JAVA Lambda

프로그램은 사람이 이해하는 코드를 작성.

느려도 꾸준하면 경기에서 이긴다.

작성자 : 홍효상
이메일 : hyomee@naver.com
소스 : https://github.com/hyomee/JAVA_EDU

Content

16. Lambda

1. 익명(Anonymous) Class
2. Lambda
3. 함수형 인터페이스
4. 타입 추론

1. 익명(Anonymous) Class

" 이름을 알 수 없는 객체로 한번만 사용하고 버려지는 객체"

사용하는 이유

- 프로그램에서 일시적으로 한번만 사용되고 버려지는 객체를 매번 객체를 만들어야 하나 ?
- 확장성을 고려해서 객체를 생성 해야 하는데 ... 수정이 편 할 까 ?
- 사용 처 : 인스턴스 변수, 인스턴스 메서드, 인스턴스 메소드의 매개변수

구현 하는 방법

- 클래스 생성
- 인터페이스의 구현

익명(Anonymous) Class

- 클래스를 정의하지 않고 필요할 때 이름없이 즉시 선언하고 인스턴스화 해서 사용
- 객체 안에 만드는 로컬 클래스와 동일 하다
- new 수식이 올 수 있는 곳 어디든지 사용 가능하나 생성자는 정의 할 수 없음
- 익명 클래스내부에서 외부의 메소드 내 변수를 참조할 때는 메소드의 지역 변수 중 final로 선언된 변수만 참조 가능
  - 변수는 Stack에 있고 객체는 Heap에 있음, 즉 Method 실행 이 끝나고 Stack는 사라지지만 Heap에 있는 Method는 사라지지 않기 때문

01. 클래스 생성

➢ 1. 추상 클래스 생성

예제 : AnonymousAbstract.java

```
abstract class AnonymousAbstract {
  abstract void doWork();
  abstract void doWork(String str);
}
```

➢ 2. 추상 클래스 구현 Class 생성

예제 : AnonymousAbstarctClass.java - 인스턴스 변수

```
// 인스턴스 변수
AnonymousAbstract workerMember = new AnonymousAbstract() {
  String getWorkerName() {
    return workerName;
  }
  @Override
  void doWork() {
    System.out.println("작업자 : " + workerName);
  }
  @Override
  void doWork(String str) {
      System.out.println("작업자 : " + str);
  }
};
```

예제 : https://github.com/hyomee/JAVA_EDU/tree/main/L/src/com/hyomee/anonymous

1. 익명(Anonymous) Class

➢ 2.추상 클래스 구현 Class 생성

예제 : AnonymousAbstarctClass.java - 인스턴스 Method

```
// 인스턴스 Method
void workerMethod(String workerNm) {
  AnonymousAbstract worker = new AnonymousAbstract() {
    @Override
    void doWork() {
      System.out.println("기본 작업자 : " + workerName + ", 작업자 : " + workerNm);
    }
    @Override
    void doWork(String str) {
      System.out.println("작업자 : " + str);
    }
  };
  worker.doWork();;
  worker.doWork("홍길동");
}
```

기본 작업자 : 기본작업자, 작업자 : 인스턴스의 지역변수
작업자 : 홍길동

➢ 2 추상 클래스 구현 Class 생성

예제 : AnonymousAbstarctClass.java - 인스턴스 메서드의 파라메터

```
// 인스턴스 메서드의 파라메터
void workerMethod(AnonymousAbstract worker) {
  worker.doWork();
  worker.doWork("익명객체 파라메터");
}
```

기본 작업자 없습니다.
작업자 : 익명객체 파라메터

➢ 3. 실행

예제 : AnonymousMain.java

```
public static void main(String... args) {

  AnonymousAbstarctClass anonymousClass = new AnonymousAbstarctClass();
  anonymousClass.workerMember.doWork();
  anonymousClass.workerMember.doWork("인스턴변수");
  // anonymousClass.workerMember.getWorkerName() 익명 함수 참조 불가
  anonymousClass.workerMethod("인스턴스의 지역변수");

  anonymousClass.workerMethod(new AnonymousAbstract() {
    @Override
    void doWork() {
      // private String workerName 접근 못함
      System.out.println("기본 작업자 없습니다.");
    }

    @Override
    void doWork(String str) {
      System.out.println("작업자 : " + str);
    }
  });

  AnonymousChild anonymousChild = new AnonymousChild();
  String str = anonymousChild.action.action("홍길동");
  System.out.println(str);

  anonymousChild.actionMethod("인스턴스메서드");
}
```

