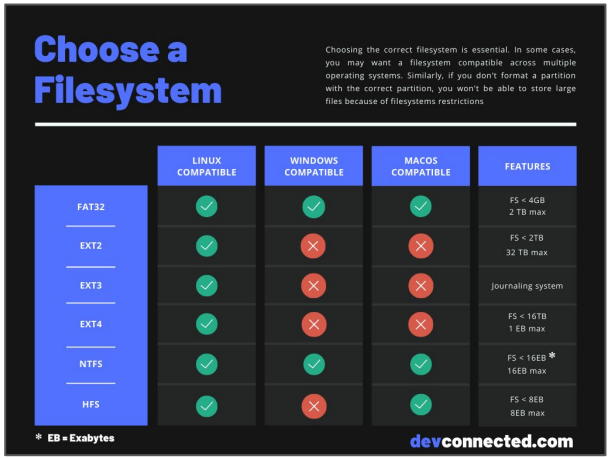
우아한 Enum

Option과 Result

Enumeration?

Enum의 역할

struct가 데이터의 집합이라면 enum은 “종류”의 집합



https://www.onbrix.co.kr/front/product/909
https://namu.wiki/w/%EB%8F%99%EC%A0%84
https://devconnected.com/how-to-format-disk-partitions-on-linux/

Enum의 역할

struct가 데이터의 집합이라면 enum은 “카테고리”의 집합

카테고리의 범위를 어떻게 정하느냐에 따라서 활용하는 것은 자유자재

- 과일의 종류
- 동전의 종류
- IP Address의 종류
- File system의 포맷 방식
- Rest API 호출 방식 (CRUD)

전통원시적인 Enum

C – Enum 따위 그냥 #define 아냐? 응 아냐

```c
#include <stdio.h>

enum DayOfWeek {
    Sunday = 0,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};

int main()
{
    enum DayOfWeek week;

    week = Tuesday;

    printf("%d\n", week);

    return 0;
}
```

```c
#include <stdio.h>

enum DayOfWeek {
    Sunday = 0,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};

#define Tuesday 0    // enum을 덮어써버림

int main()
{
    enum DayOfWeek week;

    week = Tuesday;

    printf("%d\n", week);

    return 0;
}
```

전통원시적인 Enum

전역적으로 그냥 선언되는 매크로의 집합정도로 여겨지던 시절

```
// Configuration flags stored in io.ConfigFlags. Set by user/application.
enum ImGuiConfigFlags_
{
    ImGuiConfigFlags_None                   = 0,
    ImGuiConfigFlags_NavEnableKeyboard      = 1 << 0,
    ImGuiConfigFlags_NavEnableGamepad       = 1 << 1,
    ImGuiConfigFlags_NavEnableSetMousePos   = 1 << 2,
    ImGuiConfigFlags_NavNoCaptureKeyboard   = 1 << 3,
    ImGuiConfigFlags_NoMouse                = 1 << 4,
    ImGuiConfigFlags_NoMouseCursorChange    = 1 << 5,
    ImGuiConfigFlags_IsSRGB                 = 1 << 20,
    ImGuiConfigFlags_IsTouchScreen          = 1 << 21,
};
```

https://github.com/ocornut/imgui/blob/master/imgui.h#L1465-L1479

전통적인 Enum

C++ – 그래도 할 건 하자

```cpp
#include <iostream>

enum DayOfWeek : uint8_t {
    Sunday = 0,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};

#define Tuesday 0    // 컴파일 에러

int main()
{
    DayOfWeek week;

    week = DayOfWeek::Tuesday;

    std::cout << static_cast<int32_t>(week) << std::endl;

    return 0;
}
```

```
ERROR!
/tmp/4uv366YFpL.cpp: In function 'int main()':
/tmp/4uv366YFpL.cpp:13:17: error: expected unqualified-id before numeric
constant
   13 | #define Tuesday 0    // 컴파일 에러
      |                 ^
/tmp/4uv366YFpL.cpp:19:23: note: in expansion of macro 'Tuesday'
   19 |     week = DayOfWeek::Tuesday;
      |                       ^~~~~~~


=== Code Exited With Errors ===
```

