Traits

- interface와 유사

```
trait Summary {
    fn summarize(&self) -> String;
    // 또는 아래와 같이 default 구현을 제공할 수 있음
    // fn summarize(&self) -> String {
    //     String::from("(Read more...)")
    // }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    } // 또는 summarize(&self)를 구현하지 않고 default 구현을 사용할 수 있음
}
```

Traits

- 특정 타입에 구현된 trait을 사용하려면 trait도 스코프에 가져와야 함

```
use aggregator::{Summary, Tweet};
...
    println!("1 new tweet: {}", tweet.summarize());
...
```

- orphan rule: trait이나 trait을 구현할 타입 중 하나는 현재 crate에 정의되어 있어야 함

Traits

- interface처럼, 매개변수 타입으로 사용 가능

```
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

- 반환 타입으로도 사용 가능

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        ...
    }
}
```

Trait bound

- 아래 두 코드의 차이점은?

```
pub fn notify(item1: &impl Summary, item2: &impl Summary) {
    ...
}
pub fn notify<T: Summary>(item1: &T, item2: &T) {
    ...
}
```

- 여러개의 trait을 사용하고 싶을 때는 +

Trait bound

- 조건부 메서드 구현

```
impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        ...
    }
}
```

Lifetime

- 대충 이런거 못하게 하는거

```
fn main() {
    let r;

    {
        let x = 5; <- 얘 수명이 r보다 짧음
        r = &x;
    }

    println!("r: {}", r);
}
```

Lifetime Parameter

- Lifetime 문제점: Borrow Checker는 sound하지만 complete하지 않다

```
// 상식적으로, 반환값의 수명은 x와 y의 수명 중 짧은 것. 하지만 borrow checker는 이를 알 수 없음
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

- 해결: lifetime parameter

```
// 그러니까 프로그래머가 명시해준다
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime Parameter

- 구조체에도 써보자

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let first_sentence = ...
    let i = ImportantExcerpt {
        part: first_sentence, // <- 얘가 살아있는 동안 i도 살아있어야 함
    };
}
```

Lifetime Elision

- rust 팀이 생각해보니 안전하게 추론할 수 있는 결정론적 패턴이 몇가지 있음
- completeness를 조금 확장
- 3가지 규칙
  - 각각의 참조 매개변수는 각각의 lifetime parameter를 가짐
  - 하나의 참조 매개변수만 있는 경우, 그 참조의 lifetime은 모든 반환값의 lifetime parameter
  - 여러개의 참조 매개변수가 있지만, 그 중 하나가 &self나 &mut self인 경우, 그 참조의 lifetime은 모든 반환값의 lifetime parameter

Static Lifetime

- 프로그램 내내 살아있는 lifetime
- 모든 문자열 리터럴은 static lifetime을 가짐

```
let s: &'static str = "I have a static lifetime.";
```

- 웬만하면 쓰지 말자

Review

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```