작업자 : 기본작업자
작업자 : 인스턴변수

파라메터로 객체 생성 하여 파라메터로 전달

예제 : AnonymousChild .java 참조

작업자 : 홍길동
작업자 : 인스턴스메서드

예제 : https://github.com/hyomee/JAVA_EDU/tree/main/L/src/com/hyomee/anonymous

1. 익명(Anonymous) Class

02. 인터페이스의 구현

➢ 1. 인터페이스 구현

예제 : AnonymousInterface.java

```java
package com.hyomee.lambda;

public interface AnonymousInterface {
  public String action(String str);
}
```

➢ 2. 인터페이스 구현 체

예제 : AnonymousInterfaceClass.java

```java
package com.hyomee.lambda;

public class AnonymousInterfaceClass implements AnonymousInterface {

  @Override
  public String action(String str) {
    return str;
  }
}
```

➢ 3. 실행

```java
public static void main(String... args) {
  AnonymousInterfaceClass ac01 = new AnonymousInterfaceClass();
  System.out.println("객체 지향으로 함수형 : " + ac01.action("기본"));

  AnonymousInterface ac02 = new AnonymousInterface() {
    @Override
    public String action(String str) {
      return str;
    }
  };

  System.out.println("익명 이너 클래스 : " + ac02.action("익명"));

  AnonymousInterface ac03 = (String str) -> { return  str; };
  System.out.println("람다 : " + ac02.action("람다"));
}
```

객체 지향으로 함수형 : 기본

익명 객체 생성

익명 이너 클래스 : 익명

람다 생성

람다 : 람다

예제 : https://github.com/hyomee/JAVA_EDU/tree/main/L/src/com/hyomee/anonymous

1. Lambda

" 자바에서 함수형 프로그램 방식으로 구현 해 주는 문법"

람다 (Lambda)

- 자바8이전에는 Method라는 함수 형태가 존재하지만 객체를 통해서만 접근이 가능하고, Method 그 자체를 변수로 사용하지는 못한다.
- 자바8에서는 함수를 변수처럼 사용할 수 있기 때문에, 파라미터로 다른 메소드의 인자로 전달할 수 있고, 리턴 값으로 함수를 받을 수도 있다.
- 이름없는 익명 함수 구현에서 주로 사용하며 함수형 인터페이스의 인스턴스(구현 객체)를 표현
  : **함수형 인터페이스 (추상 메소드가 하나인 인터페이스)를 구현 객체를 람다식으로 표현**

JAVA에서 함수형 프로그램

➢ 함수

- 함수 정의 -> 구현 -> 사용

```
void fun() {
   ...
}

fun()
```

익명 멤버 변수로 구현 방법 적용 →

➢ JAVA

- 함수는 없음
- Class, Interface 의 Method

```
class C {
  void method() {
     .....
  }
}

C c = new C();
c.method();
```

➢ JAVA Lambda

- 하나의 추상 메서드만 있는 interface 생성 -> 익명 Class (멤버 변수) 작성 -> Lambda로 변경

01. 인터페이스

```
Interface Example {
   R apply(A arg);
}
```

02. 인스턴스 생성

```
Example exp = new Example() {
   @Override
   public R apply(A arg) {
      body
   }
};
```

03. 인자 목록과 함수 몸통을 제외 하고 모두 제거

```
Example exp = (arg) {
    body
};
```

04. 문법 적용

```
Example exp = (arg) - >{
    body
} ;
```

1. Lambda

"자바에서 함수형 프로그램 방식으로 구현 해 주는 문법"

표현식

- (arg1, arg2...) -> { body }

- (params) -> expression
- (params) -> statement
- (params) -> { statements }

- (int a, int b) -> { return a + b; };
  > 타입 추론에 의한 타입 제거
  : (a, b) -> { return a+b } ;
  > 무엇인가를 반환 하거나 한 줄 표현식이 가능 하면 return 삭제
  : (a, b) -> a+b;

- () -> System.out.println("Hello ");
  > 파라미터없고 Hello 출력 System.out::println;

- () -> System.out.println("Hello "); // 파라미터 없고 Hello 출력
- (String s) -> { System.out.println(s); } // String s입력매개변수로 받아 출력
- () -> 8514790 //파라미터없고 8514790가 리턴
- () -> { return 3.14 }; //파라미터없고 3.14리턴

