Rust for Safety

References & Borrowing

What you can own, you can borrow

Ownership provides a solid semantic base, but is for values. Reuse of data after a function call is not possible with ownership if the called function doesn’t return the ownership to the value again.

References

```rust
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32
}

fn main() {
    let mut p = Point { x: 1, y: 2 }; ❶
    inspect(&p); ❸
    p.x = 2; ❹
    inspect(&p);
    ❺
}

fn inspect(p: &Point) { ❷
    println!("{:?}", p);
}
```

❶ Just a normal stack allocation

❷ `inspect` takes a reference using & instead of a bare value

❸ The call to `inspect` also needs a & to reference the stack value

❹ In between the two calls, we can modify the value

❺ Deallocation point

Immutable references

`&` is the so-called "immutable" reference. They are:

- Available multiple times
- Always valid (always pointing to living data)
- Never `null`
- Guaranteed to never observe mutation of the pointed value

Modifying immutable references

```
fn main() {
    let mut point = Point { x:
1, y: 2 };
    let ref = &point; // -\ ❶
    point.x = 2;      //  | ❷
    inspect(ref);     // -/ ❸
}
```

❶ reference is taken here.

❷ mutation happens here.

❸ because the reference is still alive, it would observe mutation

Error

```
error[E0506]: cannot assign to `point.x` because it is borrowed
  --> scratch.rs:10:5
   |
9  |     let reference = &point;
   |                     ------ borrow of `point.x` occurs here
10 |     point.x = 2;
   |     ^^^^^^^^^^^ assignment to borrowed `point.x` occurs here
11 |     inspect(reference);
   |             --------- borrow later used here

error: aborting due to previous error
```

Lingo

Immutable references *borrow immutably*.

Mutation

```
fn main() {
    let mut p = Point { x: 1,
y: 2 };
    inspect(&p); ❹
    move_point(&mut p,3,3); ❸
    inspect(&p); ❹
}

fn move_point(
    p: &mut Point, ❶ ❷
    x: i32, y: i32
) {
    p.x = x;
    p.y = y;
}
```

❶ Instead ot `&`, use `&mut` to *mutably borrow*.

❷ Mutable borrows are unique at any time in the program!

❸ Use `&mut` at the call site. This requires a mutable value!

❹ Immutable borrows still cannot observe mutation.

The Borrowing Rules

Values can be:

- Borrowed immutably as often as you’d like
- Or mutably exactly once
- The two rules are mutually exclusive.

Rust forbids *shared mutability*.

What does that save us from?

```
fn push_all(on: &mut Vec<u8>, from: &Vec<u8>) {

}
```

Dereferencing

```
fn main() {
    let number: &mut i32 = &mut 4;
    *number = 10;
    println!("{}", number);
}
```

Other kinds of borrows

```
struct ExampleIter<'iter, T> {
❷
    vec: &'iter Vec<T>, ❶
    pos: usize,
}

fn main() {
    let vec: Vec<u32> = vec!
[1,2,3]; ❹
    let iter: Iter<'_, u32> =
vec.iter(); ❸ ❹
    for i in iter {
        println!("{}", i);
    }
}
```

❶ Iterators carry an inner refernce to what they *iterate over*. They are invalid if that went away.

❷ Therefore, they carry a *lifetime*, to bind them to the value.

❸ Iterators are gained from what they iterate over.

❹ Both `Vec` and `Iter` are owned values!

Lingo

This iterators *borrows* the `Vec` it iterators over.

Let’s try to break it!

```
fn main() {
    let vec = vec![1,2,3];
    let iter = vec.iter(); ❶
    drop(vec); ❷
    for i in iter { ❸
        println!("{}", i);
    }
}
```

❶ creates an iterator over a vector.

❷ forcibly deallocates the vector.

❸ tries to iterate and would iterate over deallocated memory

Or, as `rustc` would say...

```
error[E0505]: cannot move out of `vec` because it is borrowed
  --> scratch.rs:11:10
   |
10 |     let iter: Iter<'_, u32> = vec.iter();
   |                               --- borrow of `vec` occurs here
11 |     drop(vec);
   |          ^^^ move out of `vec` occurs here
12 |     for i in iter {
   |              ---- borrow later used her
```

Summary

- The borrowing rules keep references safe
- They apply to values with inner references and references alike!
- Inner referencing behaviour is always appearant from the type signature
- *Owners* decide about the time values are in memory
- Rust does *never* reorder your code. It only points at its flaws.