


2. 동작 파라미터화 코드 전달하기 - 너의 행동 가져오렴

동적 파라미터화 코드 전달하기

 클라이언트의 다양한 요구사항에 따라 조건이 바뀌는 상황에서 매번 새로운 메소드를 작성하기엔 유지보수가 지옥이다. 해결책으로 "동적 파라미터화"를 알아보자.

동적 파라미터화란?

동적 파라미터화란 아직은 어떻게 실행할 것인지 결정하지 않은 코드 블록을 의미한다. 이 코드 블록은 나중에 프로그램에서 호출한다. 즉, 코드 블록의 실행은 나중에 미뤄진다.

변화하는 요구사항에 대응하기

"변화에 대응하는 코드를 구현하는 것은 어려운 일이다."

실제 개발 현장에서 자주 마주하는 상황을 사과 필터링 예제로 살펴보자.

1 첫 번째 시도: 녹색 사과 필터링


```
import java.util.ArrayList;
import java.util.List;

enum Color {RED, GREEN}

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

문제점

- 갑자기 녹색 사과 말고 빨간 사과도 필터링하고 싶어졌다면?
- 메소드 복사해서 filterRedApples 라는 새로운 메소드를 만들고 조건문을 바꾸는 방법이 있다.
- 하지만 나중에 좀 더 다양한 색으로 필터링하는 등의 변화에는 적절하게 대응할 수 없다.

 설계 원칙: "거의 비슷한 코드가 반복 존재한다면 그 코드를 추상화한다."

2 두 번째 시도: 색을 파라미터화

```

public static List<Apple> filterApplesByColor(List<Apple> inventory, Color color)
{
    List<Apple> result = new ArrayList<>(); // 사과 누적 리스트
    for(Apple apple : inventory){
        if(apple.getColor().equals(color)){ // 색상 조건 확인
            result.add(apple);
        }
    }
    return result;
}

// 사용 예시
List<Apple> greenApples = filterApplesByColor(inventory, GREEN);
List<Apple> redApples = filterApplesByColor(inventory, RED);

```

✅ 개선점

- 색상에 대한 요구사항 변화에 유연하게 대응 가능하다.

❌ 새로운 문제

- 만약 무게에 대한 요구사항이 추가된다면?
- 요구사항을 계속 듣다 보면 색과 마찬가지로 앞으로 무게의 기준도 얼마든지 바뀔 수 있을 것이다.

```

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

```

❌ 새로운 문제점

- 각 사과에 필터링 조건을 적용하는 부분의 코드가 색 필터링 코드와 대부분 중복된다.
- 이는 소프트웨어 공학의 **DRY(Don't Repeat Yourself)** 원칙을 어기는 것이다.
- 탐색 과정을 고쳐서 성능을 개선하려면 무슨 일이 일어날까?
- 한 줄이 아니라 메소드 전체 구현을 고쳐야 한다. 엔지니어링적으로 비싼 대가를 치러야 한다.

3 세 번째 시도: 가능한 모든 속성으로 필터링 (안티패턴)

만류에도 불구하고 모든 속성을 메소드 파라미터로 추가한 모습이다.

```

public static List<Apple> filterApples(List<Apple> inventory, Color color, int
weight, boolean flag) {
    List<Apple> result = new ArrayList<>();

```

```

    for(Apple apple : inventory){
        if((flag && apple.getColor().equals(color)) || (!flag &&
apple.getWeight() > weight)){
            result.add(apple);
        }
    }
    return result;
}

```

❌ 심각한 문제점

- 형편없는 코드다!
- flag 파라미터의 의미가 불명확하다.
- 결국 여러 중복된 필터 메소드를 만들거나 모든 것을 처리하는 거대한 하나의 필터 메소드를 구현해야 한다.
- 하지만 어떤 기준으로 사과를 필터링할 것인지 효과적으로 전달할 수 있다면 더 좋을 것이다.

이제 **동적 파라미터화**를 이용해서 유연성을 얻는 방법을 설명한다.

동적 파라미터화

요구사항에 좀 더 유연하게 대응할 수 있는 방법이 절실하다는 것을 확인했다. 만약 사과의 어떤 속성에 기초해서 boolean 값을 반환하는 방법이 있다면?

참 또는 거짓을 반환하는 함수를 **프레디케이트(Predicate)**라고 한다.

선택 조건을 결정하는 인터페이스 정의

```

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

```

다양한 선택 조건을 대표하는 여러 버전의 ApplePredicate 를 정의할 수 있다.

```

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 조건에 따라 filter 메소드가 다르게 동작할 것이라고 예상할 수 있다.

전략 디자인 패턴

이를 **전략 디자인 패턴**이라고 한다. 전략 디자인 패턴은 각 알고리즘을 캡슐화하는 알고리즘 패밀리를 정의해둔 다음에 런타임에 알고리즘을 선택하는 기법이다.

- ApplePredicate 가 **알고리즘 패밀리**
- AppleHeavyWeightPredicate , AppleGreenColorPredicate 가 **전략**

4 네 번째 시도: 추상적 조건 필터링

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

코드/동작 전달하기

첫 번째 코드에 비해 더 **유연한 코드**를 얻었으며 동시에 **가독성도 좋아졌을** 뿐 아니라 **사용하기도 쉬워졌다**.

