References and Borrowing

- As referenced at the end of the ownership summary, we have a way to pass code into functions without losing ownership. We do this through **references**.

```
fn main(){
  let s1 = String::from("ok");
  let s2 = using_references(&s1); // ☆-1
}

fn using_references(s : &String) -> usize{ // ☆-2
  s.len();
}
```

- These ampersands (&) at `☆-1` and `☆-2` , denote that we are using a reference.
  - $Def^n /$ A **reference** is a "data type" that contains a pointer to the original variable.

> There's also something called *dereferencing* which is denoted by the asterisk operator ($*$) $\rightarrow$ more on this later.

- When a reference goes out of scope, all it happens is that it (the pointer) gets popped off the stack.
- References are function parameters $\equiv$ **borrowing**.
- References are **immutable** by default.
  - To make 'em mutable:

  ```
  fn main(){
    let s1 = String::from("ok");
    let s2 = using_mut_references(&mut s1); // ☆-1
  }

  fn using_mut_references(s : &mut String) -> usize { // ☆-2
    s.push_str(", ko");
  }
  ```

  > Note s1 is now mutable and we are also passing and receiving mutable references to the fxn.

- A couple of restrictions:
  - There can only be **one mutable reference per scope** (allows us to avoid *Data Races*)

  > $Def^n /$ **Data Race**:

> Two or more race pointers access the same data at the same time.
> At least one of the pointers is being used to write to the data.
> There's no mechanism being used to synchronize access to data.

- We cannot create a **mutable reference when an immutable references already exists**
- Not OK:

  ```
  let mut s = String::from("ok");
  let r = &s; // immutable ref
  foo(&mut s); // mutable ref
  ```

- Subtle not OK:

  ```
  let mut s = String::from("ok");
  let r = &s;
  s.push_str(", ko"); // mutable borrow occurs here
  ```

- OK:

  ```
  let r = &s;
  println!("r : {}", r); // r scope ends when it is last used
  s.push_str(", ko"); // this is ok
  ```

- To understand this, it is important to think of the heap and stack. Let's talk about it in steps:

1. Create `String`
   - Allocated heap space.
2. Created immutable reference to `String` .
3. Changed the value of `String` . For example, maybe it became bigger changing its' place in the heap.
4. PROBLEM! The reference was immutable and stored in the stack meaning its' length was fixed and we cannot modify it.
5. Rust compiler is nice and shows you that is a problem.

- We can create new scopes by encompassing our code by curly brackets.
  - Not OK:

```
let mut s = String::from("hello");
let r1 = &mut s;
let r2 = &mut s;
```

  - OK:

```
let mut s = String::from("hello");

{
    let r1 = &mut s;

} // r1 goes out of scope here, so we can make a new reference with no problems.

let r2 = &mut s;
```

- $Def^n /$ **Dangling references**: when a variable goes out of scope previous to its' reference (i.e. the reference ends up pointing to something else).
  - This will **never** happen in Rust due to its' design choices.

  ```
  fn main() {
      let r = dangle();
  }

  fn dangle() -> &String{
      let s = String::from("ok");
      &s
      /*
      returns a reference to a
      value that is about to be dropped
      a line earlier (i.e. reference
      will point to something completely
      unrelated).
      */
  }
  ```

  - Error will say: "this function's return type contains a borrowed value, but there is no value for it to be borrowed from."
  - Solution is to give ownership of the `s` variable to someone else.

Summary

- We can only have one mutable reference to the same variable per scope.
- We cannot have a mutable reference to a variable after an immutable reference to the same variable in the same scope.