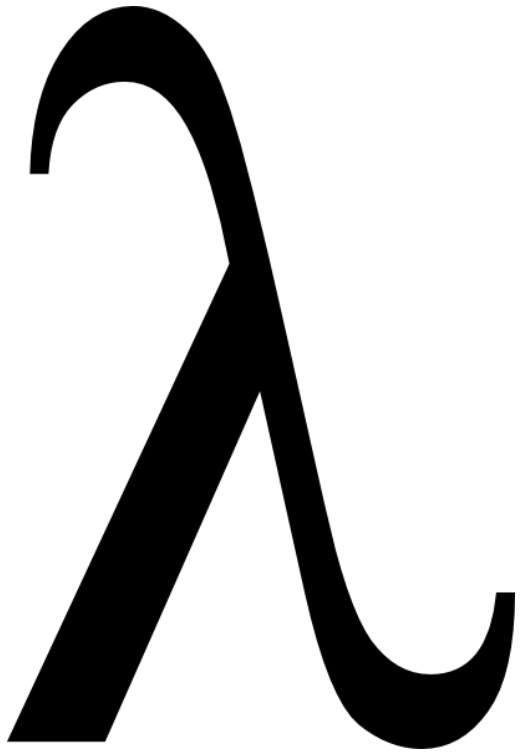
Chapter 03.
람다 표현식

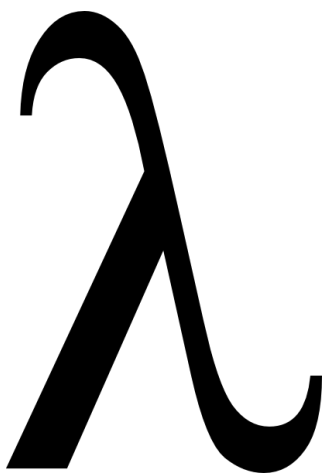
2019. 08. 27 | Java in Action Seminar | 박대원



wisoft

이 장의 내용

- 람다란 무엇인가?
- 어디에, 어떻게 람다를 사용하는가?
- 실행 어라운드 패턴
- 함수형 인터페이스, 형식 추론
- 메서드 참조
- 람다 만들기



1.

람다란 무엇인가?

람다 표현식은 메서드로 전달할 수 있는 익명 함수를 단순화 한 것.

- **익명**
  보통의 메서드와 달리 이름이 없으므로 **익명**이라 표현한다.

- **함수**
  람다는 메서드처럼 특정 클래스에 종속되지 않으므로 함수라고 부른다.
  하지만 메서드처럼 파라미터 리스트, 바디, 반환 형식, 가능한 예외 리스트를 포함한다.

- **전달**
  람다 표현식을 메서드 인수로 전달하거나 변수로 저장할 수 있다.

- **간결성**
  익명 클래스처럼 많은 자질구레한 코드를 구현할 필요가 없다.

1. 람다란 무엇인가?

예제

> 기존 코드

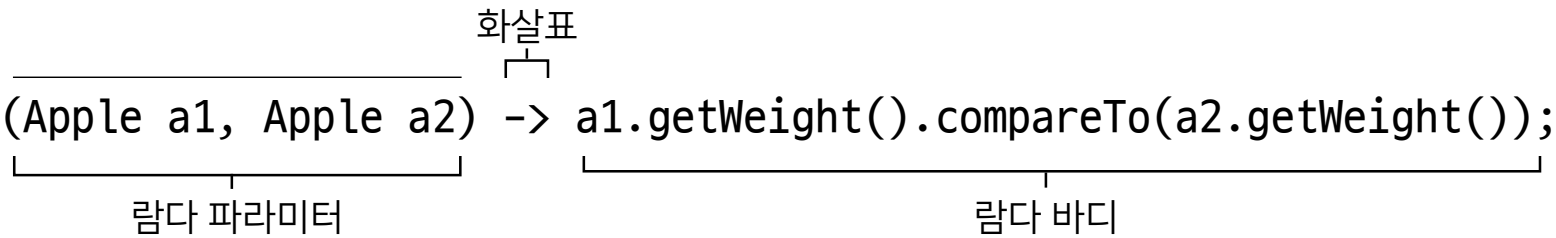
```
Comparator<Apple> byWeight = new Comparator<Apple>() {
  public int compare(Apple a1, Apple a2) {
    return a1.getWeight().compareTo(a2.getWeight());
  }
}
```

> 람다를 이용한 코드

```
Comparator<Apple> byWeight =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
}
```

1. 람다란 무엇인가?