1. Lambda

예제 : LambdaFunctional.java

추상 메서드를 이용한 람다

	interface	익명 이너 클래스	함수	실행

리턴 : 없음
매개 : 있음
	interface IfNoRnNoArg {
	  void method();
	}
		IfNoRnNoArg iNrNa = new IfNoRnNoArg() {
		  void method () {
		  …
		  }
		};
			IfNoRnNoArg iNrNa = ( ) - > { … };	iNrNa.method();

리턴 : 없음
매개 : 있음
	interface IfNoRnArg {
	  void method(int a);
	}
		IfNoRnArg iNra = new IfNoRnArg() {
		  void method(int a) {
		  …
		  }
		};
			IfNoRnArg iNra = (int a) -> {
			  …
			}
				iNra.method(1);

리턴 : 없음
매개 : 있음
	interface IfRnNoArg {
	  int method();
	}
		IfRnNoArg irNa = new IfRnNoArg () {
		  int method() {
		  int a = 0
		  …
		  return a
		  }
		};
			IfRnNoArg irNa = () -> {
			  int a = 10;
			  return a;
			};
				irNa.method();

리턴 : 없음
매개 : 있음
	interface IfRnArg {
	  int method(int a, int b);
	}
		IfRnArg ira = new IfRnNoArg () {
		  int method(int a, int b) {
		  return a + b;
		  }
		};
			IfRnArg ira = (a, b) -> {
			  return a + b;
			};
				ira.method(5, 10)

배열	interface IfArray {
	  int [] method(int length)
	}
		IfRnArg ira = new IfArray () {
		  int method(int length) {
		  return new int[length];
		  }
		};
			IfRnArg ira = (length) -> {
			  return new int[length];
			};
			or
			IfRnArg ira = int[] :: new;
				int [] a1 = ira.
				ira.method(10);

1. Lambda

추상 메서드를 이용한 람다

```
interface IfNoRnNoArg {
  void method();
}

interface IfNoRnArg {
  void method(int a);
}

interface IfRnNoArg {
  int method();
}

interface IfRnArg {
  int method(int a, int b);
}
```

예제 : LambdaFunctional.java

```
public class LambdaFunctional {

  public void LamdaTest() {
    IfNoRnNoArg iNrNa = () -> { System.out.println("IfNoRnNoArg.method");};
    iNrNa.method();
```

```
IfNoRnNoArg.method
```

```
    IfNoRnArg iNra = (a) -> { System.out.println("IfNoRnArg.method a : " + a); };
    iNra.method(1);
```

```
IfNoRnArg.method a : 1
```

```
    IfRnNoArg irNa = () -> {
      int a = 10;
      return a;
    };
    System.out.println("IfRnNoArg.method : " + irNa.method());
```

```
IfRnNoArg.method : 10
```

```
    IfRnArg ira = (a, b) -> { return a + b; };
    System.out.println("IfRnArg.method : " + ira.method(5, 10));
  }
}
```

```
IfRnArg.method : 15
```

1. Lambda

존재 하는 객체 람다 표현

01. 객체 생성

람다 선언 방법 : 클래스객체 :: 인스턴스메서드명

예제 : ExistsClassLamda.java

```
interface IfExistsClass {
  Customer getCust(String name);
}
```

```
class Customer {
  private int custNo;
  private String custName;

  Customer(int custNo, String custName) {
    this.custNo = custNo;
    this.custName = custName;
  }

  int getCustNo() {
    return this.custNo;
  }

  String getCustName() {
    return this.custName;
  }
  void printCust() {
    System.out.println(this.custNo + "," + this.custName);
  }

  void changeName(String custName) {
    this.custName = custName;
  }
}
```

```
IfExistsClass iec = (name) -> {
  Customer customer = new Customer(20, "춘향") ;
  customer.printCust();
  customer.changeName(name);
  customer.printCust();
  return customer;
};

Customer customer = iec.getCust("이도령");
System.out.println(customer.getCustNo());
System.out.println(customer.getCustName());
```

람다로 객체 생성 방법

```
BiFunction<Integer, String, Customer> customerFn = Customer ::new;
Customer customer = customerFn.apply(20, "춘향");
```

```
BiFunction<Integer, String, Customer> customerFn = (custNo, custNm) -> new Customer(custNo, custNm);
Customer customer = customerFn.apply(20, "춘향");
```

1. Lambda

존재 하는 객체 람다 표현

02. 기존 객체의 메서드 실행

람다 선언 방법 : 클래스객체 :: 인스턴스메서드명

예제 : ExistsClassLamda.java

```
interface IfExistsClassMethod {
  String getCust();
}

interface IfExistsClassMethod01 {
  void changeName(String name);
}
```

