# 10. generic type, trait, lifetime

# 10. 제네릭 타입, 트레이트, 라이프타임

## 10.0. Introduction

- 중복되는 개념을 효율적으로 처리하기 위한 도구로 generic이 존재함.
  - concrete 타입이나 기타 속성에 대해 추상화된 대역임.
- 라이프타임은 generic의 일종임.

## 10.1. Generic Data type

- generic data type은 일반화 하는데 유용함.

### 제네릭 구조체 정의

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
        let mixed = Point { x: 1.0, y: 4};
}
```

- 모든 T 타입은 같아야 함.
  - 위처럼 mixed는 안됨.
- 두 가지를 자유롭게 타입을 지정하고 싶다면 아래처럼 구조체를 변경해야 함 .

```
struct Point<T, P> {
        x: T,
        y: P,
}
```

## 제네릭 메서드 정의

```
struct Point<T> {
        x: T,
        y: T,
}
impl<T> Point<T> {
        fn x(&self) -> &T {
                self.x
        }
}
```

- 제네릭으로 impl 뒤에 T를 명시해주고 Point<T>를 명시해 주어야 제대로 인식이 됨.
- 아래처럼도 쓸 수 있기 때문

```
impl Point<i32> {
        fn getdist(&self) -> &i32 {
                self.x*self.x + self.y*self.y
        }
}
```

## 제네릭 코드의 성능

- 제네릭 타입 매개변수를 사용하면 런타임 비용이 발생하는지에 대한 문제가 있음.

- 러스트에서는 제네릭 타입의 사용이 구체적 타입을 사용했을 때와 비교하면 비슷함.
    - 컴파일 타임에 제네릭을 사용하는 코드를 단형성화 하기 때문
        - 단형성화(monomorphization)은 실제 구체 타입으로 채워진 특정 코드로 바꾸는 과정임.

## 10.2 트레이트로 공통된 동작을 정의하기

- 트레이트는 흔히 말하는 interface같은 존재임. (동일하진 않음)
- 특정한 타입을 가지고 있으면서 다른 타입과 공유할 수 있는 기능을 정의함.

### trait 정의하기

- 메서드 시그니처를 그룹화해 특정 목적을 달성하는데 필요한 일련의 동작을 trait이라고 함.

```
pub trait Summary {
        fn summarize(&self) -> String;
}
```

## 특정 타입에 trait 구현하기

```
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline,
self.author, self.location)
    }
}

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)
    }
}
```

- Tweet과 NewsArticle 구조체에 대해 각각의 Summary를 정의함.
- String으로 반환되는 것은 같지만, 그 내용이 다름.
- 다른 라이브러리에서 만든 crate에 대해서는 trait를 구현할 수 없음.
    - 일관성 규칙에 의함.
        - 자세히는 고아 규칙(부모 타입이 존재하지 않음)

## 기본 구현

- 기본 동작을 구현한다면 만약 오버라이드 하지 않을 시 기본 동작을 수행하게 됨.

```
pub trait Summary {
        fn summarize(&self) -> String {
                String::from("(Read more...)")
        }
}
```

## 매개변수로서의 trait

- impl 키워드와 트레이트 이름을 명시해야 함.

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

## 트레이트 바운드 문법 <- 좀 어려움..

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

트레이트 바운드는 꺽쇠괄호 안의 제네릭 타입 매개변수 선언에 콜론뒤에 위치함.
즉 `<T: Summary>`의 Summary에 해당함.
예를 들어 trait 문법을 사용하는 아래의 코드를 trait bound 문법으로 바꾸어 보자

```
pub fn notify(item1: &impl Summary, item2: &impl Summary){}
// turn it trait bound
pub fn notify<T: Summary>(item1: &T, item2: &T){}
```

이렇게 구현하게 되면 차이점이 존재하지만 두 Summary의 타입이 같다면 동일한 구현으로 볼 수 있다.

## + 구문으로 trait 여러 개 지정하기

```
pub fn notify(item : &(impl Summary + Display)) {}
pub fn notify<T: Symmary + Display>(item: &T) {}
```

- 여러개의 트레이트의 바운드를 위처럼 지정할 수 있다.
- item을 {}로 포맷팅할수도 있음.

## where 조항으로 트레이트 바운드 정리하기

- 트레이트 바운드가 여러개인 상황에서 조금 더 간단하게 만들기위한 문법임.

