모던 자바 인 액션 스터디 - 2주차

chanE

목차

모던 자바 인 액션 2장

변화하는 요구사항 - 1

모던 자바 인 액션 2장

개발자의 입장

```
public static List<Apple> filterGreenApples(List<Apple> inventory) {  no usages
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory) {
        if (GREEN.equals(apple.getColor())) {
            result.add(apple);
        }
    }
    return result;
}
```

Java 8에 추가된 내용 이외에 개선할 점이 있나요?

모던 자바 인 액션 2장

개발자의 입장

```
public static List<Apple> filterGreenApples(List<Apple> inventory) {  no usages
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory) {
        if (GREEN == apple.getColor()) {
            result.add(apple);
        }
    }
    return result;
}
```

enum은 == 비교가 가능합니다. == 를 사용하면 NullPointerException이 발생하지 않고, 컴파일 타임에 타입 미스매치를 잡아줍니다.

https://chan9.tistory.com/174

요구사항을 받은 지 하루가 지난 후, 농부는 이런 말을 합니다.

모던 자바 인 액션 2장

변화하는 요구사항 - 2

150그램 이상인 사과를
모두 찾고 싶어요!

모던 자바 인 액션 2장

개발자의 입장

```
public static List<Apple> filterHeavyApples(List<Apple> inventory) {  no usages
    List<Apple> result = new ArrayList<>();
    for (Apple apple : inventory) {
        if (apple.getWeight() > 150) {
            result.add(apple);
        }
    }
    return result;
}
```

크게 고민하지 않고, 원래 있던 코드를 복사 붙여넣기 해서 안의 if문의 조건만 변경했습니다.
그런데, **하루 뒤에** 농부가 찾아와서 다음과 같은 말을 합니다.

모던 자바 인 액션 2장

변화하는 요구사항 - 3

모던 자바 인 액션 2장

개발자의 입장

나중에 농부가 좀 더 다양한 색(옅은 녹색, 어두운 빨간색, 노란색 등)으로 필터링하는 변화를 요구하면? -> **적절하게 대응할 수 없습니다.**

거의 비슷한 코드가 반복 존재한다면 그 코드를 추상화합니다.

모던 자바 인 액션 2장

리팩터링 전에.. 알아두면 좋아요!

```
public class FilteringApples {

    Run | Debug
    public static void main(String[] args) {
        List<Apple> inventory = Arrays.asList(
                new Apple(weight:80, GREEN),
                new Apple(weight:155, GREEN),
                new Apple(weight:120, RED));
```

Arrays.asList로 생성한 리스트는 java.util.ArrayList 가 아닙니다.
java.util.Arrays의 static inner class인 ArrayList입니다.

getClass().getName() 을 출력해보면
다음과 같이 나옵니다. `java.util.Arrays$ArrayList`

요소를 갱신할 수는 있지만, 새 요소를 추가하거나(add) 요소를 삭제(remove) 할 수는 없습니다. 요소를 추가하거나 삭제하려 하면 UnsupportedOperationException이 발생합니다.

내부적으로 **고정된 크기**의 **변환할 수 있는 배열**로 구현되었기 때문에 이와 같은 일이 일어납니다.

```java
class ArrayTest {  Complexity is 3 Everything is cool!

    List<String> friends;  4 usages

    @BeforeEach
    void setUp() {
        friends = Arrays.asList("Raphael", "Olivia", "Thibaut");
    }

    @Test
    void Arrays_asList로_만들어진_리스트는_요소의_추가가_불가능하다() {
        assertThatThrownBy(() -> friends.add("Chih-Chun"))
                .isInstanceOf(UnsupportedOperationException.class);
    }

    @Test
    void Arrays_asList로_만들어진_리스트는_요소의_삭제가_불가능하다() {
        assertThatThrownBy(() -> friends.remove( index: 0))
                .isInstanceOf(UnsupportedOperationException.class);
    }

    @Test
    void Arrays_asList로_만들어진_리스트는_요소의_갱신은_가능하다() {
        assertThatNoException()
                .isThrownBy(() -> friends.set(0, "Richard"));
    }
}
```

모던 자바 인 액션 2장

동작 파라미터화

동작 파라미터화(behavior parameterization)를 이용하면 자주 바뀌는 요구사항에 효과적으로 대응할 수 있습니다.

동작 파라미터화란 아직은 어떻게 실행할 것인지 결정하지 않은 코드 블록을 의미합니다.
이 코드 블록은 나중에 프로그램에서 호출합니다. 즉, 코드 블록의 실행은 나중으로 미뤄집니다.

