Error Handling and Debugging

Zachary Bennett
Lead Software Developer

@z_bennett_ | zachbennettcodes.com

Rust Error Concepts

Recoverable Errors

A Rust program should recover from these errors in some way and alert the user

Unrecoverable Errors

In case of these errors, the Rust program should immediately terminate

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

The Result Type

The Result type allows calling code to appropriately handle any potential recoverable errors that might occur.

Recovering from Errors with the Result Type

```
let open_file_result = File::open(“test.txt”);

match open_file_result {
    Ok(testFile) => println!(“We successfully opened the file!”),
    Err(err) => println!(“Can’t open this file! Reason: {:?}”, err)
}
```

Unrecoverable Errors and “panic!”

```
let open_file_result = File::open(“test.txt”);

match open_file_result {
    Ok(testFile) => println!(“We successfully opened the file!”),
    Err(err) => panic!(“Can’t open this file! Reason: {:?}”, err)
}
```

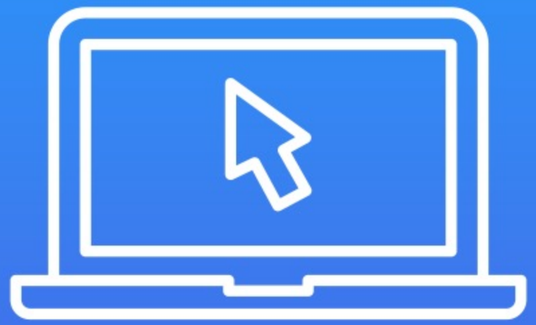
Use panic! when you don’t want to give the calling code/user a chance to recover!

Propagating Errors with the Question Mark Operator

```
fn multiply_str_by_ten(a: str&) -> Result<i32, ParseIntError> {
    let parsed_a: i32 = a.parse()?; // See how this unwraps the value?
    parsed_a * 10
}

multiply_str_by_ten(“100”); // Returns a Result that is Ok(1000)
multiply_str_by_ten(”abc”); // Returns an Error! The “?” operator propagates it
```

Demo



Handling Errors

- Recoverable errors
- Unrecoverable errors and “panic!”
- Using the “?” operator

Debugging in VSCode