람다 표현식의 구성



- **파라미터 리스트**
  Comparator의 compare 메서드 파라미터(여기서는 사과 두 개).
- **화살표**
  화살표(->)는 람다의 파라미터 리스트와 바디를 구분한다.
- **람다 바디**
  두 사과의 무게를 비교한다. 람다의 반환값에 해당하는 표현식이다.

1. 람다란 무엇인가?

자바 8의 유효한 람다 표현식

- (String s) -> s.length()
  String 형식의 파라미터를 하나 가지며 int를 반환한다.
  람다 표현식에는 return이 함축되어 있으므로 명시적으로 사용하지 않아도 된다.

- (Apple a) -> a.getWeight() > 150
  Apple 형식의 파라미터 하나를 가지며 boolean을 반환한다.

- (int x, int y) -> {
  System.out.println("Result : ");
  System.out.println(x + y);
  }
  int 형식의 파라미터 두 개를 가지며 리턴값이 없다(void 리턴). 여러 행의 문장을 포함할 수 있다.

- () -> 42
  파라미터가 없으며 int 42를 반환한다.

- (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
  Apple 형식의 파라미터 두 개를 가지며 int(두 사과의 무게 비교 결과)를 반환한다.

1. 람다 예제와 사용 사례

람다란 무엇인가?

사용 사례	람다 예제
boolean 표현식	(List<String> list) -> list.isEmpty()
객체 생성	() -> new Apple(10)

객체에서 소비	(Apple a) -> {
	System.out.println(a.getWeight());
	}

객체에서 선택/추출	(String s) -> s.length()
두 값을 조합	(int a, int b) -> a * b
두 객체 비교	(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())

2.
어디에, 어떻게 람다를 사용할까?

이전 예제

> Comparator

```
Comparator<Apple> byWeight =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
}
```

> filter 메서드

```
List<Apple> greenApples =
  filter(inventory, (Apple a) -> GREEN.equals(a.getColor()));
```

Predicate<Apple>을 기대하는 filter 메서드의 두 번째 인자.

2. 어디에, 어떻게 람다를 사용할까?

2.1 함수형 인터페이스

정확히 하나의 **추상 메서드**를 지정하는 인터페이스.

> Predicate<T>

```
1 public interface Predicate<T> {
2   boolean test (T t);
3 }
```

인터페이스에 많은 디폴트 메서드가 있더라도

추상 메서드가 오직 하나면 함수형 인터페이스다.

함수형 인터페이스로 뭘 할 수 있을까?

람다 표현식으로 함수형 인터페이스의 추상 메서드 구현을 직접 전달할 수 있으므로

전체 표현식을 함수형 인터페이스의 인스턴스로 취급 할 수 있다.

2.
어디에, 어떻게 람다를 사용할까?

2.1 함수형 인터페이스

> 예제 : Runnable 함수형 인터페이스 사용

```
1  Runnable r1 = () -> System.out.println("Hello World 1"); //람다 사용
2
3  Runnable r2 = new Runnable() {                           //익명 클래스 사용
4    public void run() {
5      System.out.println("Hello World 2");
6    }
7  };
8
9  public static void process(Runnable r) {
10   r.run();
11 }
12
13 process(r1);
14 process(r2);
15 process(() -> System.out.println("Hello World 3"));      //직접 람다식 전달
```

2.
어디에, 어떻게 람다를 사용할까?

2.2 함수 디스크립터

함수형 인터페이스의 추상 메서드 시그니처는 람다 표현식의 시그니처를 가리킨다.
람다 표현식의 시그니처를 서술하는 메서드를 **함수 디스크립터**라고 부른다.

> Runnable

```
1 public interface Runnable {
2   void run();
3 }
```

run 메서드는 인수와 반환값이 없으므로(void 반환) Runnable 인터페이스는 인수와 반환값이 없는 시그니처.
이 장에서는 람다와 함수형 인터페이스를 가리키는 특별한 표기법으로 () -> void와 같이 표기한다.

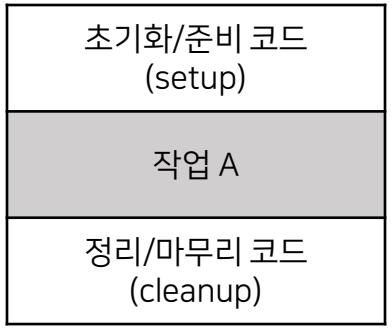
@FunctionalInterface는 무엇인가?