이제 빨간 사과를 검색해달라고 부탁하면 우리는 ApplePredicate 를 적절하게 구현하는 클래스만 만들면 된다.

```
public class AppleRedAndHeavyPredicate implements ApplePredicate {  
    public boolean test(Apple apple){  
        return RED.equals(apple.getColor()) && apple.getWeight() > 150;  
    }  
}  
  
// 사용 예시  
List<Apple> redAndHeavyApples = filterApples(inventory, new  
    AppleRedAndHeavyPredicate());
```

핵심 개념

전달한 ApplePredicate 객체에 의해 filterApples 메소드의 동작이 결정된다!

- 예제에서 가장 중요한 구현은 **test 메소드**다.
- filterApples 메소드의 새로운 동작을 정의하는 것이 test 메소드다.
- 메소드는 객체만 인수로 받으므로 test 메소드를 ApplePredicate 객체로 감싸서 전달해야 한다.
- test 메소드를 구현하는 객체를 이용해서 불리언 표현식 등을 전달할 수 있으므로 이는 '**코드를 전달**' 할 수 있는 것이나 다름없다.

한 개의 파라미터, 다양한 동작

컬렉션 탐색 로직과 각 항목에 적용할 동작을 분리할 수 있다는 것이 **동작 파라미터화**의 강점이다.

장점

1. **재사용성**: 한 메소드가 다른 동작을 수행하도록 재할용할 수 있다.
2. **유연성**: 유연한 API를 만들 때 동작 파라미터화가 중요한 역할을 한다.
3. **확장성**: 새로운 조건이 추가되어도 기존 코드를 수정하지 않고 새로운 전략만 추가하면 된다.

실제 적용 예시

```
// 무거운 사과 필터링
List<Apple> heavyApples = filterApples(inventory, new
AppleHeavyWeightPredicate());

// 녹색 사과 필터링
List<Apple> greenApples = filterApples(inventory, new
AppleGreenColorPredicate());

// 빨간색이면서 무거운 사과 필터링
List<Apple> redAndHeavyApples = filterApples(inventory, new
AppleRedAndHeavyPredicate());
```

정리

발전 과정 요약


```
// 1단계: 하드코딩된 조건
filterGreenApples(inventory)

// 2단계: 파라미터화된 조건
filterApplesByColor(inventory, GREEN)

// 3단계: 복잡한 파라미터 (안티패턴)
filterApples(inventory, GREEN, 150, true)

// 4단계: 동작 파라미터화 (최적해)
filterApples(inventory, new AppleGreenColorPredicate())
```

동적 파라미터화 - 복잡한 과정 간소화

 사용하기 복잡한 기능이나 개념을 사용하고 싶은 사람은 아무도 없다. 동적 파라미터화를 더욱 간소화하는 방법을 알아보자.

🔧 복잡한 과정 간소화

현재 `filterApples` 메소드로 새로운 동작을 전달하려면 `ApplePredicate` 인터페이스를 구현하는 여러 클래스를 정의한 다음에 인스턴스화해야 한다. 상당히 번거로운 작업이다.

📄 현재 코드의 번거로움

```
import com.study.ch02.ApplePredicate;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

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());    }}

public class FilterApples {
    public static void main(String[] args) {        List<Apple> inventory =
Arrays.asList(                new Apple(80, GREEN),                new Apple(155,
GREEN),                new Apple(120, RED)        );
        List<Apple> heavyApples = filterApples(inventory, new
AppleHeavyWeightPredicate());        List<Apple> greenApples =
filterApples(inventory, new AppleGreenColorPredicate());    }
    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;
    }
}
```

❌ 문제점

- 로직과 관련 없는 코드가 많이 추가되었다.
- 간단한 조건 하나를 위해 전체 클래스를 만들어야 한다.

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

😺 익명 클래스 (Anonymous Class)

익명 클래스는 자바의 지역 클래스와 비슷한 개념이다. 익명 클래스는 말 그대로 이름이 없는 클래스다. 익명 클래스를 이용하면 클래스 선언과 인스턴스화를 동시에 할 수 있다.

🎯 사용 목적

- 한 번만 사용할 클래스를 만들기 위해
- 간단한 인터페이스 구현/추상 클래스 상속
- 코드를 짧고 응집력 있게 작성

📖 익명 클래스 상세 설명

이름 없는 일회성(1회용) 내부 클래스다. 객체를 생성하면서 곧바로 클래스를 정의·구현까지 한 번에 해버리는 문법이다.

```
인터페이스나_상위클래스 ref = new 인터페이스나_상위클래스() {
    // 메서드 구현(override)
    @Override    public void run() {        System.out.println("Hello");    }
};
```

🔍 특징

- new 키워드 뒤에 클래스 이름 없이 바로 { ... } 블록으로 정의
- 즉시 한 번만 인스턴스화 → 별도 .class 파일이 생성되지 않는다.

