Chapter 1

목차

- 자바가 변화하는 이유
- 컴퓨팅의 변화
- 자바 발전에 대한 압박
- 자바 8과 자바 9의 핵심 기능

개요

`Java 8` 은 어느 버전보다도 많은 변화가 이루어졌다.

리스트 내의 아이템을 정렬하는 예제코드

```
Collections.sort(inventory, new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
});
```

위의 코드를 Java 8에서는 아래와 같이 바꿀 수 있다.

```
inventory.sort(comparing(Apple::getWeight));
```

이것은 `apple weight` 를 비교해서 `inventory` 를 정렬하라고 읽을 수 있다.

여기에는 하드웨어의 영향도 있다. `CPU가 멀티코어` 가 되었다. 그러나 기존 자바의 대부분은 1개의 코어를 사용했다. 그러므로 `Java 8`의 이전 버전에서 스레드를 쓰는 것은 굉장히 어렵고 에러가 나기 쉬운 환경이었다. `Java 5` 에서 스레드 풀과 동시성 컬렉션, `Java 7` 에서 포크/조인 프레임워크, 병렬화가 추가되었지만 여전히 병렬 처리는 어려웠다. `Java 8` 에서 병렬화를 처리하기 위한 방법이 소개되었지만, 몇가지 규칙을 따른다. `Java 9` 에서 `동적 프로그래밍` (동시성을 위한 구조적 메소드)을 추가했다. 이는 `RxJava` 와 `Akka` 를 이용하는 수단을 표준화한다.

RxJava란?

Akka란?

Java 8의 핵심 아이디어는 다음과 같다.

- Stream API
- 메소드에 코드를 전달하는 방법
- 인터페이스의 Default method

Stream API

`Stream API` 는 낮은 실행 수준을 보여준다. 결과적으로 코드 작성량을 줄여 에러 발생과 저비용으로 실행할 수 있게 해준다.

멀티코어 CPU는 각각의 프로세스에 할당되는 분리된 캐시를 가진다. `Lock` 을 위해서는 이 캐시들이 동기화 되어야 하며, 느린, 일관성 있는 코어 간 통신이 필요하다.

또한 Stream API는 `lambda`와 `default method` 추가의 직접적인 원인이 되었다고 볼 수 있다.

- 매개변수를 보내는데 있어 `추상 클래스`를 사용한 것과 같은 효과
- `함수형 프로그래밍` 을 제공하며 생산적인 코드를 제공

계속 변화하는 이유

프로그래밍 언어는 생태계의 양상을 띈다. 새로운 언어가 생기면, 오래된 언어는 사라진다.

예시

예를 들어, `C` 와 `C++` 은 운영체제와 임베디드 시스템을 구축하는데 주로 쓰이는 언어로 남았다. 프로그래밍 안정성에 비해 짧은 런타임이 그 이유이다. 이들의 안정성은 예측 불가능한 종료나 노출에 취약하다. 실제로, `Java` 와 `C#` 과 같은 객체지향 언어들이 추가적인 런타임이 있더라도 다양한 애플리케이션에 사용된다.

생태계 속의 자바

자바는 처음 배포되었을 때부터 디자인이 잘 되어 있고, 다양한 라이브러리가 포함된 객체지향언어였다. 작은 크기의 프로그램에도 적합하고, `JVM` 컴파일러가 모든 브라우저에서 지원되는 수준으로 웹에 적합한 언어가 되었다. 최근에는 JVM이 지원하는 자바의 경쟁 언어 (Scala, Groovy, Kotlin) 들이 생겼지만, 여전히 자바는 스마트카드, 토스터, 셋톱박스, 자동차 주행 시스템 등의 분야에서 쓰이고 있다.

자바는 어떻게 시장을 공략했는가?

객체지향은 1990년대부터 두가지 이유로 인기를 얻게 되었다.

1. `캡슐화` 를 통해 C에 비해 적은 오류
2. 윈도우 95 버전 이상에서 윔프 프로그래밍 모델의 기초

즉, 모든것은 객체이고, 동작이 핸들러로 전달되는 형식. 한번에 입력으로 다양한 출력을 구현할 수 있고 일찍이 브라우저들이 자바 코드가 동작하는 환경을 갖춰 대학에서 시장을 구축했고, 산업으로 이어졌다. C와 C++보다는 추가 실행 비용이 있었지만, 하드웨어가 빠르게 성장하면서 프로그래머의 역량도 중요해졌다. MS의 `C#` 은 객체 지향모델을 더욱 검증하게 되었다.

그러나 최근 프로그램 언어 생태계에는 변화가 있다. 프로그래머들은 `빅데이터` (TB 크기 이상의 데이터)와 `멀티코어` 환경에서 효율적으로 동작하기를 원한다. 즉, 자바에게 친숙하지 않은 `병렬` 시스템을 원하는 것이다. Java 8에서 추가되는 항목들은 존재하는 프로그래밍적 문제를 빠르고, 중요하고, 쉬운 방법으로 해결할 수 있게 돕는다는 것이다.