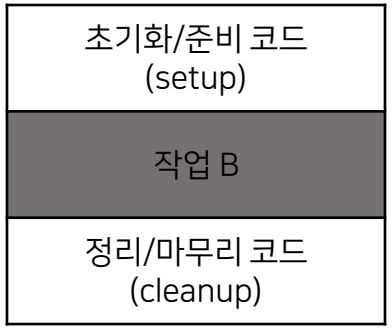
함수형 인터페이스임을 가리키는 어노테이션. 실제로 함수형 인터페이스가 아니라면 컴파일러가 에러를 발생시킨다.

3.
람다 활용 : 실행 어라운드 패턴

들어가기

순환 패턴(recurrent pattern)은 자원을 열고, 처리한 다음에, 자원을 닫는 순서로 이루어진다.
설정(setup)과 정리(cleanup)과정은 대부분 비슷하다.





즉, 실제 자원을 처리하는 코드를 설정과 정리 두 과정이 둘러싸는 형태를 갖는 형식의 코드를
실행 어라운드 패턴(execute around pattern)이라고 부른다.

```
1  public static String processFile() throws IOException {
2    try (BufferedReader br =
3              new BufferedReader(new FileReader("data.txt"))) {
4      return br.readLine();
5    }
6  }
```

3.
람다 활용 : 실행 어라운드 패턴

3.1 1단계 : 동작 파라미터화를 기억하라

```
1  public static String processFile() throws IOException {
2    try (BufferedReader br =
3              new BufferedReader(new FileReader("data.txt"))) {
4      return br.readLine();
5    }
6  }
```

processFile 메서드가 BufferdReader를 이용해서 다른 동작을 수행할 수 있도록 메서드에 동작을 전달해야 한다.

람다를 이용해서 동작을 전달할 수 있다. 우선 BufferdReader를 인수로 받아서 String을 반환하는 람다를 추가하자.

```
1 String result = processFile(
2   (BufferedReader br) -> br.readLine() + br.readLine());
```

3.
람다 활용 : 실행 어라운드 패턴

3.2 2단계 : 함수형 인터페이스를 이용해서 동작 전달

함수형 인터페이스 자리에 람다를 사용할 수 있다. 따라서 BufferdReader -> String과 IOException을 던질 수 있는 시그니처와 일치하는 함수형 인터페이스를 만들어야 한다.

```
1  @FunctionalInterface
2  public interface BufferedReaderProcessor {
3    String process(BufferedReader b) throws IOException;
4  }
```

정의한 인터페이스를 processFile 메서드의 인수로 전달할 수 있다.

```
1  public String processFile(BufferedReaderProcessor p) throws IOException {
2    ...
3  }
```

3.
람다 활용 : 실행 어라운드 패턴

3.3 3단계 : 동작 실행

이제 process 메서드의 시그니처(BufferedReader -> String)와 일치하는 람다를 전달할 수 있다.

```
1  public String processFile(BufferedReaderProcessor p) throws IOException {
2    try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {
3      return p.process(br);    // BufferedReader 객체 처리
4    }
5  }
```

람다 표현식으로 함수형 인터페이스의 추상 메서드 구현을 직접 전달할 수 있으며

전달된 코드는 함수형 인터페이스의 인스턴스로 전달된 코드와 같은 방식으로 처리한다.

따라서 processFile 바디 내에서 BufferedReader 객체의 process를 호출할 수 있다.

3.
람다 활용 : 실행 어라운드 패턴

3.4 4단계 : 람다 전달

이제 람다를 이용해서 다양한 동작을 processFile 메서드에 전달할 수 있다.

> 한 행을 처리하는 코드

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

> 두 행을 처리하는 코드

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

4.
함수형 인터페이스 사용

들어가기

함수형 인터페이스의 추상 메서드는 람다 표현식의 시그니처를 묘사한다.

함수형 인터페이스의 추상 메서드 시그니처를 **함수 디스크립터(function descriptor)**라고 한다.

Predicate? Consumer? Function?

4.
함수형 인터페이스 사용

4.1 Predicate

java.util.function.Predicate<T> 인터페이스는 test라는 추상 메서드를 정의하며
test는 제네릭 형식의 T의 객체를 인수로 받아 불리언을 반환한다.

