# Introduction to Rust

Benjamin Corey

## Why Rust?

---

There are so many great things going for Rust, but here are a few of the main reasons:

- It is fast & compiles into a static binary.
- The compiler keeps you from writing certain classes of bugs
- It has a very expressive type system
- The package manager is fast and very familiar

## Runs anywhere!

---

- Microcontrollers!
- Web browsers!
- Servers!
- Desktop!
- Mobile!

In Rust:

```rust
fn my_func() -> Option<String> {
    None
}

fn main() {
    let foo: Option<String> = my_func();
    foo.len(); // won't compile!
}
```

[finished with error]

```
error[E0624]: method `len` is private
    --> /tmp/.presentermGhiz0L/snippet.rs:7:9
     |
   7 |     foo.len(); // won't compile!
     |         ^^^ private method
     |
    :::
/home/gvqz/.rustup/toolchains/1.92.0-x86_64-unknown-
linux-gnu/lib/rustlib/src/rust/library/core/src/opti
on.rs:809:5
     |
 809 |     const fn len(&self) -> usize {
     |     -------------------------- private
method defined here
     |
note: the method `len` exists on the type `String`
```

In Java:

```java
class Main {
    static String myFunc() {
        return null;
    }

    public static void main(String[] args) {
        String foo = myFunc();
        foo.length(); // fails at runtime!
    }
}
```

[finished with error]

```
Exception in thread "main"
java.lang.NullPointerException: Cannot invoke
"String.length()" because "<local1>" is null
    at Main.main(Snippet.java:8)
```

## Handling Options

In Rust:

```
fn my_func() -> Option<String> {
    None
}

fn main() {
    let foo: Option<String> = my_func();
     match foo {
        Some(s) => println!("{}", s.len()),
        None => println!("nothing!"),
    }
}
```

[finished]

```
nothing!
```

In Rust:

```rust
fn my_func() -> Result<String, String> {
    Err("Failed".to_string())
}

fn main() {
    let foo = my_func();
    match foo {
        Err(e) => println!("Error: {e}"),
        Ok(s) => println!("{s}"),
    }
}
```

In Java:

```java
class Main {
    static String myFunc() {
        throw Exception;
    }

    public static void main(String[] args) {
        String foo;
        try {
            foo = myFunc();
        } catch(Exception e) {
            return;
        }
        foo.length();
    }
}
```

## Types: Enums

---

Enums can have stuff in them!

```
enum TransitVehicle {
    Bus {
        capacity: u8,
    },
    Train {
        cars: u8,
        car_capacity: u8
    }
}
```

## Types: Structs

This is what you might think of as a class in another language.

```
// they can have fields
struct Car {
    year: u16,
    model: String,
}

// They can be zero-sized
struct Bollard;

// they can have 'tuple' fields
struct Map(HashMap<Point2D, Item>);

// they can have methods!
impl Map {
    fn place_car(&mut self, point: Point2D, car: Car) {...}
}
```

## Types: Numbers

---

We need to know what size everything is! all the time! Even integers and floats.

| Length | Signed | Unsigned |
|---|---|---|
| 8-bit | i8 | u8 |
| 16-bit | i16 | u16 |
| 32-bit | i32 | u32 |
| 64-bit | i64 | u64 |
| 128-bit | i128 | u128 |
| Architecture-dependent | isize | usize |

And let's not forget `f32` and `f64`.

## Types: Strings

---

`String` vs `str`

`str` is a slice of characters. It is a fixed size always - like a Java Array.

`String` is flexibily sized, like a Java ArrayList.

```rust
fn main() {
    {
        let x = 5;
    }

    println!("{x}");
}
```

[finished with error]

```
error[E0425]: cannot find value `x` in this scope
 --> /tmp/.presentermN2SSH7/snippet.rs:6:16
  |
6 |     println!("{x}");
  |                ^
  |
help: the binding `x` is available in a different scope in the same
function
 --> /tmp/.presentermN2SSH7/snippet.rs:3:13
  |
3 |         let x = 5;
  |             ^

error: aborting due to 1 previous error

For more information about this error, try `rustc --explain E0425`.
```

```rust
fn main() {
    let s = make_string();

    println!("{s}");
}

fn make_string() -> str {
    "hello"
}
```

[finished with error]

```
error[E0277]: the size for values of type `str` cannot be known at compilation time
 --> /tmp/.presentermCbyQI6/snippet.rs:7:21
  |
7 | fn make_string() -> str {
  |                     ^^^ doesn't have a size known at compile-time
  |
  = help: the trait `Sized` is not implemented for `str`
  = note: the return type of a function must have a statically known size

error[E0277]: the size for values of type `str` cannot be known at compilation time
 --> /tmp/.presentermCbyQI6/snippet.rs:2:9
  |
2 |     let s = make_string();
  |         ^ doesn't have a size known at compile-time
  |
  = help: the trait `Sized` is not implemented for `str`
  = note: all local variables must have a statically known size
```

```rust
fn main() {
    let s = make_string();

    println!("{s}");
}

fn make_string() -> &str {
    "hello"
}
```

[finished with error]

```
error[E0106]: missing lifetime specifier
 --> /tmp/.presentermgdaDsh/snippet.rs:7:21
  |
7 | fn make_string() -> &str {
  |                     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is no value for it to
be borrowed from
help: consider using the `'static` lifetime, but this is uncommon unless you're returning a
borrowed value from a `const` or a `static`
  |
7 | fn make_string() -> &'static str {
  |                      +++++++
help: instead, you are more likely to want to return an owned value
  |
7 - fn make_string() -> &str {
7 + fn make_string() -> String {
```

```rust
fn main() {
    let s = make_string();

    println!("{s}");
}

fn make_string() -> &'static str {
    "hello"
}
```

[finished]

```
hello
```

```rust
fn main() {
    let s = make_string();

    println!("{s}");
}

fn make_string() -> &'static str {
    "hello"
}
```

[finished]

```
hello
```