📁 사용 상황별 가이드

상황	이유
전략·콜백을 간단히 주입	쓰고 버릴 구현체를 빨리 만드는 용도
GUI 이벤트 처리	버튼 클릭 리스너 등 코드량 적고 국지적 로직
람다 도입 전(Java 7 이하)	Comparator, Runnable 등 함수형 인터페이스 구현

⚠️ 특징 & 주의점

- 외부 지역 변수는 final 또는 effectively final (값 변경 안 됨)이어야 참조 가능하다.
- 생성자 없음 → 인스턴스 초기화는 이니셜라이저 블록 또는 메서드 내부에서 처리한다.
- 클래스명 없으니 재사용 불가, 긴 코드면 가독성이 나빠진다.
- Java 8 이후 대부분 람다로 대체 가능 → 익명 클래스는 인터페이스에 여러 메서드가 있을 때만 필요하다.

🔄 단계별 코드 개선 과정

5 다섯 번째 시도: 익명 클래스 사용

익명 클래스를 이용해서 ApplePredicate 를 구현하는 객체를 만드는 방법으로 필터링한 코드다.

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate(){
    public boolean test(Apple apple){        return RED.equals(apple.getColor());
    });
// filterApples 메소드의 동작을 직접 파라미터화했다.
```

❌ 여전한 문제점

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate(){
    public boolean test(Apple apple){
```

이 부분이 반복되어 지저분한 코드가 된다.

- 많은 프로그래머가 익명 클래스의 사용에 익숙하지 않다.
- 코드의 장황함은 나쁜 특성이다.
- 장황한 코드는 구현하고 유지보수하는 데 시간이 오래 걸릴 뿐 아니라 읽는 즐거움을 빼앗는 요소다.

6 여섯 번째 시도: 람다 표현식 사용

Java 8의 람다 표현식을 이용해서 위 예제 코드를 간단하게 재구현할 수 있다.

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

✅ 개선점

- 코드가 훨씬 간결해졌다.
- 가독성이 크게 향상되었다.
- 보일러플레이트 코드가 제거되었다.

7 일곱 번째 시도: 리스트 형식으로 추상화

```
import java.util.ArrayList;
import java.util.List;

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;
}
```

🎉 최종 결과

이제 바나나, 오렌지, 정수, 문자열 등의 리스트에 필터 메소드를 적용할 수 있다!

```
// 사과 필터링
List<Apple> redApples = filter(inventory, (Apple apple) ->
    RED.equals(apple.getColor()));

// 숫자 필터링
List<Integer> evenNumbers = filter(numbers, (Integer i) -> i % 2 == 0);
```



```
// 문자열 필터링
List<String> shortWords = filter(words, (String s) -> s.length() < 5);
```

실전 예제

동작 파라미터화가 변화하는 요구사항에 쉽게 적응하는 유용한 패턴임을 확인했다. 동작 파라미터화 패턴은 동작을 (한 조각의 코드로) 캡슐화한 다음에 메소드로 전달해서 메소드의 동작을 파라미터화한다.

실제 개발에서의 활용 예시

```
// Comparator를 이용한 정렬
inventory.sort((Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));

// 스레드 실행
Thread t = new Thread(() -> System.out.println("Hello World"));

// GUI 이벤트 처리 (JavaFX 예시)
button.setOnAction(event -> System.out.println("Button clicked!"));
```

마무리

핵심 요약

동작 파라미터화에서는 메소드 내부적으로 다양한 동작을 수행할 수 있도록 코드를 메소드 인수로 전달한다. 이를 이용하면 다음과 같은 이점을 얻을 수 있다:

얻은 것들

1. **유연성**: 변화하는 요구사항에 더 잘 대응할 수 있는 코드를 구현할 수 있다.
2. **비용 절약**: 나중에 엔지니어링 비용을 줄일 수 있다.
3. **코드 재사용**: 동일한 메소드로 다양한 동작을 수행할 수 있다.
4. **가독성**: 람다 표현식을 통해 코드가 더욱 간결하고 명확해진다.

발전 과정

```
// 기존: 복잡한 클래스 구현
public class AppleRedColorPredicate implements ApplePredicate {
    public boolean test(Apple apple) {        return
RED.equals(apple.getColor());    }}

// 익명 클래스: 조금 더 간단
filterApples(inventory, new ApplePredicate() {
    public boolean test(Apple apple) {        return
```

```
RED.equals(apple.getColor());    });  
  
// 람다 표현식: 매우 간결  
filterApples(inventory, apple -> RED.equals(apple.getColor()));
```

코드 전달 기법을 이용하면 동작을 메소드의 인수로 전달할 수 있다. Java 8 이전에는 코드를 지저분하게 구현해야 했으며, 익명 클래스로도 어느 정도 코드를 깔끔하게 만들 수 있지만 인터페이스를 상속받아 여러 클래스를 구현해야 하는 수고를 없앨 수 있는 방법을 **람다 표현식**이 제공한다.

💡 **다음 단계:** 람다 표현식과 메소드 참조를 더 깊이 학습하여 함수형 프로그래밍의 진정한 힘을 경험해보자!

태그

#Java #동적파라미터화 #익명클래스 #람다표현식 #제네릭 #함수형프로그래밍 #클린코드