현대의 Enum

Java, Swift, Python(이게됨?) 등

```java
public enum TableStatus {
  Y("1", true),
  N("0", false);

  private String table1Value;
  private boolean table2Value;

  TableStatus(String table1Value, boolean table2Value) {
    this.table1Value = table1Value;
    this.table2Value = table2Value;
  }

  public String getTable1Value() {
    return table1Value;
  }

  public boolean isTable2Value() {
    return table2Value;
  }
}
```

```python
class AutoName(Enum):
    def _generate_next_value_(name, start, count, last_values):
        return name
class Ordinal(AutoName):
    NORTH = auto()
    SOUTH = auto()
    EAST = auto()
    WEST = auto()

>>> list(Ordinal)
[<Ordinal.NORTH: 'NORTH'>, <Ordinal.SOUTH: 'SOUTH'>, <Ordinal.EAST: 'EAST'>,
<Ordinal.WEST: 'WEST'>]
```

Rust의 Enum

Rust의 Enum

크게 다를바는 없지만 몇몇 현대 언어에서 지원하는 다중 타입 값 지원

- match와 if let 등 기초 문법 & syntax sugar도 지원

```rust
pub enum CellType {
  Int(i32),
  Float(f64),
  Text(String),
}

let row = vec![
  CellType::Int(3),
  CellType::Text(String::from("My Phone number")),
  CellType::Float(50.25),
];
```

```rust
match cell {
  CellType::Int => "It was an Integer",
  CellType::Float => "It was a Float",
  CellType::Text => "It was a Character sequence",
}
```

Enum과 찰떡인 match

match를 사용해서 condition 제어하기

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

```
fn plus_one(x: Option<i32>) -> Option<i32> {
  match x {
    None => None,
    Some(i) => Some(i + 1),
  }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

match의 간결한 버전 if let

Option이나 Result처럼 타입이 두개만 있다면 if let으로 간결하게

```
fn plus_one(x: Option<i32>) -> Option<i32> {
  match x {
    None => None,
    Some(i) => Some(i + 1),
  }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

```
fn plus_one(x: Option<i32>) -> Option<i32> {
  if let Some(i) = x {
    Some(i + 1)
  } else {
    None
  }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

Enum의 꽃은 Option과 Result

둘다 defensive programming의 기본.

사실상 impl에 있는 구현들이 더 중요하다.

```rust
/// The `Option` type. See [the module level documentation](
#[derive(Copy, PartialOrd, Eq, Ord, Debug, Hash)]
#[rustc_diagnostic_item = "Option"]
#[lang = "Option"]
#[stable(feature = "rust1", since = "1.0.0")]
#[allow(clippy::derived_hash_with_manual_eq)] // PartialEq i
pub enum Option<T> {
    /// No value.
    #[lang = "None"]
    #[stable(feature = "rust1", since = "1.0.0")]
    None,
    /// Some value of type `T`.
    #[lang = "Some"]
    #[stable(feature = "rust1", since = "1.0.0")]
    Some(#[stable(feature = "rust1", since = "1.0.0")] T),
}
```

```rust
/// `Result` is a type that represents either success ([`Ok`]) or failure
([`Err`]).
///
/// See the [module documentation](self) for details.
#[derive(Copy, PartialEq, PartialOrd, Eq, Ord, Debug, Hash)]
#[must_use = "this `Result` may be an `Err` variant, which should be
handled"]
#[rustc_diagnostic_item = "Result"]
#[stable(feature = "rust1", since = "1.0.0")]
pub enum Result<T, E> {
    /// Contains the success value
    #[lang = "Ok"]
    #[stable(feature = "rust1", since = "1.0.0")]
    Ok(#[stable(feature = "rust1", since = "1.0.0")] T),

    /// Contains the error value
    #[lang = "Err"]
    #[stable(feature = "rust1", since = "1.0.0")]
    Err(#[stable(feature = "rust1", since = "1.0.0")] E),
}
```

https://doc.rust-lang.org/src/core/option.rs.html#566-581
https://doc.rust-lang.org/src/core/result.rs.html#502-512

Option의 존재 이유는

null이어도 되는지 **한 번 더 고민**하고, 로직으로 대처 할 수 있도록 한다