결과적으로 코드 블록에 따라 메서드의 동작이 파라미터화됩니다.

```
/**
 * 룸메이트에게
 * 1. 우체국에 가서
 * 2. 이 고객 번호를 사용하고
 * 3. 관리자에게 이야기한 다음에
 * 4. 소포를 가져오면 된다.
 */
public void go(/** 1번부터 4번까지의 동작을 go 메서드의 인수로 전달할 수 있다.**/) {  no usages
    // TODO
}
```

모던 자바 인 액션 2장

변화하는 요구사항에 대처하기 - 실습

첫 번째 시도: 녹색 사과만 필터링하는 기능을 추가

```
// 1. 첫 번째 시도: 녹색 사과 필터링 With 태훈
public static List<Apple> filterGreenApples(List<Apple> inventory) {  no usages
    List<Apple> apples = new ArrayList<>();
    for (Apple apple : inventory) {
        if (GREEN == apple.getColor()) {
            apples.add(apple);
        }
    }
    return apples;
}
```

모던 자바 인 액션 2장

변화하는 요구사항에 대처하기 - 실습

어떻게 해야 filterGreenApples의 코드를 반복 사용하지 않고 filterRedApples를 구현할 수 있을까요?
색을 파라미터화할 수 있도록 메서드에 파라미터를 추가하면 변화하는 요구사항에 좀 더 유연하게 대응하는 코드를 만들 수 있습니다.

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

두 번째 시도: 색을 파라미터화

```
// 2. 두 번째 시도: 색을 파라미터화 With 유승님
public static List<Apple> filterApplesByColor(List<Apple> inventory, Color color) {
    List<Apple> list = new ArrayList<>();
    for (Apple apple : inventory) {
        if (color == apple.getColor()) {
            list.add(apple);
        }
    }
    return list;
}
```

그런데 농부가 색 이외에도 **가벼운 사과와 무거운 사과**로 구분할 수 있으면 좋겠다고 합니다. 보통 무게가 150그램 이상인 사과가 무거운 사과입니다.

모던 자바 인 액션 2장

변화하는 요구사항에 대처하기 - 실습

두 번째 시도: 무게 정보 파라미터도 추가

```java
// 2. 두 번째 시도: 무게 정보 파라미터도 추가 With 도현님
public static List<Apple> filterApplesByWeight(List<Apple> inventory, int weight) {  Complexity is 5 Ever
    List<Apple> list = new ArrayList<>();
    for (Apple apple : inventory) {
        if (apple.getWeight() > weight) {
            list.add(apple);
        }
    }
    return list;
}
```

위 코드도 좋은 해결책이라고 할 수 있지만, 소프트웨어 공학의 DRY(don’t repeat yourself) 원칙을 어기고 있습니다.
그 다음 방법으로 색과 무게 중 어떤 것을 기준으로 필터링할지 가리키는 플래그를 추가하는 방법이 있습니다.

모던 자바 인 액션 2장

변화하는 요구사항에 대처하기 - 실습

세 번째 시도: 가능한 모든 속성으로 필터링

```
public static List<Apple> filterApples(List<Apple> inventory, Color color, int weight, boolean flag) {  no u
    List<Apple> result = new ArrayList<>();
    for (Apple apple : inventory) {
        if ((flag && color == apple.getColor()) || (!flag && apple.getWeight() > weight)) {
            result.add(apple);
        }
    }
    return result;
}
```

다음처럼 위 메서드를 사용할 수 있지만, 가독성이 떨어집니다.

또한 요구사항이 바뀌었을 때 유연하게 대응할 수도 없습니다. 예를 들어 사과의 **크기, 모양** 등으로 사과를 필터링하고 싶다면? 결국 여러 중복된 필터 메서드를 만들거나 아니면 모든 것을 처리하는 거대한 하나의 필터 메서드를 구현해야 합니다.

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

모던 자바 인 액션 2장

동작 파라미터화

filterApples에 어떤 기준으로 사과를 필터링할지 효과적으로 전달할 수 있다면 더 좋을것입니다.

사과의 어떤 속성에 기초해서 불리언값을 반환(예를 들어 사과가 녹색인가? 150그램 이상인가?) 하는 방법이 있습니다.
참 거짓을 반환하는 함수를 **프레디케이트**라고 합니다.

선택 조건을 결정하는 인터페이스를 다음과 같이 정의할 수 있습니다.

```
@FunctionalInterface  no usages
public interface ApplePredicate {

    boolean test(Apple apple);  no usages
}
```

모던 자바 인 액션 2장

@FunctionalInterface

인터페이스 유형 선언이 JLS에 정의된 대로 **funtional interface**가 되도록 의도되었음을 나타내는 데 사용되는 유익한 annotation type입니다.
개념적으로, funtional interface에는 **정확히 하나의 추상 메서드**가 있습니다. default method에는 구현이 있으므로 추상 메서드가 아닙니다.
인터페이스가 java.lang.Object의 공용 메서드 중 하나를 재정의하는 추상 메서드를 선언하는 경우 인터페이스의 모든 구현이 java.lang.Object 또는 다른 곳에서 구현되기 때문에 인터페이스의 추상 메서드 수에 포함되지 않습니다.

그런데.. 책에는 이 애노테이션이 없던데요 ❓

이 메서드를 사용하는 이유는 크게 세 가지 이유가 있습니다.