자바의 메소드

프로그래밍 언어에 있어서 `function` 이라는 단어는 `method` 의 의미로 사용된다.

Java 8에서는 람다와 익명 메소드를 포함하여 값으로서의 메소드를 제공한다.

```
public static List<Apple> filterHeavyApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (apple.getWeight() > 150) {
            result.add(apple);
        }
    }
    return result;
}
```

이제 한번밖에 쓰이지 않을 메소드를 작성하지 않아도 된다. 거쳐가는 메소드를 작성할 필요가 없기 때문이다.

```
filterApples(inventory, (Apple a) -> GREEN.equals(a.getColor()) );
filterApples(inventory, (Apple a) -> a.getWeight() > 150 );
filterApples(inventory, (Apple a) -> a.getWeight() < 80 ||
                                     RED.equals(a.getColor()) );
```

또한 `filter` 를 사용해서 같은 동작을 실행할 수 있다.

```
static <T> Collection<T> filter(Collection<T> c, Predicate<T> p);
filterApples(inventory, (Apple a) -> a.getWeight() > 150 );
filter(inventory, (Apple a) -> a.getWeight() > 150 ); //filter 사용
```

Streams

거의 모든 자바 애플리케이션들이 콜렉션을 만들거나 수행한다. 그러나 항상 콜렉션을 사용하는게 이상적이지는 않다.

```
Map<Currency, List<Transaction>> transactionsByCurrencies =
    new HashMap<>();
for (Transaction transaction : transactions) {
    if(transaction.getPrice() > 1000){
        Currency currency = transaction.getCurrency();
        List<Transaction> transactionsForCurrency =
            transactionsByCurrencies.get(currency);
        if (transactionsForCurrency == null) {
            transactionsForCurrency = new ArrayList<>();
            transactionsByCurrencies.put(currency,
                                         transactionsForCurrency);
        }
        transactionsForCurrency.add(transaction);
    }
}
```

이 코드는 싸여 있는 코드의 분기가 많아 이해하기 어렵다. `Stream API` 를 사용한다면 다음과 같이 해결할 수 있다.

```
import static java.util.stream.Collectors.groupingBy;
Map<Currency, List<Transaction>> transactionsByCurrencies =
    transactions.stream()
                .filter((Transaction t) -> t.getPrice() > 1000)
                .collect(groupingBy(Transaction::getCurrency));
```

Stream API는 Collection의 요소들을 비교하는데 사용되고 있다. collection을 사용하면 개발자가 직접 반복을 제어할 수 있다(`external iteration`). 반대로, Stream을 사용하면 반복에 대해 생각할 필요가 없다. 라이브러리 내부에서 수행되기 때문이다 (`internal iteration`)

Multicore Computers

자바는 한개의 코어만 사용하고, 전력이 낭비된다. 유사하게, 많은 회사들은 `computing clusters` 를 많은 데이터를 효율적으로 처리하기 위해 사용한다. Java 8은 이런 컴퓨터에도 최적화될 수 있는 방법을 제시한다.

이전 버전의 자바를 가지고 멀티 스레드 환경에서 쓰는 것은 어려웠다. 스레드가 동시에 업데이트나 접근이 가능했기 때문에 데이터들이 정합성이나 일관성을 유지하지 못했다. 동기화 라는 단어를 통해서, 계속해서 작은 오류가 생겼다. Stream 기반의 병렬은 함수형 프로그래밍의 사용을 증가시키고 동기화까지 이루어냈다.

자바의 병렬과 공유 불가능한 상태

Java 8 에서 병렬을 해결하는 것에는 두 가지가 있다.

1. 라이브러리가 큰 스트림을 핸들링해서 작은 단위로 나눠 처리한다.
2. 이 병렬 처리는 스트림에게 자유롭고, `filter` 와 같은 라이브러리 메소드를 거치며 상호작용하지 않아야만 작동한다.

이 조건들은 컴포넌트 간 실행 시 상호작용이 없는 것과 같은 효과를 준다.

요약

- 언어 생태계를 기억하고, 언어의 진화와 도태에 대해서 생각해야 한다. 자바는 아직 건재한 언어이지만, `COBOL` 과 같은 전성기가 있었지만 쇠퇴한 언어도 존재한다.
- 자바 8의 핵심 추가점은 함수적으로 프로그래밍하여 사용성과 효율성 모두를 갖춘 것이다.
- 멀티코어 프로세서들은 완전히 지원되지는 않는다.
- 함수들은 첫번째 클래스 값이다. 어떻게 메소드가 함수적 값이 될 수 있고, 익명 함수가 사용되는 방법을 알아야 한다.
- Java 8 의 스트림은 collection의 양상을 일반화했지만, 형태적으로는 읽기 쉽고 병렬 처리가 가능하다.
- 대규모 프로그래밍과 시스템 인터페이스는 자바에서 제공되지 않았다. 이제는 모듈을 특정화하여 구조적 시스템화를 할 수 있다.