RETURN VALUES VS. EXCEPTIONS

RETURN VALUES

Pros:

- Simplicity
- The set of possible return values is well defined
- Responsibility for error handling is clear

Cons:

- Errors need to be handled immediately
- Manual propagation of errors up the call stack
- Can be verbose and repetetive
- Can easily be ignored (in most languages)

EXCEPTIONS

Pros:

- Automatic propagation up the call stack
- Generalized classes of Errors
    - Defer handling of errors up the call stack
- Distinguish between exceptional errors and expected errors

Cons:

- Set of possible exceptions to handle unclear
  - Caller required to handle errors thrown x levels down the call stack
  - Can change due to changes in other parts of the code
- Can easily be ignored (in some languages)

ALTERNATIVES (NOT COVERED IN THIS TALK)

- Global Error Variables
- Signals
- Hardware Interrupts
- setjmp

RESULT< T, E >

A GENERIC ENUM WITH TWO VARIANTS

```rust
pub enum Result<T, E> {
    /// Contains the success value
    Ok(T),
    /// Contains the error value
    Err(E),
}
```

OPTION< T >

A GENERIC ENUM WITH TWO VARIANTS

```
pub enum Option<T> {
    /// No value
    None,
    /// Some value `T`
    Some(T),
}
```

PATTERN MATCHING

MATCH

```
fn check_answer(answer: Result<u32, String> ) {
    match answer {
        Ok(i) => println!("Success - {} is the answer!", i),
        Err(s) => println!("Error: {}", s)
    }
}
```

- Match cases called "arms"
- All possible states need to be covered
- Use _ as catch-all

IF LET

```
if let Err(s) = do_ask_propagate_error(answer) {
    println!("Error: {}", s);
}
```

- Use `if let` are only interested in *one* arm needs to be matched

SHORTCUTS

? - FOR PROPAGATING ERRORS

From:

```rust
fn propagate_error(answer: Result<u32, String> ) -> Result<(),
    match answer {
        Ok(i) => {
            println!("Success - {} is the answer!", i);
            Ok(())
        },
        Err(s) => Err(s)
    }
}
```

To:

```rust
fn propagate_error(answer: Result<u32, String> ) -> Result<(),
    println!("Success - {} is the answer!", answer?);
    Ok(())
}
```

CONVERTING ERROR TYPES

Use `map_err` if the Error return type mismatches:

```
enum AskError {
    NeedToWait
}

fn do_ask_propagate_error(answer: Result<u32, String> ) -> Res
    println!("Success - {} is the answer!",
        answer.map_err(|_| AskError::NeedToWait)?);
    Ok(())
}
```

UNWRAP - EVIL!

May be ok in unit tests, but shunned in production code - will panic and crash if the Result is of the Err variant!

```
let x: u32 = answer.unwrap();
```

EXPECT - EVIL!

May be ok in unit tests, but shunned in production code - will panic and crash if the Result is of the Err variant!

```rust
let x: u32 = answer.expect("Answer must be valid!");
```

UNWRAP_OR - LESSER EVIL!

Returns the value or, if the Result is of Err variant, the given default value.

```
let x: u32 = answer.unwrap_or(default_value);
```

CONVERTING RESULT TO OPTION

```
fn result_to_option(answer: Result<u32, String>) -> Option<u32
    answer.ok()
}
```

CONVERTING OPTION TO RESULT

```
fn option_to_result(answer: Option<u32>) -> Result<u32, AskErr
    answer.ok_or(AskError::NeedToWait)
}
```

QUESTIONS?