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

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

Predicate<String> nonEmptyStringPredicate = (Stirng s) -> !s.isEmpty();
List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);

```

4.
함수형 인터페이스 사용

4.2 Consumer

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

```java
@FunctionalInterface
public interface Consumer<T> {
  void accept(T t);
}

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

forEach(
  Arrays.asList(1, 2, 3, 4, 5),
  (Integer i) -> System.out.println(i)  // Consumer의 accept 메소드를 구현하는 람다
);
```

4.
함수형 인터페이스 사용

4.3 Function

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

```java
@FunctionalInterface
public interface Function<T, R> {
  R apply(T t);
}

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

List<Integer> l = map(
  Arrays.asList("lambdas", "in", "action"),
  (String s) -> s.length()  // Function의 apply 메서드를 구현하는 람다
);
```

4.
함수형 인터페이스 사용

기본형 특화

기본형을 입출력으로 사용하는 상황에서 오토박싱을 피할 수 있도록 특별한 함수형 인터페이스를 제공한다.

> 예시 : IntPredicate

```
1  public interface IntPredicate {
2    boolean test(int t);
3  }
4
5  IntPredicate evenNumber = (int i) -> i % 2 == 0;
6  evenNumbers.test(1000);   // 박싱 없음
7
8  predicate<Integer> oddNumbers = (Integer i) -> i % 2 != 0;
9  oddNumbers.test(1000);    // 박싱
```

4.
함수형 인터페이스 사용

자바 8에 추가된 함수형 인터페이스

함수형 인터페이스	함수 디스크립터	기본형 특화
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<L, R>	(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>

4.
함수형 인터페이스 사용

람다와 함수형 인터페이스 예제

사용 사례	람다 예제	대응하는 함수형 인터페이스
불리언 표현	(List<String> list) -> list.isEmpty()	Predicate<List<String>>
객체 생성	() -> new Apple(10)	Supplier<Apple>
객체에서 소비	(Apple a) -> System.out .println(a.getWeight())	Consumer<Apple>
객체에서 선택/추출	(String s) -> s.length()	Function<String, Integer> 또는 ToIntFunction<String>
두 값 조합	(int a, int b) -> a * b	IntBinaryOperator
두 객체 비교	(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())	Comparator<Apple> 또는 BiFunction<Apple, Apple, Integer> 또는 ToIntBiFunction<Apple, Apple>

5.
형식 검사, 형식 추론, 제약

5.1 형식 검사

람다가 사용되는 콘텍스트(context, 람다가 전달될 메서드 파라미터나 람다가 할당되는 변수 등)를 이용해서 람다의 형식을 추론할 수 있다. 어떤 콘텍스트에서 기대되는 람다 표현식의 형식을 **대상 형식(target type)**이라 한다.

```
List<Apple> heavierThan150g =
  filter(inventory, (Apple apple) -> apple.getWeight() > 150);
```

1. 람다가 사용된 콘텍스트는 무엇인가? 우선 filter의 정의를 확인하자.

   filter(List<Apple> inventory, Predicate<Apple> p)
2. 대상 형식은 Predicate<Apple>이다. (T는 Apple로 대치됨.)
3. Predicate<Apple>의 추상 메서드는 무엇인가?

   boolean test(Apple apple)
4. Apple을 인수로 받아 boolean을 반환하는 test 메서드다!

   Apple -> boolean
5. 함수 디스크립터는 Apple -> boolean이므로 람다의 시그니처와 일치한다. 코드 형식 검사가 완료된다.

5.
형식 검사, 형식 추론, 제약

5.2 같은 람다, 다른 함수형 인터페이스

대상 형식이라는 특징 때문에 같은 람다식이더라도 호환되는 다른 함수형 인터페이스로 사용될 수 있다.