첫 번째, 해당 클래스의 코드나 설명 문서를 읽을 이에게 그 인터페이스가 **람다용**으로 설계된 것임을 알려준다.
두 번째, 해당 인터페이스가 추상 메서드를 오직 하나만 가지고 있어야 컴파일되게 해준다.
세 번째, 그 결과 유지보수 과정에서 누군가 실수로 메서드를 추가하지 못하게 막아준다. - 이펙티브 자바 아이템 44

모던 자바 인 액션 2장

동작 파라미터화

다음처럼 다양한 선택 조건을 대표하는 여러 버전의 ApplePredicate를 정의할 수 있습니다.

```
public class AppleHeavyWeightPredicate implements ApplePredicate {  no usages

    @Override  no usages
    public boolean test(final Apple apple) {  Complexity is 3 Everything is cool!
        return apple.getWeight() > 150;
    }
}
```

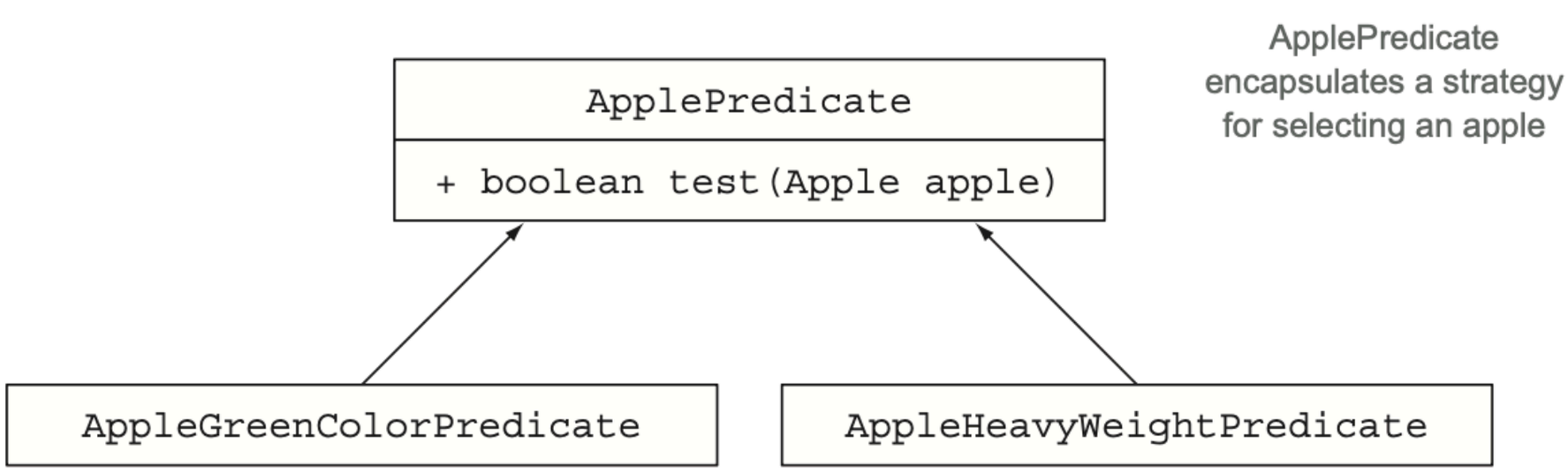
```
public class AppleGreenColorPredicate implements ApplePredicate {  no usages

    @Override  no usages
    public boolean test(final Apple apple) {  Complexity is 3 Everything is cool!
        return Color.GREEN == apple.getColor();
    }
}
```

모던 자바 인 액션 2장

동작 파라미터화 - 전략 디자인 패턴

위 조건에 따라 filter 메서드가 다르게 동작할 것이라고 예상할 수 있습니다. 이를 **전략 디자인 패턴**이라고 합니다.
전략 디자인 패턴은 각 알고리즘(전략이라 불리는)을 캡슐화하는 알고리즘 패밀리를 정의해둔 다음에 런타임에 알고리즘을 선택하는 기법입니다.



이제, filterAppples에서 ApplePredicate 객체를 받아 사과의 조건을 검사하도록 메서드를 고쳐야 합니다. 이렇게 동작 파라미터화, 즉 메서드가 다양한 동작(또는 전략)을 받아서 내부적으로 다양한 동작을 수행할 수 있습니다.

모던 자바 인 액션 2장

변화하는 요구사항에 대처하기 - 실습

네 번째 시도: 추상적 조건으로 필터링

```
public static List<Apple> filterApples(List<Apple> inventory, ApplePredicate p) {  no usages
    List<Apple> result = new ArrayList<>();
    for (Apple apple : inventory) {
        if (p.test(apple)) {
            result.add(apple);
        }
    }
    return result;
}
```

이제 필요한 대로 다양한 ApplePredicate 구현체를 만들어서 filterApples 메서드로 전달할 수 있습니다.
우리가 전달한 ApplePredicate 객체에 의해 filterApples 메서드의 동작이 결정됩니다.

즉, 우리는 filterApples의 **동작을 파라미터화**했습니다. ❗

모던 자바 인 액션 2장

변화하는 요구사항에 대처하기 - 실습

ApplePredicate object

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

Pass as argument

```
filterApples(inventory,          );
```

Pass a strategy to the filter method: filter the apples by using the boolean expression encapsulated within the ApplePredicate object. To encapsulate this piece of code, it is wrapped with a lot of boilerplate code (in bold).

여기까지는 좋은데..

여러 클래스를 구현해서 인스턴스화하는 과정이 조금은 거추장스럽게 느껴질 수 있습니다.

