Error Handling

panic!

```rust
// Similar to throwing exception
panic!("Crash and burn");
```

```
 Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 1.34s
     Running `target/debug/playground`
thread 'main' panicked at 'Crash and burn', src/main.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
   0: rust_begin_unwind
             at /rustc/90743e…e29bfeb6/library/std/src/panicking.rs:575:5
   1: core::panicking::panic_fmt
             at /rustc/90743e…e29bfeb6/library/core/src/panicking.rs:65:14
   2: playground::main
             at ./src/main.rs:2:5
   3: core::ops::function::FnOnce::call_once
             at /rustc/90743e…4e29bfeb6/library/core/src/ops/function.rs:251:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

Option<T> and Result<L, R>

```rust
enum Option<T> {
    None,
    Some(T),
}
```

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


unwrap() and expect()

- Can be applied to any function that returns either `Option<T>` or `Result<T, E>`.

```rust
// Idealist's dream
let f = File::open("hello.txt").unwrap(); // panics if Err returns

// Describe the reason you expect the Result should be Ok.
// If Err returns, panics with the given message.
let f = File::open("hello.txt").expect("Fail to open hello.txt");
```

Pattern Match

```
let f: Result<File, io::Error> = File::open("hello.txt");

let f = match f {
    Ok(file) => file,
    Err(error) => panic!("Problem opening the file: {:?}", error),
};
```


Nested Pattern Match

```
let f = File::open("hello.txt");

let f = match f {
    Ok(file) => file,
    Err(error) =>
        match error.kind() {
            ErrorKind::NotFound =>
                match File::create("hello.txt") {
                    Ok(fc) => fc,
                    Err(e) => panic!("Problem creating the file: {:?}", e),
                }
            other_error => panic!("Problem opening the file: {:?}", other_error),
        }
};
```

unwrap_or || unwrap_or_else

```
fn find(vs: &[i32]) -> Result<i32, String> {
    todo!("Find the first non-negative number in the slice");
}

fn main() {
    let vs = vec![-1, -2, -3, -4, -5];
    let value = find(&vs).unwrap_or(0);
    // ...
}
```

```
let f = File::open("hello.txt").unwrap_or_else(|error| {
    if error.kind() == ErrorKind::NotFound {
        File::create("hello.txt").unwrap_or_else(|error| {
            panic!("Problem creating the file: {:?}", error);
        })
    } else {
        panic!("Problem opening the file: {:?}", error);
    }
});
```


Error Propagation

```
pub fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => {
            return Err(e);
        }
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

Error Propagation Shortcut

In std::io module

```
type Result<T> = Result<T, io::Error>
```

```
pub fn read_username_from_file() -> Result<String> {
    // if error, returns error from current function
    // for caller to handle
    let mut f = File::open("hello.txt")?;

    let mut s = String::new();
    f.read_to_string(&mut s)?; // same here!
    Ok(s)
}
```


Error Propagation Shortcut (Cont'd)

```
type Result<T> = Result<T, io::Error>
```

```
pub fn read_username_from_file() -> Result<String> {
    let mut s = String::new();
    let f = File::open("hello.txt")?.read_to_string(&mut s)?;
    Ok(s)
}
```

? shortcut and `std::error::Error` Trait

- For ? shortcut, you must implement `std::error::Error`.
- Requirement for `std::error::Error` types: any type that implements `Error` also has to implement both:
  - the `Display` trait, meaning that it can be format!ed with `{}`, and
  - the `Debug` trait, meaning that it can be format!ed with `{:?}`.
- Prefer to implement `Error` trait or your error types.
  - If you don't use ? shortcut, the `E` type parameter for a `Result` doesn't have to be a type that implements `Error`.
- Currently, the only method in the trait is `source()`, which allows an `Error` type to expose an inner, nested error: default is `None`.

Error Handling Crates

- **thiserror** provides a convenient derive macro for the standard library's `std::error::Error` trait.

- **anyhow** provides `anyhow::Error`, a trait object based error type for easy idiomatic error handling in Rust applications.