```
Comparator<Apple> c1 =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
ToIntBiFunction<Apple, Apple> c2 =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
BiFunction<Apple, Apple, Integer> c3 =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

다이아몬드 연산자(<>)

다이아몬드 연산자로 콘텍스트에 따른 제네릭 형식을 추론할 수 있다.

```
List<String> listOfStrings = new ArrayList<>();
List<Integer> listOfIntegers = new ArrayList<>();
```

특별한 void 호환 규칙

람다의 바디에 일반 표현식이 있으면 void를 반환하는 함수 디스크립터와 호환된다.

```
Predicate<String> p = s -> list.add(s);  // Predicate는 불리언 반환값을 갖는다.
Consumer<String> b = s -> list.add(s);   // Consumer는 void 반환값을 갖는다.
```

5.
형식 검사, 형식 추론, 제약

5.3 형식 추론

대상 형식을 이용해서 함수 디스크립터를 알 수 있으므로 컴파일러는 람다의 시그니처도 추론할 수 있다.
결과적으로 컴파일러는 람다 표현식의 파라미터 형식에 접근할 수 있으므로 람다 문법에서 이를 생략 가능.

```
// 파라미터 apple에는 형식을 명시적으로 지정하지 않았다.
List<Apple> greenApples =
  filter(inventory, apple -> GREEN.equals(apple.getColor()));
```

여러 파라미터를 포함하는 람다 표현식에서는 코드 가독성 향상이 더 두드러진다.

```
// 형식을 추론하지 않음
Comparator<Apple> c =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
// 형식을 추론
Comparator<Apple> c =
  (a1, a2) -> a1.getWeight().compareTo(a2.getWeight());
```

상황에 따라 명시적으로 형식을 포함하는 것이 좋을 때도 있고 형식을 배제하는 것이 가독성을 향상시킬 때도 있다.

5.4 지역 변수 사용

람다 표현식에서는 익명 함수가 하는 것처럼 **자유 변수(파라미터로 넘겨진 변수가 아닌 외부에서 정의된 변수)**를 활용할 수 있다. 이와 같은 동작을 **람다 캡처링(capturing lambda)** 이라고 부른다.

```
1  int portNumber = 1337;
2  Runnable r = () -> System.out.println(portNumber);
```

람다는 인스턴스, 정적 변수를 자유롭게 캡처링 할 수 있지만, 그러려면 지역 변수는 명시적으로 final로 선언되거나 final처럼 사용되어야 한다. 즉, 람다식은 한 번만 할당할 수 있는 지역 변수를 캡처할 수 있다.

```
1  int portNumber = 1337;
2  Runnable r = () -> System.out.println(portNumber);
3  portNumber = 31337;
```

지역 변수의 제약

인스턴스 변수는 스레드가 공유하는 힙에 저장되지만, 지역 변수는 스택에 위치한다. 람다에서 지역 변수에 바로 접근할 수 있다는 가정하에 람다가 스레드에서 실행된다면 변수를 할당한 스레드가 사라져 해제되어도 람다를 실행하는 스레드에서는 해당 변수에 접근하려 할 수 있다. 따라서 자바 구현에서는 자유 지역 변수의 복사본을 제공한다. 복사본의 값이 바뀌지 않아야 하므로 지역 변수에는 한번만 할당하는 제한이 생긴 것이다.

6.

메서드 참조

6.1 요약

> 기존 코드

```
1 inventory.sort((Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

> 메서드 참조와 java.util.Comparator.comparing 사용

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

메서드 참조 예제

람다	메서드 참조 단축 표현
(Apple apple) -> apple.getWeight()	Apple::getWeight
() -> Thread.currentThread().dumpStack()	Tread.currentThread()::dumpStack
(str, i) -> str.substring(i)	String::substring
(String s) -> System.out.println(s)	System.out::println
(String s) -> this.isValidName(s)	this::isValidName

6.

메서드 참조

메서드 참조를 만드는 법

메서드 참조는 세가지 유형으로 구분할 수 있다.

정적 메서드 참조 : 예를 들어 Integer의 parseInt 메서드는 Integer::parseInt로 표현할 수 있다.

다양한 형식의 인스턴스 메서드 참조 : 예를 들어 String의 length 메서드는 String::length로 표현할 수 있다.

기존 객체의 인스턴스 메서드 참조 : 예를 들어 Transaction 객체를 할당받은 expensiveTransaction 지역 변수가 있고 Transaction 객체에는 getValue 메서드가 있다면, 이를 expensiveTransaction::getValue라고 표현할 수 있다.

컴파일러는 람다 표현식의 형식을 검사하던 방식과 비슷한 과정으로 메서드 참조가 주어진 함수형 인터페이스와 호환하는지 확인한다. 즉, 메서드 참조는 콘텍스트의 형식과 일치해야 한다.

6.

메서드 참조

6.2 생성자 참조 (1/2)

ClassName::new 처럼 기존 생성자의 참조를 만들 수 있다. 이것은 정적 메서드의 참조를 만드는 것과 비슷하다.