모던 자바 인 액션 2장

복잡한 과정 간소화 - 실습

다섯 번째 시도: 익명 클래스 사용

익명 클래스를 이용하면 **클래스 선언**과 **인스턴스화**를 동시에 할 수 있다. 즉, 즉석에서 필요한 구현을 만들어서 사용할 수 있습니다.

```
// 5. 다섯 번째 시도: 익명 클래스 사용
filterApples(inventory, new ApplePredicate() {  Complexity is 4 Everything is cool!
    @Override
    public boolean test(final Apple apple) {  Complexity is 3 Everything is cool!
        return RED == apple.getColor();
    }
});
```

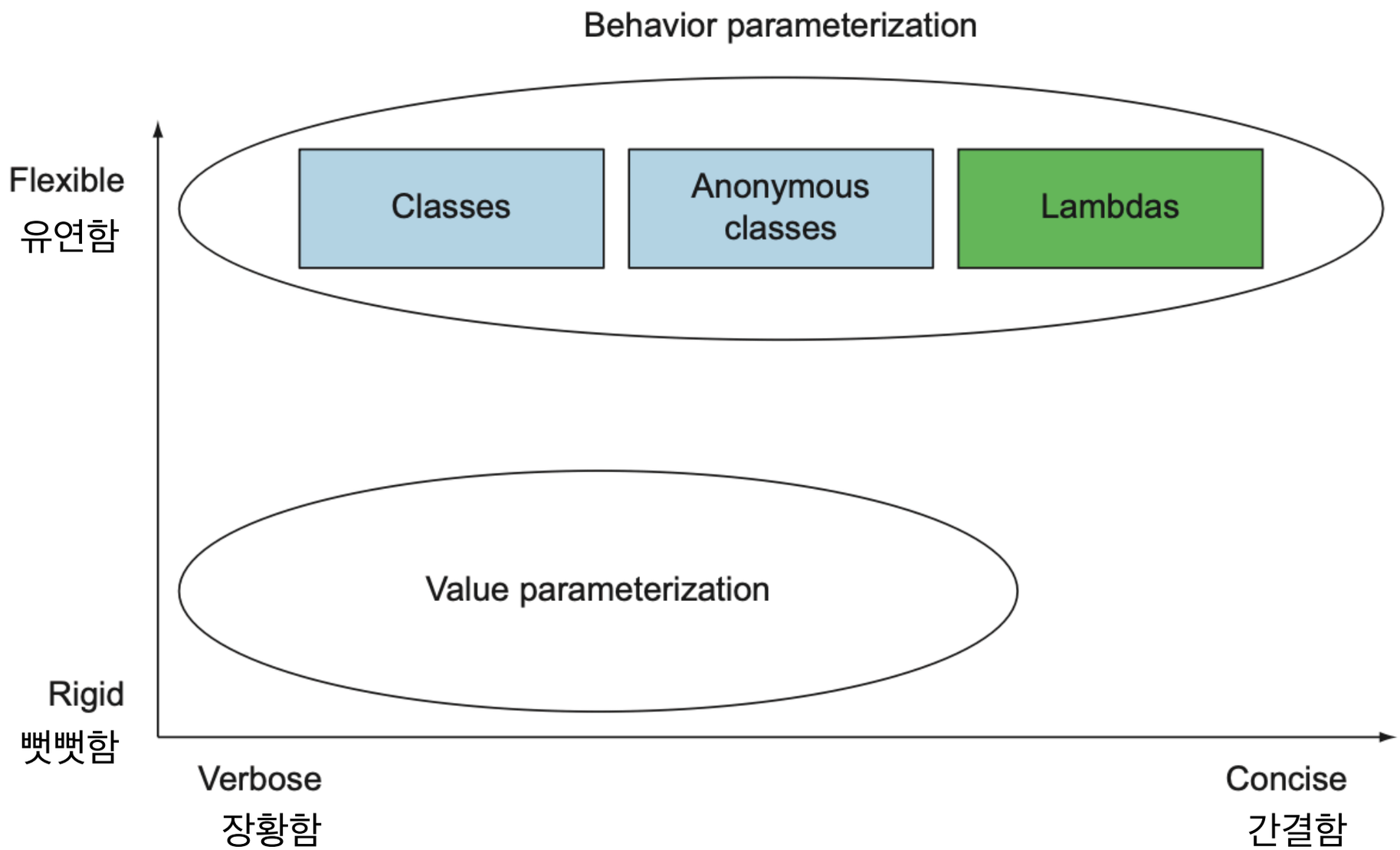
모던 자바 인 액션 2장

복잡한 과정 간소화 - 실습

여섯 번째 시도: 람다 표현식 이용

```
// 6. 여섯 번째 시도: 람다 표현식 사용
filterApples(inventory, (Apple apple) → RED == apple.getColor());
```