```
struct Point {
  x: f32,
  y: f32,
}

fn try_match(x: Option<Point>) {
  match x {
    Some(p) => { println!("Co-ordinate: {}, {}", p.x, p.y);}
    _ => {panic!("No match");},
  }
}
```

Result의 존재 이유는

크래시가 나도 되는지, 처리가 가능한 부분인지 한 번 더 고민하고,
로직으로 대처 할 수 있도록 한다

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

fn main() {
    let res = File::open("result.csv");

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

Result의 존재 이유는

프로그램을 죽여야만 하는 경우, 그리고 self-healing 필요할 때가 다르다.

- System command라서 예기치 못한 입력은 피해야 하는 경우
  - panic!(“그거 뭐임? 몰?루”); 하고 죽는게 오히려 불필요한 side-effect를 줄일 수 있다.

- Inter-process communication 상황에서 하나의 프로세스가 로직이 꼬임
  - State machine 같은 상황에서는 되려 예기치 못한 상황에 죽기보단 재초기화, 연결 초기화 등을 통해서 다시금 시도할 수 있도록 하는 것이 더 안정적인 방법

꽤 유용한 Option과 Result의 함수들

- unwrap_or_else() – 그냥 unwrap()을 하다가 죽으면 panic!이기 때문에 그 다음 동작을 같이 정의할 수 있다. Result에는 unwrap_or_else는 없고 unwrap_or_default만 존재한다.

```
let k = 10;
assert_eq!(Some(4).unwrap_or_else(|| 2 * k), 4);
assert_eq!(None.unwrap_or_else(|| 2 * k), 20);
```

** 비슷하면서 다른 unwrap_or_default는 적용되는 경우가 조금은 더 특별하다고 봐야할 것 같다.
– 데이터베이스에 저장해야해서 차라리 panic!보다는 쓰레기 값이라도 쓰는게 나은 경우 등

꽤 유용한 Option과 Result의 함수들

- expect() – unwrap()을 시도하다가 panic!에 걸리면 메세지를 뱉어줌. 공식문서에서는 왜 Option, 또는 Result가 Some이어야 하는지 쓰는 것을 권장한다.

Recommended Message Style

We recommend that `expect` messages are used to describe the reason you *expect* the `Option` should be `Some`.

```
let item = slice.get(0)
    .expect("slice should not be empty");
```

```
let path = std::env::var("IMPORTANT_PATH")
    .expect("env variable `IMPORTANT_PATH` must be set`");
```

꽤 유용한 Option과 Result의 함수들

- expect_err() – 되려 에러가 나야하는 경우인데 error가 안날 때를 위해 존재하는 함수. expect()와 정확히 반대되는 동작이기도 하다.

```
let x: Result<u32, &str> = Err("emergency failure");
assert_eq!(x.unwrap_err(), "emergency failure");
```

꽤 유용한 Option과 Result의 함수들

- map() – 안에 있는 내용물을 바꿔치기 할 수 있게 해줌!

  이렇게 하면 굳이 안에 있는 값을 다른 변수로 옮기고 바꾸고 할 필요가 없어짐

```rust
let maybe_some_string = Some(String::from("Hello, World!"));
// `Option::map` takes self *by value*, consuming `maybe_some_string`
let maybe_some_len = maybe_some_string.map(|s| s.len());
assert_eq!(maybe_some_len, Some(13));

let x: Option<&str> = None;
assert_eq!(x.map(|s| s.len()), None);
```

```rust
let line = "1\n2\n3\n4\n";

for num in line.lines() {
    match num.parse::<i32>().map(|i| i * 2) {
        Ok(n) => println!("{n}"),
        Err(..) => {}
    }
}
```

Move를 항상 조심하세요

rust의 기본 철학은 move

```rust
struct Point {
  x: f32,
  y: f32,
}

fn try_match(x: Option<Point>) {
  match x {
    Some(p) => { println!("Co-ordinate: {}, {}", p.x, p.y);}
    _ => {panic!("No match");},
  }

  // 컴파일 에러
  if let Some(_p) = x {
    println!("Try again...");
  }
}

fn main()
{
    let p = Point{x: 32.0, y: 64.0};
    try_match(Some(p));
}
```

```
   Compiling playground v0.0.1 (/playground)
error[E0382]: use of moved value
  --> src/main.rs:12:15
   |
8  |     Some(p) => { println!("Co-ordinate: {}, {}", p.x, p.y);}
   |          - value moved here
...
12 |   if let Some(_p) = x {
   |               ^^ value used here after move
   |
   = note: move occurs because value has type `Point`, which does not
implement the `Copy` trait
help: borrow this binding in the pattern to avoid moving the value
   |
8  |     Some(ref p) => { println!("Co-ordinate: {}, {}", p.x, p.y);}
   |          +++

For more information about this error, try `rustc --explain E0382`.
error: could not compile `playground` (bin "playground") due to 1 previous
error
```

끗