Chapter 2. 동작 파라미터화 코드 전달하기

- **동작 파라미터화**란 아직은 어떻게 실행할 것인지 결정하지 않은 코드 블록을 의미한다.
- 이 코드 블록은 나중에 프로그램에서 호출한다. 즉, 코드 실행은 나중으로 미뤄진다.
- 예를 들어 메서드의 인수로 코드 블록을 전달할 수 있으며 결과적으로 메서드의 동작이 파라미터화된다.

2.1 변화하는 요구사항에 대응하기

- 농부의 재고 조사를 쉽게 할 수 있도록 돕는 애플리케이션

2.1.1 첫 번째 시도 : 녹색 사과 필터링

```
enum Color {RED, GREEN}
```

```
public static List<Apple> filterGreenApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple : inventory) {
        if (GREEN.equals(apple.getColor())) {
            result.add(apple);
        }
    }
    return result;
}
```

- 만약 빨간 사과도 필터링 하고 싶어졌다면?
    - => filterRedApples라는 메서드를 만들고 if문의 조건을 빨간 사과로 바꾼다.
- 만약 옅은 녹색, 어두운 빨간색, 노란색 등으로 필터링하고자 한다면 적절하게 대응x
    - => 코드를 추상화한다.

2.1.2 두 번째 시도 : 색을 파라미터화

- 색을 파라미터화할 수 있도록 메서드에 파라미터를 추가하면 변화하는 요구사항에 좀 더 유연하게 대응o

```
public static List<Apple> filterAppleByColor(List<Apple> inventory, Color
color) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple : inventory) {
        if (apple.getColor().equals(color)) {
            result.add(apple);
        }
    }
    return result;
}
```

```
List<Apple> greenApples = filterApplesByColor(inventory, GREEN);
List<Apple> redApples = filterApplesByColor(inventory, RED);
```

- 갑자기 농부가 색 이외에도 가벼운 사과와 무거운 사과로 구분할 수 있다면 좋겠어요.
- 무거운 사과는 150g을 기준으로 하고 싶어요.

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

=> 이는 소프트웨어 공학의 DRY(don't repeat yourself, 같은 것을 반복하지 마라) 원칙을 어기게 된다.

2.1.3 세 번째 시도 : 가능한 모든 속성으로 필터링

```
public static List<Apple> filterAppleByColor(List<Apple> inventory, Color color,
                                             int weight) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple : inventory) {
        if ((flag && apple.getColor().equals(color)) ||
            (!flag && apple.getWeight() > weight)) {
            result.add(apple);
        }
    }
    return result;
}
```

```
List<Apple> greenApples = filterApples(inventory, GREEN, 0, true);
List<Apple> heavyApples = filterApples(inventory, null, 150, false);
```

만약 사과의 크기, 모양 등으로 필터링하고 싶다면?

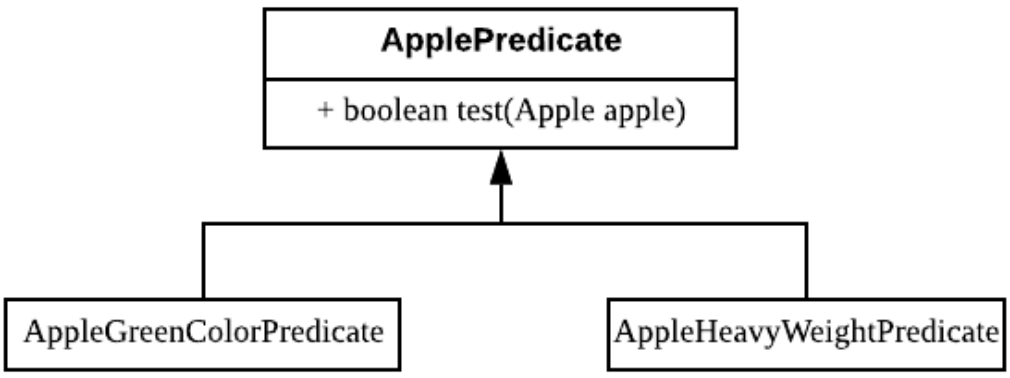
2.2 동작 파라미터화

- 지금까지는 동작을 파라미터화하기 보다 값을 파라미터로 추가해서 문제를 해결하였다.
- 따라서 파라미터를 추가하는 방법이 아닌 변화하는 요구사항에 좀 더 유연하게 대응할 수 있는 방법이 알고 싶을 것이다.

```
public interface ApplePredicate {
    boolean test (Apple apple);
}
```

```
public class AppleHeavyWeightPredicate implements ApplePredicate {
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
}
```

```
public class AppleGreenColorPredicate implements ApplePredicate {
    public boolean test(Apple apple) {
        return GREEN.equals(apple.getColor());
    }
}
```



위 구현체에 따라 filter 메서드가 다르게 동작한다. 이를 전략 디자인 패턴이라고 부른다.

