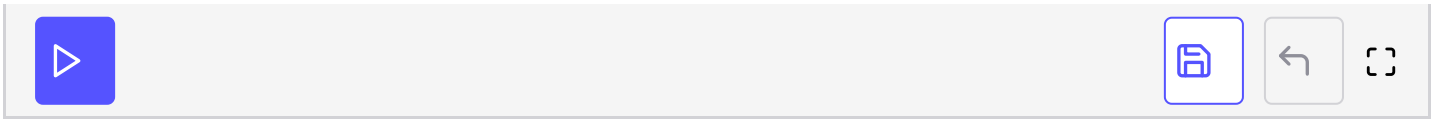
References must always be valid.

> Cannot reference a value that is moved, i.e., a non-primitive data type.

Example

The following example explains rule # 2.

```
fn main() {

  let a = String::from("Rust"); //variable a

  println!("This is a variable a: {}", a);

  let b = a; // moves value of a to b

  println!("Value of variable a is moved to b.\n b : {}", b);
  println!("Now a becomes invalid.Accessing a will give error");

  //let c = &a;
  //println!("This is a variable c trying to access value a: {}", c);
}
```



- On **line 3**, variable `a` is defined as type `String`.
- On **line 5**, the value of **a is moved to** variable `b`.

> Now `a` becomes inaccessible and cannot be accessed. If you uncomment the lines 12 and 13 you'll get a compiler error, ✕.

Functions and Borrowing

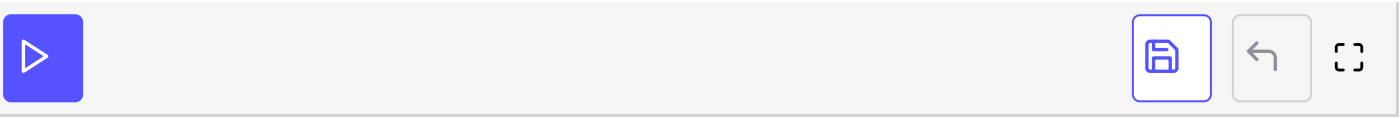
Recall pass by reference in which `& mut` was used as a function parameter when mutating values inside the function.

Example

The following example declares a function `example` which takes an owner variable, shared borrow and a mutable borrow as arguments to the function.

```
// 'a' an owner variable
// 'b' a shared borrow
// 'c' a mutable borrow
fn example(a: i32, b:& i32,c : &mut i32){
  println!("a: {}, b: {}, c: {}", a , b , c);
   *c=9;
}
fn main(){
```

```
fn main(){
   let a = 1;
   let b = 2;
   let mut c = 3;
   example( a, &b , &mut c);
   println!("a: {}, b: {}, c: {}", a , b , c);
}
```



In the above example, a function `example` is declared which takes *three parameters: an owner variable, a shared borrow variable, and a mutable borrow*.

- On **line 12**, the function is invoked with arguments `a` passed by value, `b` passed as a shared borrow and `c` a mutable borrow, all of type integer respectively.
- On **line 6**, the value of `c` is mutated.
- After the function call, the values of `a`, `b` and **updated value of** `c` are printed

Borrowing and Slicing

It is possible to borrow a slice of an array, vector or string. Recall the syntax of slicing. It used an `&` before the name of the variable to be borrowed.

```
let arr:[i32;4] = [1, 2, 3, 4];
let borrow_a = &arr[0..2];

let str=String::from("Rust Programming");
let borrow_str = &str[0..2];

let my_vec = vec![1, 2, 3, 4, 5];
let borrow_vec = &my_vec[0..2];
```

Here `&` indicates a **shared borrow**.

Example

The following examples revise your concept of slicing in arrays, strings, and vectors respectively.

```
fn main() {
  let arr:[i32;4] = [1, 2, 3, 4]; // define an array
  let borrow_arr = &arr[0..2]; // slice an array

  println!("arr : {:?}", arr); // print the array
  println!("sliced arr : {:?}", borrow_arr); // print the sliced array
```

```
  let str = String::from("Rust Programming"); // define a String object
  let borrow_str = &str[0..2]; // slice the String object

  println!("str : {:?}", str); // print the String Object
  println!("sliced_str : {:?}", borrow_str); // print the sliced String

  let my_vec = vec![1, 2, 3, 4, 5]; // define a vector
  let borrow_vec = &my_vec[0..2]; // slice the vector

  println!("vec: {:?}", my_vec);  // print the vector
  println!("sliced_vec : {:?}", borrow_vec); // print the sliced vector
}
```

Quiz

Test your understanding of borrowing in Rust.

Quick Quiz on Borrowing!

1 Can a variable have a shared borrow or mutable borrow reference within the same scope?

2 Why does this code give an error?

```
fn main() {
  let a = String::from("Rust");
  let b = a;
  let c = &b;
  println!("println!{}", a);
}
```

3

Why does this code give an error?

```
fn main() {
  let a = String::from("Rust");
  let b = a;
  let c = &a;
}
```

Retake Quiz

Now that you have learned about referencing a variable using the borrowing concept, let’s learn about lifetimes in the next lesson.