프로그래밍 패러다임들

- 선언형
  - 함수형
  - 논리형
  - 제한형
- 명령형

구조적, 비구조적, 절차적, 객체지향, 값수준, 함수수준, 컴포넌트 지향, 클래스 기반, 메타, 프로토타입 기반, 규칙 기반, 주체지향, 정책 기반...

명령형 프로그래밍 vs 선언형 프로그래밍

명령형 프로그램은 *상태를 변형하는 일련의 명령들*로 구성된다

선언형 프로그램은 *목표를 명시하고, 구체적인 방법은 작성하지 않는다*

명령형 프로그래밍

전형적인 `for` 루프가 훌륭한 예

```
public String cleanNames(List<String> listOfNames) {
  StringBuilder result = new StringBuilder();
  for (int i = 0; i < listOfNames.size(); i++) {
    if (listOfNames.get(i).length() > 1) { // 한 글자짜리 이
      result.append(
          capitalizeString(listOfNames.get(i))). // 이름들을
      append(",");
    }
  }
  return result.substring(0, result.length() - 1).toStrin
}

public String capitalizeString(String s) { // 문자열의 가장 
  return s.substring(0, 1).toUpperCase() + s.substring(1,
}
```

각 단계에서 컴퓨터가 해야 할 *유용한 작업*을 *저수준의 메커니즘*으로 명시

선언형 프로그래밍

```
val result = listOfNames
  .filter(_.length() > 1) // 한 글자짜리 이름 필터
  .map(_.capitalize) // 이름들을 대문자로 변형
  .reduce(_ + "," + _) // 하나의 문자열로 변환
```

세부 구현에 신경쓸 필요 없이 *해법을 개념화*하여 컴퓨터에게 일러주기만 하면 됨

Syntax

낯선 문법들

`filter`, `map`, `reduce` ...

Paradigm

Scala를 Java처럼 쓰기

*사고 방식*이 바뀌어야 함

XSLT

Extensible Stylesheet Language Transformations

XML 문서를 다른 XML 문서로 변환하는데 사용하는 XML 기반 언어

기존 XML에서 새로운 XML을 만들기 위해 당연히 **상태**를 변경하려고 했는데?

순수 함수

부작용(side-effect)이 없는 함수, 다시 말해 주어진 입력으로 계산하는 것 외에 어떤 외부 상태에도 영향을 미치지 않는 함수

*참조 투명성*을 보장

- 외부의 변수를 변경
- 변수값을 갱신
- 파일이나 네트워크로 데이터 내보내기
- 예외

참조 투명성

입력 인자가 같다면 함수의 결과는 동일

표현식(expression)을 모두 *그 표현식의 결과로 치환*해도 프로그램에 아무 영향이 없다면 그 표현식은 참조에 투명

함수가 모든 입력값에 대해 *참조에 투명*하면 그 함수는 **순수**

혼란

Clean code, Refactoring, Design Pattern...

객체를 만들고, 객체의 상태와 행위를 적절히 규정하고, 객체 간의 상호 작용을 정의하고...

관점의 차이

> 객체지향 프로그래밍은 움직이는 부분을 *캡슐화*하여 코드 이해를 돕고,
> 함수형 프로그래밍은 움직이는 부분을 *최소화*하여 코드 이해를 돕는다.
> 마이클 페더스

객체지향 프로그래밍에서 캡슐화, 스코핑, 가시성 등의 메커니즘은 상태 변화의 *세밀한 제어*를 위해 존재한다.
함수형 언어는 상태를 제어하는 메커니즘을 구축하기보다, 그런 *움직이는 부분을 아예 제거*하는 데 주력한다.

사고의 전환

객체지향 프로그래밍의 재사용 단위는 클래스와 그 클래스들이 주고 받는 메시지. *고유한 자료구조*를 작성하는 것을 권장

함수형 프로그래밍은 몇몇 자료구조를 적절히 조작하는 *함수*의 사용을 권장

재사용의 단위가 클래스에서 함수로 내려오게 됨

Design Pattern

```
object CurryTest extends App {
  def filter(xs: List[Int], p: Int => Boolean): List[Int]
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
    else filter(xs.tail, p)

  def dividesBy(n: Int)(x: Int) = ((x % n) == 0)

  val nums = List(1, 2, 3, 4, 5, 6, 7, 8)

  println(filter(nums, dividesBy(2)))
  println(filter(nums, dividesBy(3)))
}
```

Multi Paradigm

최근 언어들은 하나의 패러다임만을 지원하지 않음

C++만 봐도 절차적, 객체기반, 객체지향, 제네릭

https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages

언어의 철학

Python

There should be one-- and preferably only one --obvious way to do it

Perl

There is more than one way to do it

http://hyperpolyglot.org/

당신 인생의 이야기

> 빛이 한 각도로 수면에 도달하고, 다른 각도로 수중을 나아가는 현상을 생각해보자. 굴절률의 차이 때문에 빛이 방향을 바꿨다고 설명한다면, 이것은 인류의 관점에서 세계를 보고 있다는 얘기가 된다. 빛이 목적지에 도달하는 시간을 최소화했다고 설명한다면, 당신은 헵타포드의 관점에서 세계를 보고 있는 것이다. 완전히 다른 두 가지의 해석이다.
>
> ...
>
> **한 가지 방식은 인과적이고, 다른 방식은 목적론적이다. 두 가지 모두 타당하고, 한쪽에서 아무리 많은 문맥을 동원하더라도 다른 한쪽이 부적격 판정을 받는 일은 없다.**