전략 디자인 패턴은 각 알고리즘을 캡슐하하는 알고리즘 패밀리를 정의해둔 다음에 런타임에 알고리즘을 선택하는 기법이다. 이 예제에서는 ApplePredicate가 알고리즘 패밀리이고 AppleHeavyWeightPredicate와 AppleGreenColorPredicate가 전략이다.

filterApples에서 ApplePredicate 객체를 받아 애플의 조건을 검사하도록 메서드를 고쳐야 한다.

이렇게 **동작 파라미터화**, 즉 메서드가 다양한 동작을 **받아서** 내부적으로 다양한 동작을 **수행**할 수 있다.

2.2.1 네 번째 시도 : 추상적 조건으로 필터링

```
public static List<Apple> filterApples(List<Apple> inventory, ApplePredicate p)
{
    List<Apple> result = new ArrayList<>();
    for(Apple apple : inventory) {
        if(p.test(apple)) {
            result.add(apple);
        }
    }
    return result;
}
```

> **코드/동작 전달하기**
>
> 이전까지의 코드에 비해 유연한 코드를 얻었으며 동시에 가독성도 좋아졌을 뿐 아니라 사용하기도 쉬워졌다. 이제 필요한 대로 다양한 ApplePredicate 구현체를 만들어서 filterApples 메서드로 전달할 수 있다.

```
public class AppleRedAndHeavyPredicate implements ApplePredicate {
    public boolean test(Apple apple) {
        return RED.equals(apple.getColor()) && apple.getWeight() > 150;
    }
}
```

```
List<Apple> redAndHeavyApples = filterApples(inventory,
                                             new AppleRedAndHeavyPredicate());
```

ApplePredicate 객체를 전달하고 있으나 구현체의 test메서드를 실행하기 때문에 '코드를 전달' 할 수 있는 것이나 다름없다.

퀴즈 2-1 유연한 prettyPrintApple 메서드 구현하기

사과 리스트를 인수로 받아 다양한 방법으로 문자열을 생성(커스터마이즈된 다양한 toString메서드와 같이)할 수 있도록 파라미터화된 prettyPrintApple 메서드를 구현하시오. 예를 들어 preetyPrintApple 메서드가 각각의 사과 무게를 출력하도록 지시할 수 있다. 혹은 각각의 사과가 무거운지, 가벼운지 출력하도록 지시할 수 있다. preetyPrintApple 메서드는 지금까지 살펴본 필터링 예제와 비슷한 방법으로 구현할 수 있다. 독자 여러분이 좀 더 쉽게 문제를 해결할 수 있도록 대략적은 코드를 공개한다.

```
public static void prettyPrintApple(List<Apple> inventory, AppleFormatter
formatter) {
    for (Apple apple : inventory) {
        String output = formatter.accept(apple);
        System.out.println(output);
    }
}
```

정답

```
public interface AppleFormatter {
    Strint accept(Apple apple);
}
```

```
public class AppleFancyFormatter implements AppleFormatter {
    public String accept(Apple apple){
        String characteristic = apple.getWeight() > 150 ? "heavy" : "light";
        return "A " + characteristic + " " + apple.getColor() + " apple";
    }
}
```

```
public class AppleSimpleFormatter implements AppleFormatter {
    public String accept(Apple apple) {
        return "An apple of " + apple.getWeight() + "g";
    }
}
```

```
public class Printer {
    public static void prettyPrintApple(List<Apple> inventory, AppleFormatter
formatter) {
        for (Apple apple : inventory) {
            String output = formatter.accept(apple);
            System.out.println(output);
        }
    }
}
```

```
public class PrinterExample {
    public static void main(String[] args) {
        List<Apple> appleList = new ArrayList<>();
        appleList.add(new Apple(130, RED));
        appleList.add(new Apple(160, GREEN));

        Printer.prettyPrintApple(appleList, new AppleFancyFormatter());
        Printer.prettyPrintApple(appleList, new AppleSimpleFormatter());
    }
}
```

> 실행 결과

```
A light RED apple
A heavy GREEN apple
An apple of 130g
An apple of 160g
```

2.3 복잡한 과정 간소화

- 현재 filterApples 메서드로 새로운 동작을 전달하려면 ApplePredicate 인터페이스를 구현하는 여러 클래스를 정의한 다음에 인스턴스화해야 한다.