클래스의 기존 함수 지정

```
int getCustNo() {
  return this.custNo;
}

String getCustName() {
  return this.custName;
}
void printCust() {
  System.out.println(this.custNo + "," + this.custName);
}

void changeName(String custName) {
  this.custName = custName;
}
```

클래스의 기존 함수 선택

클래스의 기존 함수 선택

```
interface IfExistsClassCreate {
  Customer create(int custNo, String CustNm);
}
```

```
IfExistsClassCreate iecmc = Customer ::new;;
Customer customer01 = iecmc.create(10, "온달");
```

```
Customer customer01 = new Customer(10, "온달");
IfExistsClassMethod iecm = customer01::getCustName;
System.out.println("iecm :: " + iecm.getCust());

IfExistsClassMethod01 iecm01 = customer01::changeName;
iecm01.changeName("평강");
System.out.println("iecm01 :: " + customer01.getCustNo());
System.out.println("iecm01 :: " + customer01.getCustName());
```

람다로 Class 생성

1. Lambda

존재 하는 객체 람다 표현

03. Static 메서드 실행

람다 선언 방법 : 클래스객체 :: 정적메서드명

예제 : ExistsClassLamda.java

```
static int staticMethod(int a) {
  return a;
}
```

```
interface IfExistsClassStaticMethod {
  int runStaticMethod(int ages);
}
```

```
IfExistsClassStaticMethod iecsm = Customer::staticMethod;
System.out.println("iecm01 :: " + iecsm.runStaticMethod(20));
```

04. 첫번째 매개변수로 전달된 매개변수를 사용 함

```
void changeName(String custName) {
  this.custName = custName;
}
```

```
interface IfExistsClassRefClass {
  void changeName(Customer customer, String custName);
}
```

```
IfExistsClassRefClass iecrc = Customer :: changeName;
iecrc.changeName(customer01, "탐라");
System.out.println("iecm01 :: " + customer01.getCustNo());
System.out.println("iecm01 :: " + customer01.getCustName());
```

객체를 메소드 내부로 보냈으므로 메서드 내부에서 생성 할 필요 없음

1. 함수형 인터페이스

함수형 인터페이스

- 추상Method가 하나뿐인 인터페이스 (Single Abstract Method : SAM)
- 여러 개의 Default Method가 있을 수 있다.
- @FunctionalInterface 어노테이션은 함수형 인터페이스이다
- Runnable, ActionListener, Comparable은 함수형 인터페이스
  : 자바 8 이전 : 익명 클래스 이용
  : 자바 8 이후 : 람다식 이용

java.util.function 에서 제공 하는 함수형 인터페이스

- Predicate: 하나의 매개변수를 주는 boolean형을 반환
- Consumer: 하나의 매개변수를 주는 void 형 accept 메소드
- Function: T 유형의 인수를 취하고 R 유형의 결과를 반환하는 추상 메소드 apply
- Supplier: 메소드 인자는 없고 T 유형의 결과를 반환하는 추상 메소드 get
- UnaryOperator: 하나의 인자와 리턴타입을 가진다. T -> T
- BinaryOperator: 두 개의 인수, 동일한 타입의 결과를 반환하는 추상 메서드 apply

01. 가장 기본이 되는 함수형 인터페이스

함수형 인터페이스	메서드
java.lang.Runnable	void run();
Supplier<T>	T get();
Consumer<T>	void accept(T t);
Function<T, R>	R apply (T t);
Predicate<T>	boolean test(T t);

02. 파라메터가 두개인 함수형 인터페이스

함수형 인터페이스	메서드
BiConsumer<T, U>	void accept(T t, U u);
BiPredicate<T, U>	boolean test(T t, U u);
BiFunction<T, U, R>	R apply(T t, U u);

03. 파라메터를 받고 동일한 타입을 리턴 하는 함수형 인터페이스

함수형 인터페이스	메서드
UnaryOperator<T>	T apply(T t);
BinaryOperator<T>	T apply(T t1, T t2);

04. 기본형 타입의 함수형 인퍼페이스

함수형 인터페이스	메서드
IntFunction<R>, LongFunction<R>, DoubleFunction<R>	R apply(int value), R apply(long value), R apply(double value)
ToIntFunction<T>, ToLongFunction<T>, ToDoubleFunction<T>	int applyAsInt(T t), long applyAsLong(T t), double applyAsDouble(T t)