모던 자바 인 액션 2장


Behavior parameterization
Flexible
유연함
Classes
Anonymous classes
Lambdas
Value parameterization
Rigid
뻣뻣함
Verbose
장황함
Concise
간결함

```java
public class PracticalExample {  Complexity is 9 It's time to do something...

    public static void main(String[] args) {  Complexity is 8 It's time to do something...
        List<Apple> inventory = Arrays.asList(
            new Apple( weight: 80, GREEN),
            new Apple( weight: 155, GREEN),
            new Apple( weight: 120, RED));
        // 1. Comparator 로 정렬하기
        inventory.sort(new Comparator<Apple>() {  Complexity is 3 Everything is cool!
            @Override
            public int compare(final Apple o1, final Apple o2) {
                return Integer.compare(o1.getWeight(), o2.getWeight());
            }
        });
        // 2. 람다 표현식
        inventory.sort((Apple a1, Apple a2) -> Integer.compare(a1.getWeight(), a2.getWeight()));

        // 3. 메서드 참조
        inventory.sort(Comparator.comparingInt(Apple::getWeight));

        // 4. Runnable로 코드 블록 실행하기
        Thread t = new Thread(() -> System.out.println("Hello world"));

        // 5. Callable을 결과로 반환하기
        ExecutorService executorService = Executors.newCachedThreadPool();
        executorService.submit(() -> Thread.currentThread().getName()); // () -> V
    }
}
```

모던 자바 인 액션 3장

람다란 무엇인가?

람다 표현식은 메서드로 전달할 수 있는 익명 함수를 단순화한 것이라고 할 수 있습니다.

람다 표현식에 없는 것
- 이름

람다 표현식에 있는 것
- 파라미터 리스트
- 바디
- 반환 형식
- 발생할 수 있는 예외 리스트

람다를 이용하면 2장에서 살펴본 **동작 파라미터 형식**의 코드를 더 쉽게 구현할 수 있습니다.

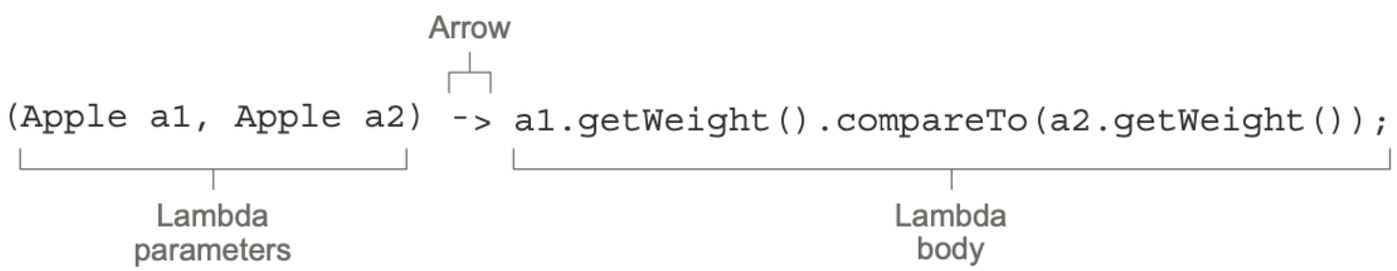
```
Comparator<Apple> byWeight = new Comparator<Apple>() { Complexity is 3 Everything is cool!
    @Override
    public int compare(final Apple o1, final Apple o2) {
        return Integer.compare(o1.getWeight(), o2.getWeight());
    }
};

Comparator<Apple> byWeight2 = (Apple a1, Apple a2) → Integer.compare(a1.getWeight(), a2.getWeight());
```

```java
Comparator<Apple> byWeight = new Comparator<Apple>() {  Complexity is 3 Everything is cool!
    @Override
    public int compare(final Apple o1, final Apple o2) {
        return Integer.compare(o1.getWeight(), o2.getWeight());
    }
};

Comparator<Apple> byWeight2 = (Apple a1, Apple a2) -> Integer.compare(a1.getWeight(), a2.getWeight());
```

람다의 구성



람다는 세 부분으로 이루어집니다.

파라미터 리스트
- comparator의 메서드 파라미터(사과 두 개)

화살표
- 화살표는 람다의 파라미터 리스트와 바디를 구분합니다

람다 바디
- 두 사과의 무게를 비교한다. 람다의 반환값에 해당하는 표현식입니다

모던 자바 인 액션 3장

Java8의 유효한 람다 표현식

```
(String s) -> s.length()
(Apple a) -> a.getWeight() > 150
(int x, int y) -> {
    System.out.println("Result:");
    System.out.println(x + y);
}
```

Takes one parameter of type String and returns an int. It has no return statement as return is implied.

Takes one parameter of type Apple and returns a boolean (whether the apple is heavier than 150 g).

Takes two parameters of type int and returns no value (void return). Its body contains two statements.

```
() -> 42
(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())
```

Takes two parameters of type Apple and returns an int representing the comparison of their weights

Takes no parameter and returns the int 42

(parameters) -> expression
(parameters) -> { statements; }

https://wisdom-and-record.tistory.com/65

모던 자바 인 액션 3장

람다 문법 퀴즈

```
1  () -> {}
2  () -> "Raoul"
3  () -> { return "Mario"; }
4  (Integer i) -> return "Alan" + i;
5  (String s) -> { "Iron Man"; }
```

앞에서 설명한 람다 규칙에 맞지 않은 람다 표현식을 고르시오.