- 자바는 클래스의 선언과 인스턴스화를 동시에 수행할 수 있도록 **익명 클래스**라는 기법을 제공한다.

2.3.1 익명 클래스

- **익명 클래스**는 자바의 지역 클래스와 비슷한 개념이다.
- 익명 클래스는 말 그대로 이름이 없는 클래스다.
- 익명 클래스를 이용하면 클래스 선언과 인스턴스화를 동시에 할 수 있다. 즉 즉석에서 사용가능.

2.3.2 다섯 번째 시도 : 익명 클래스 사용

> ApplePredicater 구현체를 익명 클래스를 사용하여 필터링 예제에 적용한 코드

```
List<Apple> redApples = Filter.filterApples(appleList, new ApplePredicate() {
    @Override
    public boolean test(Apple apple) {
        return RED.equals(apple.getColor());
    }
});
```

```
List<Apple> greenApples = Filter.filterApples(appleList, new ApplePredicate() {
    @Override
    public boolean test(Apple apple) {
        return GREEN.equals(apple.getColor());
    }
}); // 반복되어 지저분한 코드
```

> 단점

- 첫째 : 여전히 많은 공간을 차지하며 코드가 지저분하다.
- 둘째 : 많은 프로그래머가 익명 클래스의 사용에 익숙하지 않다.

> Example

```
public class MeaningOfThis {
    public final int value = 4;

    public void doIt() {
        int value = 6;
        Runnable runnable = new Runnable() {
            public final int value = 5;

            @Override
            public void run() {
                int value = 10;
                System.out.println(this.value);
            }
        };
        runnable.run();
    }

    public static void main(String[] args) {
```

```
        MeaningOfThis meaningOfThis = new MeaningOfThis();
        meaningOfThis.doIt();
    }
}
```

> 실행 결과

```
5
```

2.3.3 여섯 번째 시도 : 람다 표현식 사용

> 람다식 적용

```
List<Apple> redApples = filterAppples(inventory, (Apple apple)
                                    -> RED.equals(apple.getColor()));
```

```
List<Apple> redApples = Filter.filterApples(appleList, new ApplePredicate() {
    @Override
    public boolean test(Apple apple) {
        return RED.equals(apple.getColor());
    }
});
```

2.3.4 일곱 번째 시도 : 리스트 형식으로 추상화

```
public interface Predicate<T> {
    boolean test(T t);
}
```

```
public static <T> List<T> filter(List<T> list, Predicate<T> p) {
    List<T> result = new ArrayList<>();
    for (T e : list) {
        if (p.test(e)) {
            result.add(e);
        }
    }
    return result;
}
```

=> 이제 바나나, 오렌지, 정수, 문자열 등의 리스트에 필터 메서드를 사용할 수 있다.

2.4 실전 예제

자바에서 제공하는 API 에서 동작파라미터화 적용해보기.

2.4.1 Comparator로 정렬하기

java.util.Comparator 객체를 이용해서 정렬의 방식을 동작 파라미터화 할 수 있다.

```
//java.util.Comparator
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

> 동작 파라미터화 (익명 클래스 사용)

```
TreeSet<Fruit> treeSet = new TreeSet<>(new Comparator<Fruit>() {
    @Override
    public int compare(Fruit o1, Fruit o2) {
        if (o1.price < o2.price) return 1;
        else if(o1.price == o2.price) return 0;
        else return -1;
    }
});
```

> 람다식 적용

```
TreeSet<Fruit> treeSet = new TreeSet<>((o1, o2) -> {
    if (o1.price < o2.price) return 1;
    else if(o1.price == o2.price) return 0;
    else return -1;
});
```

2.4.2 Runnable로 코드 블록 실행하기

Runnable을 이용해서 다양한 동작을 스레드로 실행할 수 있다.

> 동작 파라미터화

```
Thread t = new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello World!");
    }
});
```

> 람다식 적용

```
Thread t = new Thread(() -> System.out.println("Hello World!"));
```

2.5 마치며

- 동작 파라미터화는 메서드 내부적으로 다양한 동작을 수행할 수 있도록 코드를 메서드 인수로 전달한다.

- 동작 파라미터화를 이용하면 변화하는 요구사항에 더 잘 대응할 수 있는 코드를 구현할 수 있다.
- 익명 클래스로도 코드를 깔끔하게 만들 수 있지만 자바8에서는 람다식을 통해 코드를 간결화 한다.
- 자바 API의 많은 메서드 또한 동작 파라미터화를 통해 다양한 동작으로 작동시킬 수 있다.