```
fn some_fn<T: Display + Clone, U: Clone + Debug>(t: &T, u:
&U) -> i32 {}
//위의 함수를 아래처럼 표현할 수 있음 .
fn some_fn<T, U>(t: &T,u:&U) -> i32
where
        T: Display + Clone,
        U: Clone + Debug,
{}
```

아래의 함수처럼 더 깔끔한 형태로 적을 수 있음.

## 트레이트를 구현하는 타입을 반환하기

- trait가 구현되어 있는 구조체를 반환한다는건가? 잘 모르겠음 .

## 트레이트 바운드를 사용해 조건부로 메서드 구현하기

- 트레이트 바운드가 적용되는 타입에 대해서만 트레이트가 구현되도록 하는 것임.

# 10.3 라이프타임 참고자의 유효성 검증하기

- 라이프타임은 어떤 참조자가 필요한 기간 동안 유효함을 보장하는 것이다.
  - 러스트의 모든 참조자는 라이프타임이라는 참조자의 유효성을 보장하는 범위를 갖음.
  - 러스트에서는 제네릭 라이프타임 매개변수로 이 관계를 명시해야 함.

## 라이프타임 명시 문법

```
&i32 // 참조자
&'a i32 // 명시적인 라이프타임이 있는 참조자
&'a mut i32 // 명시적인 라이프타임이 있는 가변참조자
```

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

`x`와 `y` 중 라이프타임이 더 작은 쪽이 제네릭 라이프 타임`'a`의 라이프타임이 됨. 그래서 코드를 아래처럼짜면 라이프타임에서 문제가 생김.

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(),
string2.as_str());
    }
    println!("The longest string is {}", result);
}
```

아래의 코드도 문제가 생김.

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

- 제네릭 라이프타임 타입 `'a`와 전혀 관련이 없는 result가 반환되기 때문임!!

## 구조체 정의에서 라이프타임 명시하기

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

fn main() {
    let novel = String::from("Call me Ishmael. Some years
ago...");
    let first_sentence =
novel.split('.').next().expect("Could not find a '.'");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

- 라이프타임을 위처럼 명시해 사용이 가능함.

## 라이프타임 생략하기

러스트 초기에는 모든 함수에 라이프타임을 명시해주어야 했음.
비효율적이라는 것을 깨닫고 라이프타임 생략 규칙을 만들었음.

### 정의

- 입력 라이프타임
  - 함수나 메서드 라이프타임
- 출력 라이프타임
  - 반환 값의 라이프타임

### 규칙

1. 컴파일러가 참조자인 매개변수 각각에게 라이프타임 매개변수를 할당함.
   ex) 매개변수가 두개면 라이프타임 매개변수가 두 개임
2. 만약 입력 라이프타임 매개변수가 딱 하나라면 모든 출력 라이프타임에 대입됨.
   ex) `fn foo<'a>(x: &'a i32) -> &'a i32`

3. 입력 라이프타임 매개변수가 여러 개일 때 그 중 하나가 `&self`나 `&mut self`라면 `self`의 라이프타임이 모든 출력 라이프타임 매개변수에 대입됨.

```
fn longest(x: &str, y: &str) -> &str {
// rule 1
fn longest<`a, `b>(x: &`a str, y: &`b str) -> &str {
```

이 함수는 3가지 규칙을 모두 적용해도 결과 값의 라이프타임을 알 수 없으므로 컴파일 에러가 남.

## 메서드 정의에서 라이프타임 명시하기

- 라이프타임을선언하게 되면 라이프타임이 구조체 타입의 일부가 되기 때문에, 구조체 필드의 라이프타임 이름은 impl 키워드 뒤에 선언한 다음 구조체 이름 뒤에 다시 명시해 주어야 함.

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str)
-> &str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

- 위 코드는 앞선 `longest(x,y)` 와는 다르게 컴파일 에러가 나지 않음. `&self`에의해 반환 값의 라이프타임이 추론되기 때문임.

## 정적 라이프타임

- static lifetime은 해당 참조자가 프로그램의 전체 생애주기 동안 살아있음을 의미함.
- 모든 문자열 리터럴은 모두 `static` 라이프 타임을 가짐.
    - `let s: &'static str = "I have a static lifetime.";`

- 문자열의 텍스트는 프로그램의 바이너리 내에 직접 저장되어 언제나 사용이 가능함.