1. 함수형 인터페이스


```
// Runnable은 인자를 받지 않고 리턴값도 없는 인터페이스
Runnable runnable = () -> System.out.println("실행하기!");
runnable.run();
```
실행하기!

```
// Supplier<T>는 인자를 받지 않고 T 타입의 객체를 리턴
Supplier<String> fnGetStr = () -> "문장을 리턴!";
String str = fnGetStr.get();
System.out.println(str);
```
문장을 리턴!

```
// Consumer<T>는 T 타입의 객체를 인자로 받고 리턴 값은 없습니다
Consumer<String> fnPrint00 = text -> System.out.println("나는 " + text + "?");
fnPrint00.accept("온달");

Consumer<String> fnPrint01 = text -> System.out.println("너는 " + text + "?");
Consumer<String> fnPrint02 = text -> System.out.println("왜 같이 살았을까?");
fnPrint01.andThen(fnPrint02).accept("평가");
```
나는 온달?
너는 평가?
왜 같이 살았을까?

```
// Function<T, R>는 T타입의 인자를 받고, R타입의 객체를 리턴합니다.
Function<Integer, Integer> fnMultiply = (value) -> value * 2;
Integer result = fnMultiply.apply(3);
System.out.println(result);
```
6

```
Function<Integer, Integer> fnAdd      = (value) -> value + 3;

// fnAdd 먼저 수행되고 그 이후에 fnMultiply 수행됨
Function<Integer, Integer> fnAddThenMultiply = fnMultiply.compose(fnAdd);
Integer rst = fnAddThenMultiply.apply(3);
System.out.println(rst);
```
12 (3 +3) * 2

```
// Predicate<T>는 T타입 인자를 받고 결과로 boolean을 리턴
Predicate<Integer> fnIsUp = num -> num > 5;
System.out.println("10은 5보다 크다 -> " + fnIsUp.test(10));
Predicate<Integer> fnisDown= num -> num < 6;
System.out.println(fnIsUp.and(fnisDown).test(10));
System.out.println(fnIsUp.or(fnisDown).test(10));
```
10은 5보다 크다 -> true
false (10 > 5 and 10 < 6)
true (10 > 5 or 10 < 6)

```
Predicate<String> isEquals = Predicate.isEqual("선화");
System.out.println(isEquals.test("선화"));
```
true

1. 함수형 인터페이스

05. Stream과 같이 사용하는 함수형 인터페이스

인터페이스	메서드	설명
Collection	boolean removeIf(Predicate<E> filter);	조건에 맞는 엘리먼트를 삭제
List	void replaceAll(UnaryOperator<E> operator);	모든 엘리먼트에 operator를 적용하여 대체(replace)
Iterable	void forEach(Consumer<T> action);	모든 엘리먼트에 action 수행

인터페이스	메서드	설명
Map	V compute(K key, BiFunction<K, V, V> f);	지정된 키에 해당하는 값에 f를 수행
Map	V computeIfAbsent(K key, Function<K, V> f);	지정된 키가 없으면 f 수행후 추가
Map	V cumputeIfPresent(K key, BiFunction<K, V, V> f)	지정된 키가 있을 때, f 수행
Map	V merge(K key, V value, BiFunction<V, V, V> f);	모든 엘리먼트에 Merge 작업 수행, 키에 해당하는 값이 있으면 f 수행해서 병합후 할당
Map	void forEach(BiConsumer<K, V> action);	모든 엘리먼트에 action 수행
Map	void replaceAll(BiFunction<K, V, V> f);	모든 엘리먼트에 f 수행후 대체

1. 타입 추론

타입 추론

- 자바 컴파일러는 람다 표현식이 사용된 컨텍스트(대상 형식)를 이용해서 람다 표현식과 관련된 함수형 인터페이스를 추론한다.
- 즉, 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 생략할 수 있다.
- 즉, 자바 컴파일러는 다음처럼 람다 파라미터 형식을 추론할 수 있다.
- 여러 파라미터를 포함하는 람다 표현식에서는 코드 가독성 향상이 더 두드러진다.

```
interface IfRnArg {
  int method(int a, int b);
}
```

```
IfRnArg ira = new IfRnNoArg () {
                    int method(int a, int b)
{
                        return a + b;
                    }
                };
```

```
IfRnArg ira = (int a, int b) -> {
      return a + b;
};
```

```
IfRnArg ira = (a, b) -> {
      return a + b;
};
```