모던 자바 인 액션 3장

람다 문법 퀴즈 - 답

```
1  () -> {}
2  () -> "Raoul"
3  () -> { return "Mario"; }
4  (Integer i) -> return "Alan" + i;
5  (String s) -> { "Iron Man"; }
```

4, 5번이 유효하지 않은 람다 표현식이다.

1. 파라미터가 없으며 void를 반환하는 람다 표현식이다.
2. 파라미터가 없으며 문자열을 반환하는 표현식이다.
3. 파라미터가 없으며 (명시적으로 return 문을 이용해서) 문자열을 반환하는 표현식이다.
4. return은 흐름 제어문이다. (control-flow statement) (Integer i) -> { return “Alan” + i; }; 처럼 되어야 올바른 람다 표현식이다.
5. “Iron Man”은 구문(statement)이 아니라 표현식(expression)이다. (String s) -> “Iron Man” 또는 (String s) -> {return “Iron Man”;} 처럼 명시적으로 return을 사용해야 한다.

모던 자바 인 액션 3장

어디에, 어떻게 람다를 사용할까?

함수형 인터페이스라는 문맥에서 람다 표현식을 사용할 수 있습니다.

```
@FunctionalInterface
public interface Predicate<T> {  Complexity is 7 It's time to do something...

    Evaluates this predicate on the given argument.
    Params:  t - the input argument
    Returns: true if the input argument matches the predicate, otherwise false
    boolean test(T t);
```

함수형 인터페이스로 뭘 할 수 있을까요?

람다 표현식으로 함수형 인터페이스의 추상 메서드 구현을 직접 전달할 수 있으므로 **전체 표현식을 함수형 인터페이스의 인스턴스로 취급**(기술적으로 따지면 함수형 인터페이스를 구현한 클래스의 인스턴스)할 수 있습니다.

모던 자바 인 액션 3장

어디에, 어떻게 람다를 사용할까?

```
public static void main(String[] args) {  Complexity is 5 Everything is cool!
    Runnable r1 = () → System.out.println("Hello World 1"); // 람다 사용
    Runnable r2 = new Runnable() { // 익명 클래스 사용
        @Override
        public void run() {
            System.out.println("Hello World 2");
        }
    };

    process(r1);
    process(r2);
    process(() → System.out.println("Hello World 3")); // 직접 전달된 람다 표현식 사용
}

public static void process(Runnable r) {  3 usages
    r.run();
}
```

모던 자바 인 액션 3장

함수 디스크립터

함수형 인터페이스의 추상 메서드 시그니처는 람다 표현식의 시그니처를 가리킵니다. 람다 표현식의 시그니처를 서술하는 메서드를 **함수 디스크립터**라고 부릅니다.

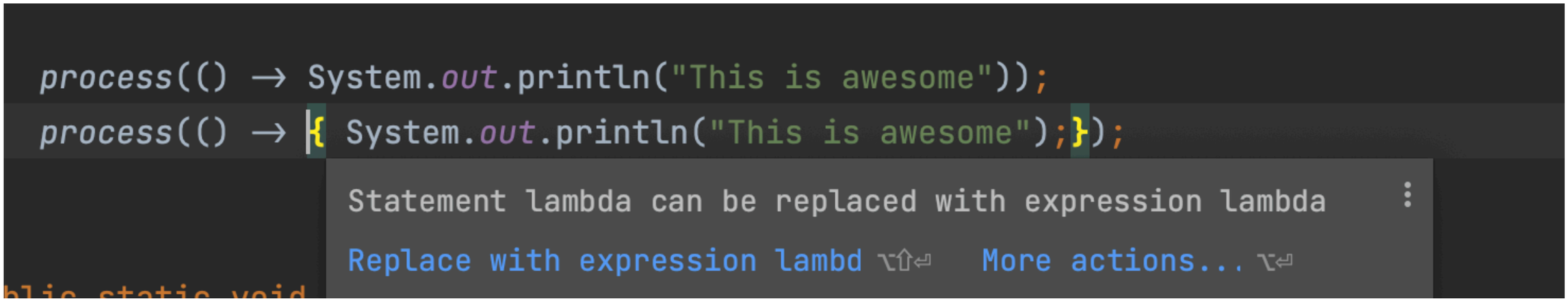
Runnable 인터페이스의 run은 인수와 반환값이 없는 시그니처로 생각할 수 있습니다.

```
public void process(Runnable r) {
    r.run();
}
process(() -> System.out.println("This is awesome!!"));
```

() -> System.out.println(“This is awesome!!”)은 인수가 없으며 void를 반환하는 람다 표현식입니다.
이는 Runnable 인터페이스의 run 메서드의 시그니처와 같습니다.

모던 자바 인 액션 3장

람다와 메서드 호출



자바 언어 명세에서는 void를 반환하는 메서드 호출과 관련한 특별한 규칙을 정하고 있습니다.

즉, 한 개의 void 메서드 호출은 중괄호로 감쌀 필요가 없습니다.

