Functional Prgramming Language

다음 키워드를 배운다

- Side effect
- Pure function
- Functional programming
- Functional programming language

모든 함수는 두 종류의 입력과 출력을 가진다

1. 일반적인 입출력
2. 숨겨진 입출력

```
// code 1
public int square(int x) {
  return x * x;
}

// code 2
public void processNext() {
  Message msg = InboxQueue.popMessage();
  if (msg != null) {
    process(msg)
  }
}
```

다음 code 1, 2 에서 입출력을 찾아보자

"숨겨진 입출력" 이것이 바로 **Side Effect(부작용)**

> 다음과 같이 구분하는 경우도 있음
>
> 숨겨진 출력 - side effect
> 숨겨진 입력 - side cause

부작용은 복잡성의 빙산이다

다음 함수가 부작용(and 부원인)을 가진다면..

```
public boolean processMessage(Channel channel) { ... }
```

이 함수가 어떤 일을 할까?

함수의 내부를 보지 않고 알 수 없음
표면 아래 잠재적으로 큰 복잡성이 숨어 있다

함수를 제대로 파악하려면?

1. 함수 정의를 파고든다
2. 복잡성을 표면 위로 들어낸다

그래서 부작용이 나쁜건가?

예상한 그대로 정확하게 동작한다면 괜찮다

하지만... 이런 코드를 테스트 할 수 있나?

코드 내부를 확인하고 숨겨인 원인과 결과를 파악하고...
그럴듯하게 시뮬레이션해함

블랙 박스 테스트

부작용이 있는 함수라면?

함수가 어디든 의존할 수 있고 무엇이든 변경할 수 있으니..
버그는 어느 곳에든 있을 수 있다.

부작용을 표면 위로 들어내보자

다음 함수는 숨겨진 입력을 가진다. 찾아보자!

```
public Program getCurrentProgram (TVGuide tg, int ch) {

  Schedule schedule = tg.getSchedule(ch);
  Program cur = schedule.programAt(new Date())

  return cur
}
```

숨겨진 입력은 바로 `new Date()` !_!

우리는 이 입력을 (아주 간단하게) 표면위로 들어 낼 수 있다

```
public Program getCurrentProgram(..., Date when) {
  Schedule schedule = tg.getSchedule(ch);

  // Program cur = schedule.programAt(new Date())
  Program cur = schedule.programAt(when)

  return cur
}
```

이제 이 함수는 숨겨진 입출력이 없다 !

파라미터가 하나 증가하여 복잡하게 보일 수 있다.
But.. 의존성을 숨긴다고 더 간단해지지는 않음..
의존성을 정직하게 들어낸다고 더 복잡해지지 않음!!

또 다른 장점으로는,

추론하기 더 쉽다

입렵과 출력 사이 관계만 테스트하면 함수 전체를 테스트한 것이 됨!!

순수함수는 무엇인가?

모든 입력이 입력으로 **선언되고**
모든 출력이 출력으로 **선언된** 함수를 **순수** 하다고 한다.

즉 **숨겨진 입출력이 없어야 한다**

반대로 숨겨진 입출력이 있으면 순수하지 않다

순수하지 않는 코드를 테스트하거나
디버깅할 때 그것이 의존하고 있는 것을 항상 신경써야됨

함수형 프로그래밍이란?

순수 함수를 작성하는 것

최대한 부작용을 제거하여
코드의 대부분이 입력과 출력의 관계를 기술하게끔 하는 것.

함수형 프로그래밍 언어란?

함수형 프로그래밍은 ... 이 아니다.

1. `map` or `reduce` 가 아니다.
2. 람다 함수가 아니다
3. 타입에 관한 것이 아니다

1. `map` or `reduce` 가 아니다.

모든 함수형 언어에서 이 함수들을 보게 된다 할지라도
이것을 인하여 언어가 함수형이 되는 것은 아님

2. 람다 함수가 아니다

부작용을 회피하기 위한 언어를 만들다보면 자연스럽게 얻어지는 것.
FP를 가능하게 하는 것이지 핵심 요소는 아님

3. 타입에 관한 것이 아니다

정적 타입은 매우 유용하지만 FP의 전제 조건은 아님.
대표적으로 Lisp -> 함수형 언어이며 동적 언어

함수형 프로그래밍은 부작용에 관한 것.

언어 차원에서 의미하는 바는 무엇인가?

1. JavaScript

`this` 는 모든 함수의 숨겨진 입력
`this` 의 참조 대상을 추적하는거 어려움..

함수형 관점에서 보면 어디서나 접근가능하다는 것은
설계 결함의 징후(design smell) 이다.

2. Java

자바1.8에 람다가 추가되었다고 함수형 프로그래밍 언어 아님.

자바의 핵심 설계 원칙 - 코드는 지역화된 부작용들로 조직화되어야 한다

자바는 부작용을 국소화(localization)하는 것이 좋은 코드

OO, FP 모두 부작용을 문제라고 인식하지만 반응이 다름
OO - 부작용을 객체라는 경계에 가두자
FP - 부작용을 제거하자

3. Scala

스칼라는 "부작용 필수"와 "부작용 금지" 사이의 격차를
해소하려는 것이라고 볼 수 있음

4. Python

잠깐 자바에서의 근본적인 부작용 패턴을 보자

```
public String getName() {
  return this.name;
}
```

this가 숨겨진 입력. 이 함수를 순수하게 만들어보자.

```
publis String getName(Person this) {
  return this.name;
}
```

이제 `getName()` 은 순수 함수이다.

파이썬은 기본적으로 두 번째 패턴을 채택.

파이썬에서 모든 객체 메소드는 this(self) 를 첫 번째 인자로 가짐

```
def get_name(self):
  return self.name
```

확실히 명시적인 것이 묵시적인 것보다 낫다.

Design Smell

부작용을 발견하기 쉬운 2가지 타겟

1. 인자 없는 함수
2. 반환값 없는 함수

1. 인자가 없으면 부원인 신호

인자가 없으면 둘 중 하나.
같은 값을 반환하거나 다른 곳으로부터 입력을 취한다(부원인이 있다)

2. 반환값이 없으면 부작용 신호

반환값이 없으면 부작용이 있거나 호출해봤자 아무 의미 없는 함수이다

함수 signature 만 보면 전혀 호출할 이유가 없음

결론

모든 언어는 순수 함수를 작성할 수 있다
모든 입출력을 함수 signature 에 작성.

그렇다면 모든 프로그래밍 언어가 "함수형" 인가?? -> NO!

함수형 프로그래밍 언어는 부작용없는 프로그래밍을 지원하고 장려하는 언어

부작용에 적대적인 언어!

우리가 가능한 부작용을 제거하고 그렇지 않은 곳에는 철저하게 제어 할 수 있도록 적극적으로 도와주는 언어

Reference

- (번역) 함수형 프로그래밍이란 무엇인가?
- (번역) 어떤 프로그래밍 언어들이 함수형인가?

더 보면 좋을 것들

- 함수형 프로그래밍 소개
- (번역) 왜 함수형 프로그래밍이 중요한가—John Hughes 1989