Error Handling

Error handling patterns

- Ignore the error
- Terminate the program
- Use a fallback value
- Propagate the error
- Propagate the error multiple errors
- Match boxed errors
- Create custom errors

Ignore the error

- Prototyping stage

```
use std::fs;

fn main() {
  let content = fs::read_to_string("./Cargo.toml").unwrap();
  println!("{}", content)
}
```

- If it returns an error, it will “panic”. Panic either terminates the program or exits the current thread.

Terminate the program

- Some errors cannot be handled or recovered from. In these cases, it’s better to fail fast by terminating the program.

```
use std::fs;
fn main() {
  let content = fs::read_to_string("./Cargo.toml").expect("Can't read Cargo.toml");
  println!("{}", content)
}
```

Use a fallback value

- In some cases, you can handle the error by falling back to a default value.

```
use std::env;

fn main() {
  let port = env::var("PORT").unwrap_or("3000".to_string());
  println!("{}", port);
}
```

Propagate the error

- When you don’t have enough context to handle the error, you can propagate the error to the caller function.

Propagate the error

```
use std::collections::HashMap;

fn main() {
  match get_current_date() {
    Ok(date) => println!("We've time travelled to {}!!", date),
    Err(e) => eprintln!(“Good grief, we don't know which era we're in! :( \n  {}", e),
  }
}

fn get_current_date() -> Result<String, reqwest::Error> {
  let url = "https://postman-echo.com/time/object";
  let res = reqwest::blocking::get(url)?.json::<HashMap<String, i32>>()?;
  let date = res["years"].to_string();

  Ok(date)
}
```

Propagate multiple errors

- In the previous example, the get and json functions return a reqwest::Error error which we've propagated using the ? operator. But what if we've another function call that returned a different error value?

```
use chrono::NaiveDate;use std::collections::HashMap;

fn main() {
  match get_current_date() {
    Ok(date) => println!("We've time travelled to {}!!", date),
    Err(e) => eprintln!(“Good brief, we don't know which era we're in! :( \n  {}", e),
  }
}

fn get_current_date() -> Result<String, Box<dyn std::error::Error>> {
  let url = "https://postman-echo.com/time/object";
  let res = reqwest::blocking::get(url)?.json::<HashMap<String, i32>>()?;
  let formatted_date = format!("{}-{}-{}", res["years"], res["months"] + 1, res["date"]);
  let parsed_date = NaiveDate::parse_from_str(formatted_date.as_str(), "%Y-%m-%d")?;
  let date = parsed_date.format("%Y %B %d").to_string();

  Ok(date)
}
```

Match boxed errors

- So far, we've only printed the errors in the main function but not handled them. If we want to handle and recover from boxed errors, we need to "downcast" them:

```
fn main() {
   match get_current_date() {
     Ok(date) => println!("We've time travelled to {}!!", date),
    Err(e) => {
      if let Some(err) = e.downcast_ref::<reqwest::Error>() {
        eprintln!("Request Error: {}", err)
      } else if let Some(err) = e.downcast_ref::<chrono::format::ParseError>() {
        eprintln!("Parse Error: {}", err)
     }
    }
   }
 }
```

Custom Erorrs

- For library code, we can convert all the errors to our own custom error and propagate them instead of boxed errors. In our example, we currently have two errors - reqwest::Error and chrono::format::ParseError. We can convert them to MyCustomError::HttpError and MyCustomError::ParseError respectively.

- The Error trait requires us to implement Debug and Display traits:

```
use std::fmt;

#[derive(Debug)]
pub enum MyCustomError {
  HttpError,
  ParseError,
}


impl fmt::Display for MyCustomError {
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    match self {
      MyCustomError::HttpError => write!(f, "HTTP Error"),
      MyCustomError::ParseError => write!(f, "Parse
Error"),
    }
  }
}
```