```
1 Supplier<Apple> c1 = Apple::new;
2 //Supplier의 get 메서드를 호출해서 새로운 Apple 객체를 만들 수 있다.
3 Apple a1 = c1.get();
```

```
1 // 람다 표현식은 디폴트 생성자를 가진 Apple을 만든다.
2 Supplier<Apple> c1 = () -> new Apple();
3 // 새로운 Apple 객체 생성
4 Apple a1 = c1.get();
```

6.

메서드 참조

6.2 생성자 참조 (2/2)

Apple(Color color, Integer weight) 처럼 두 인수를 갖는 생성자는 BiFunction 인터페이스와 같은 시그니처를 가지므로 다음처럼 사용할 수 있다.

```
BiFunction<Color, Integer, Apple> c3 = Apple::new;
Apple a3 = c3.apply(GREEN, 110);
```

인스턴스화하지 않고도 생성자에 접근할 수 있는 기능을 다양한 상황에 응용할 수 있다.

```
static Map<String, Function<Integer, Fruit>> map = new HashMap<>();
static {
  map.put("apple", Apple::new);
  map.put("orange", Orange::new);
  // 등등
}

public static Fruit giveMeFruit(String fruit, Integer weight) {
  return map.get(fruit.toLowerCase())   // map에서 Function<Integer, Fruit> 을 얻는다.
    .apply(weight);   // apply 메서드에 무게 파라미터를 제공해서 Fruit를 만들 수 있다.
}
```

7.
람다, 메서드 참조 활용하기

7.1 1단계 : 코드 전달

List API의 sort메서드 시그니처

void sort(Comparator<? super E> c)

> 1단계 완성 코드

```
1 public class AppleComparator implements Comparator<Apple> {
2   public int compare(Apple a1, Apple a2) {
3     return a1.getWeight().compareTo(a2.getWeight());
4   }
5 }
6 inventory.sort(new AppleComparator());
```

이제 sort의 **동작**은 **파라미터화** 되었다고 할 수 있다.

7.
람다, 메서드 참조 활용하기

7.2 2단계 : 익명 클래스 사용

한번만 사용할 Comparator를 위 코드처럼 구현하는 것보다는 **익명 클래스**를 이용하는 것이 좋다.

> 2단계 완성 코드

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

7.
람다, 메서드 참조 활용하기

7.3 3단계 : 람다 표현식 사용

자바 8에서는 람다 표현식이라는 경량화된 문법을 이용해서 **코드를 전달**할 수 있다.

함수형 인터페이스를 기대하는 곳 어디에서나 람다 표현식을 사용할 수 있다.

Comparator의 함수 디스크립터는 (T, T) -> int다.

> 3단계 - 1

```
inventory.sort(
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

> 3단계 - 2 람다의 파라미터 형식 추론을 통해 코드를 더 줄일 수 있다.

```
inventory.sort((a1, a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

> 3단계 - 3 Comparator는 Comparable 키를 추출해서 Comparator 객체를 만들면 더욱 가독성을 향상시킬 수 있다.

```
Comparator<Apple> c = Comparator.comparing((Apple a) -> a.getWeight());
```

> 3단계 완성 코드

```
inventory.sort(comparing(apple -> apple.getWeight()));
```

7.
람다, 메서드 참조 활용하기

7.4 4단계 : 메서드 참조 사용

메서드 참조를 사용하여 코드를 조금 더 간소화 할 수 있다.

> 4단계 완성 코드

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

단지 코드만 짧아진 것이 아니라 코드의 의미도 명확해졌다.

즉, 코드 자체로 **'Apple을 weight별로 비교해서 inventory를 sort하라'**는 의미를 전달할 수 있다.

8.

람다 표현식을 조합할 수 있는 유용한 메서드

Comparator, Function, Predicate 같은 함수형 인터페이스는 람다 표현식을 조합할 수 있도록 유틸리티 메서드를 제공한다. 즉, 간단한 여러 개의 람다 표현식을 조합해서 복잡한 람다 표현식을 만들 수 있다.

여기서 등장하는 것이 바로 **디폴트 메서드**(default method)다.

8.1 Comparator 조합

정적 메서드 comparing을 이용해서 비교에 사용할 키를 추출하는 Function 기반의 Comparator를 반환할 수 있다.

