# 9. error handling

## 에러 처리

```
Result<T, E>
```

- 위의 타입은 복구 가능한 에러를 위한 타입임.
- 복구 불가능한 에러가 발생하는 경우 `panic!` 매크로를 호출함.

## `panic!` 으로 복구 불가능한 에러 처리하기

- C언어의 `perror`처럼 에러를 일으킴.
- Rust의`RUST_BACKTRACE` 설정을 통해 상세하게 출력을 하는 정도가 달라짐.

## `Result`로 복구 가능한 에러 처리하기

- `Result<T,E>`는 제네릭 매개변수 T와 E를 가지고 있음.
- 리턴 타입으로 주로 주어지며 성공시 `OK(T)`를 반환하고 에러 발생시 `Err(E)`를 반환하는 열거형임.

```rust
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        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);
            }
        },
    };
}
```

- 위의 코드와 함께 이해해보자
  - `hello.txt`를 열 수 있었다면 greeting_file에는 `Ok(file)`이 저장되었을 것이다.
  - 그렇지않으면 `Err(error)`가 된다.
    - `error.kind`는 `io:Error` 타입을 가져온다.
    - `io::Error`타입의 `io::ErrorKind`값을 가져오기위해 kind method를 호출한다.
    - `ErrorKind::NotFound`는 열고자 하는 파일이 존재하지 않음을 의미함.
    - `other_error`는 그 외에 다른 오류임을 의미한다.
- `unwrap_or_else` method ?

- 위의 중첩 match 대신에 사용하기 위한 메서드이다.
- `Result<T, E>`와 match 대신 아래처럼 사용이 가능하다.
- 아래를 보면 `unwrap_or_else` 를 이용하여 조건문을 통해 에러를 처리하고 있다.
- 자세한 내용은 해당 메서드를 더 공부해야 할 것.

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file =
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);
        }
    });
}
```

## 에러 발생 시 패닉을 위한 unwrap과 expect

- `unwrap`메서드에 의해 에러 메세지를 출력하기 .

```
use std::fs::File;

fn main(){
        let greeting_file =
File::open("hello.txt").unwrap();
}
```

- 이제 문제가 생기면 `unwrap`이 `panic!`을 호출하여 ㅇ무엇이 문제인지에 대한 에러 메세지를 출력함.
- 이와 비슷하지만 에러 메세지를 직업 입력하는 형태로는 `expect`가 있음.

```
use std::fs::File;

fn main(){
        let greeting_file = File::open("hello.txt")
                .expect("hello.txt should be included in
this project");
}
```

## 에러 전파(propagating)하기

- 에러 전파란 호출하는 코드쪽에서 에러를 처리하도록 하는 것임.

```rust
use std::fs::File;
use std::io::{self, Read};

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

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

    let mut username = String::new();

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

- 위의 코드를 보면 Error가 발생하면 `panic!`을 사용하거나 하지 않고 에러 자체를 리턴해주는 것을 볼 수 있음.

## ? 연산자

- propagation을 위한 숏컷으로 에러가 발생하면 생성된 에러를 반환(함수에서 그냥 리턴해버림)해주고 그렇지 않으면 진행시킴

```rust
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("hello.txt")?;
    let mut username = String::new();
    username_file.read_to_string(&mut username)?;
    Ok(username)
}
```

- 더 짧게는 이렇게도 짤 수 있다고 함.

```rust
use std::fs;
use std::io;

fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

- 다만 이 `?`연산자는 main함수에서는 사용이 불가능함.
- main함수는 ()을 반환하는데 `?`연산자는 오류가 발생하면 `Err`를 리턴시켜버리기 때문임.
- 다시말하면 `?` 연산자는 반환 타입이 `Result` , `Option`, `FromResidual`인 함수만 가능함.
  참고

```rust
fn last_char_of_first_line(text: &str) -> Option<char>{
        text.lines().next()?.char().last()
}
```

- `lines`는 반복자로 만들어주는 것이고, `next`는 다음인자를 불러오는 것이므로 첫 번째 줄을 가져옴. 만약 문자열이 비었다면 None을 반환하기 때문에 Option으로서 리턴해줄 수 있음.

## 언제 panic!을 써야하는가?

- 현 상황이 복구 불가능한 상황이라고 판단되었을 때 사용함.
- 반면`Result`의 값을 사용한다는 건 호출하는 쪽에 옵션을 준다는 뜻임.
    - 실패할수도 있는 경우 `Result`를 사용하는게 일반적으로 좋은 선택임.