실행 어라운드 패턴 - 실습

1

```
public String processFile() throws IOException {
    try (BufferedReader br =
              new BufferedReader(new FileReader("data.txt"))){
        return br.readLine();
    }
}
```

2

```
public interface BufferedReaderProcessor {
  String process(BufferedReader b) throws IOException;
}

public String processFile(BufferedReaderProcessor p) throws
IOException {
   …
}
```

3

```
public String processFile(BufferedReaderProcessor p)
throws IOException {
   try (BufferedReader br =
             new BufferedReader(new FileReader("data.txt"))){
        return p.process(br);
   }
}
```

4

```
String oneLine = processFile((BufferedReader br) ->
                                  br.readLine());

String twoLines = processFile((BufferedReader br) ->
                                  br.readLine + br.readLine());
```

모던 자바 인 액션 3장

함수형 인터페이스 사용 - Predicate

Java8 라이브러리 설계자들은 java.util.funtion 패키지로 여러 가지 새로운 함수형 인터페이스를 제공합니다.

T 형식의 객체를 사용하는 불리언 표현식이 필요한 상황에서 Predicate 인터페이스를 사용할 수 있습니다.

```
public static void main(String[] args) {  Complexity is 3 Everything is cool!
    // 1. Predicate<T> T → boolean
    Predicate<String> nonEmptyStringPredicate = (String s) → !s.isEmpty();
    List<String> nonEmpty = filter(new ArrayList<>(), nonEmptyStringPredicate);
    List<String> nonEmptyWithLambda = filter(new ArrayList<>(), (String s) → !s.isEmpty());
}

public static <T> List<T> filter(List<T> list, Predicate<T> p) {  2 usages          Complexity is 4
    List<T> results = new ArrayList<>();
    for (T t : list) {
        if (p.test(t)) {
            results.add(t);
        }
    }
    return results;
}
```

모던 자바 인 액션 3장

함수형 인터페이스 사용 - Consumer

java.util.function.Consumer<T> 인터페이스는 제네릭 형식 T 객체를 받아서 void를 반환하는 accept라는 추상 메서드를 정의합니다.

T 형식의 인수를 받아서 어떤 동작을 수행하고 싶을 때 Consumer 인터페이스를 사용할 수 있습니다.

```
public static void main(String[] args) {
    forEach(
        Arrays.asList(1, 2, 3, 4, 5),
        (Integer i) -> System.out.println(i) // Consumer 의 accept 메서드를 구현하는 람다
    );
}

public static <T> void forEach(List<T> list, Consumer<T> c) {  1 usage
    for (T t : list) {
        c.accept(t);
    }
}
```

모던 자바 인 액션 3장

함수형 인터페이스 사용 - Function

java.util.function.Function<T, R> 인터페이스는 제네릭 형식 T를 인수로 받아서 제네릭 형식 R 객체를 반환하는 추상 메서드 apply를 정의합니다.

입력을 출력으로 매핑하는 람다를 정의할 때 Function 인터페이스를 활용할 수 있습니다.

```
// String 리스트를 인수로 받아 각 String의 길이를 포함하는 Integer 리스트로 변환하는 map 메서드
public static void main(String[] args) {
    List<Integer> stringLength = map(
        Arrays.asList("lambdas", "in", "action"),
        (String s) -> s.length()
    );
    System.out.println(stringLength); // [7, 2, 6]
}

public static <T, R> List<R> map(List<T> list, Function<T, R> f) {  1 usage
    List<R> result = new ArrayList<>();
    for (T t : list) {
        result.add(f.apply(t));
    }
    return result;
}
```

모던 자바 인 액션 3장

기본형 특화 함수형 인터페이스

자바의 모든 형식은 참조형 아니면 기본형에 해당합니다. 하지만 제네릭 파라미터에는 참조형만 사용할 수 있습니다.

자바에서는 기본형을 참조형으로 변환하는 기능을 제공합니다.

기본형 -> 참조형: **박싱(boxing)**
참조형 -> 기본형: **언박싱(unboxing)**

또한 프로그래머가 편리하게 코드를 구현할 수 있도록 박싱과 언박싱이 자동으로 이루어지는 **오토박싱**이라는 기능도 제공합니다.

```
List<Integer> list = new ArrayList<>();
for (int i = 300; i < 400; ++i) {
    list.add(i);
}
```

이러한 변환 과정은 비용이 소모됩니다.
[박싱한 값은 기본형을 감싸는 래퍼이며 힙에 저장됩니다. 따라서 박싱한 값은 메모리를 더 소비하며 기본형을 가져올 때도 메모리를 탐색하는 과정이 필요합니다.

따라서 Java 8에서는 기본형을 입출력으로 사용하는 상황에서 오토박싱 동작을 피할 수 있도록 특별한 버전의 함수형 인터페이스를 제공합니다. ex) DoublePredicate, IntConsumer 처럼 앞에 형식명이 붙습니다.