```
1 Comparator<Apple> c = Comparator.comparing(Apple::getWeight);
```

역정렬

인터페이스 자체에서 비교자의 순서를 바꾸는 reversed라는 디폴트 메서드를 제공하기 때문에, 역정렬을 할 수 있다.

```
1 inventory.sort(comparing(Apple::getWeight).reversed());   // 무게를 내림차순으로 정렬
```

Comparator 연결

thenComparing 메서드로 두 번째 비교자를 만들 수 있다. thenComparing은 함수를 메서드로 받아 첫 번째 비교자를 이용해서 두 객체가 같다고 판단되면 두 번째 비교자에 객체를 전달한다.

```
1 inventory.sort(comparing(Apple::getWeight)
2                .reversed()                          // 무게를 내림차순으로 정렬
3                .thenComparing(Apple::getCountry));// 두 사과의 무게가 같으면 국가별로 정렬
```

8.

람다 표현식을
조합할 수 있는
유용한 메서드

8.2 Predicate 조합

Predicate 인터페이스는 복잡한 프레디케이트를 만들 수 있도록 **negate, and, or** 세 가지 메서드를 제공한다.

> negate 메서드

```
// 기존 프레디케이트 객체 redApple의 결과를 반전시킨 객체를 만든다.
Predicate<Apple> notRedApple = redApple.negate();
```

> and 메서드

```
// 두 프레디케이트를 연결해서 새로운 프레디케이트 객체를 만든다.
Predicate<Apple> redAndHeavyApple =
  redApple.and(apple -> apple.getWeight() > 150);
```

> or 메서드

```
Predicate<Apple> redAndHeavyAppleOrGreen =
  redApple.and(apple -> apple.getWeight() > 150)
          .or(apple -> GREEN.equals(apple.getColor()));
```

8.

람다 표현식을
조합할 수 있는
유용한 메서드

8.2 Predicate 조합

Predicate 인터페이스는 복잡한 프레디케이트를 만들 수 있도록 **negate, and, or** 세 가지 메서드를 제공한다.

> negate 메서드

```
// 기존 프레디케이트 객체 redApple의 결과를 반전시킨 객체를 만든다.
Predicate<Apple> notRedApple = redApple.negate();
```

> and 메서드

```
// 두 프레디케이트를 연결해서 새로운 프레디케이트 객체를 만든다.
Predicate<Apple> redAndHeavyApple =
  redApple.and(apple -> apple.getWeight() > 150);
```

> or 메서드

```
Predicate<Apple> redAndHeavyAppleOrGreen =
  redApple.and(apple -> apple.getWeight() > 150)
          .or(apple -> GREEN.equals(apple.getColor()));
```

8.

람다 표현식을
조합할 수 있는
유용한 메서드

8.3 Function 조합

Function 인터페이스는 Function 인스턴스를 반환하는 **andThen, compose** 두 가지 디폴트 메서드를 제공한다.

> andThen 메서드

```java
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.andThen(g);
int result = h.apply(1);  // 4를 반환, g(f(x)) 와 같다.
```

> compose 메서드

```java
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.compose(g);
int result = h.apply(1);  // 3을 반환, f(g(x)) 와 같다.
```

9.

마치며

- **람다 표현식**은 익명 함수의 일종이다. 이름은 없지만, 파라미터 리스트, 바디, 반환 형식을 가지며 예외를 던질 수 있다.
- **함수형 인터페이스**는 하나의 추상 메서드만을 정의하는 인터페이스다.
- 람다 표현식을 이용해서 함수형 인터페이스의 추상 메서드를 즉석으로 제공할 수 있으며

  람다 표현식 전체가 함수형 인터페이스의 인스턴스로 취급된다.
- Java.util.function 패키지는 **Predicate<T>, Function<T, R>, Supplier<T>, Consumer<T>, BinaryOperator<T>** 등을

  포함해서 자주 사용하는 다양한 함수형 인터페이스를 제공한다.
- 자바 8은 **기본형 특화 인터페이스도 제공한다.**
- **실행 어라운드 패턴**을 람다와 활용하면 유연성과 재사용성을 추가로 얻을 수 있다.
- Comparator, Predicate, Function 같은 함수형 인터페이스는 람다 표현식을 조합할 수 있는 **다양한 디폴트 메서드를 제공한다.**

감 사 합 니 다