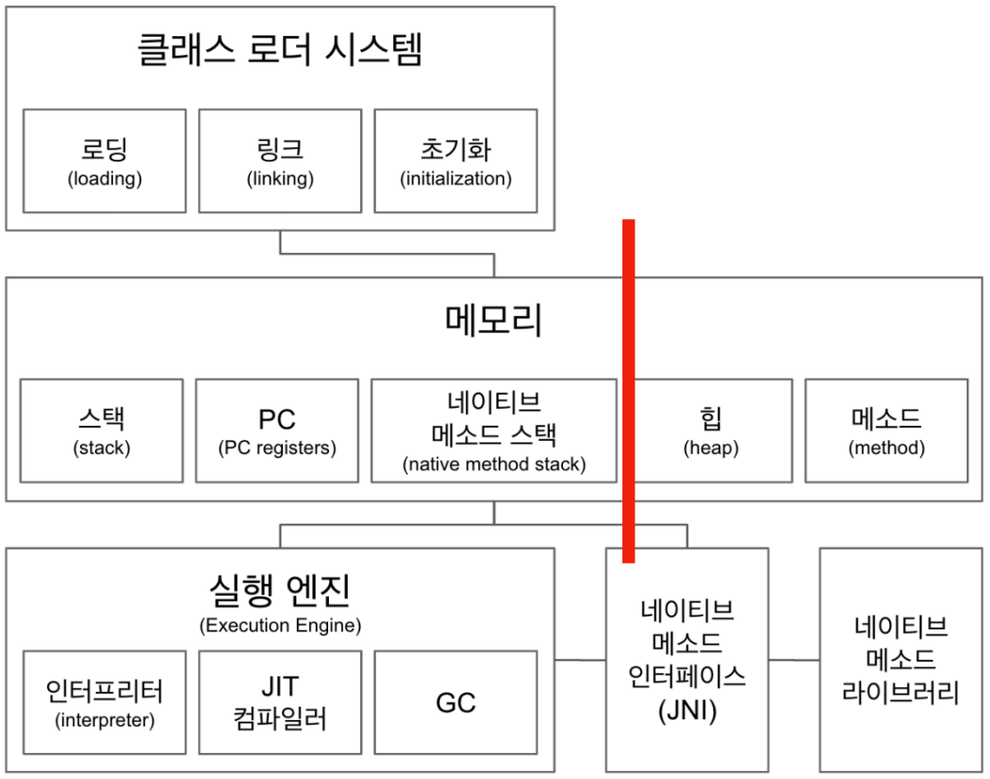
`Predicate<T>`	`T -> boolean`	`IntPredicate, LongPredicate, DoublePredicate`
`Consumer<T>`	`T -> void`	`IntConsumer, LongConsumer, DoubleConsumer`
`Function<T, R>`	`T -> R`	`IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, DoubleToIntFunction, DoubleToLongFunction, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>`
`Supplier<T>`	`() -> T`	`BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier`
`UnaryOperator<T>`	`T -> T`	`IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator`
`BinaryOperator<T>`	`(T, T) -> T`	`IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator`
`BiPredicate<T, U>`	`(T, U) -> boolean`	
`BiConsumer<T, U>`	`(T, U) -> void`	`ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>`
`BiFunction<T, U, R>`	`(T, U) -> R`	`ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>`

모던 자바 인 액션 3장

람다 연습문제 풀어보기

https://www.w3resource.com/java-exercises/lambda/index.php

effectively final



람다식 내부에서 외부 지역 변수를 참조하는 경우 final 또는 **effectively final**이어야 한다.
이러한 이유는 지역 변수가 메모리 영역중 스택(Stack) 영역에 할당되는 것과 관련이 있다.

스택 영역은 스레드 별로 고유하기 때문에 지역 변수가 할당된 스레드가 종료되면 지역 변수를 더이상 참조하지 못하게 된다.

따라서 별도 스레드에서 실행 가능한 람다에서는 외부 지역 변수를 복사하는 과정을 거치는데, 복사되는 값이 변경 가능하다면 참조하는 변수의 최신값을 보장할 수 없어 멀티 스레드 환경에서 동시성 문제가 발생할 수 있다.

따라서, 람다 내부에서 외부 지역 변수를 참조할 때는 반드시 final 또는 **effectively final**이어야 한다.
https://madplay.github.io/post/effectively-final-in-java

effectively final

effectively final

effectively final

그렇다면 정확히 어떤 경우를 effectively final이라고 말하는 것일까? 자바 언어 스펙을 살펴보면 다음과 같은 조건을 만족하는 지역 변수(local variables)는 effectively final로 간주한다.

- `final`로 선언되지 않았다.
- 초기화를 진행한 후에 다시 할당하지 않았다.
- 전위(prefix) 또는 후위(postfix)에 증감 또는 감소 연산자가 사용되지 않았다.

참고: “Java Docs: 4.12.4. final Variables”

객체의 경우에는 객체가 가리키는 참조를 변경하지 않으면 된다. 따라서 아래와 같이 객체의 상태를 변경하더라도 effectively final이다.

```
List<Person> personList = List.of(new Person(2), new Person(3));
for (Person p : personList) {
    p.setId(2);
    personList.removeIf(o -> o.getId() == p.getId());
}
```

출처

Freepik
https://www.freepik.com/free-vector/hand-drawn-flat-design-overwhelmed-people-illustration_23669506.htm#query=developer%20stress&position=1&from_view=search&track=ais

https://wisdom-and-record.tistory.com/65
https://madplay.github.io/post/effectively-final-in-java

모던 자바 인 액션 2장, 3장
이펙티브 자바 아이템 44

감사합니다