

Java 8

Lambda Expression

1. Java8에서 람다란?

1. 실행 가능한 코드블럭

```
() -> {  
    System.out.println("Hello there!");  
}
```

2. 람다는 실행가능한 코드 블록으로

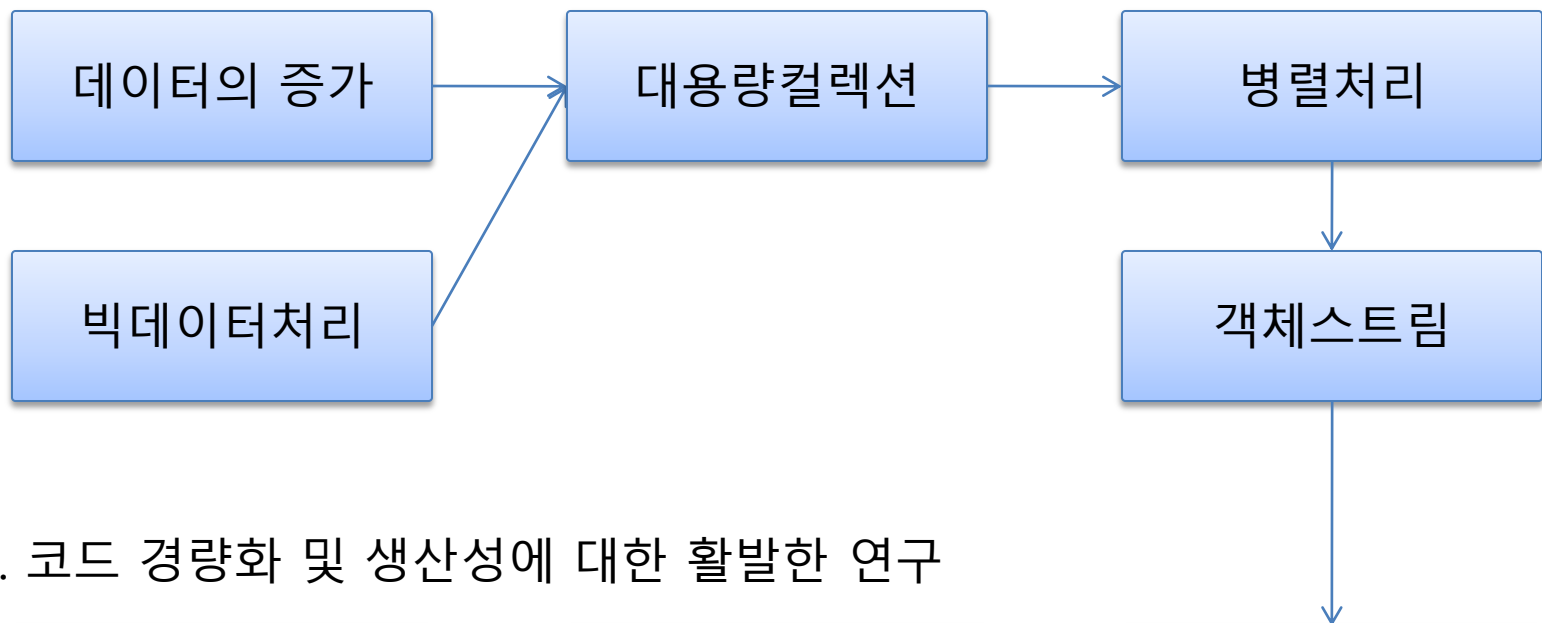
(Parameters) -> { code }

형태로 작성되며 변수에 담았다가 필요시에 호출해서 실행할 수 있다

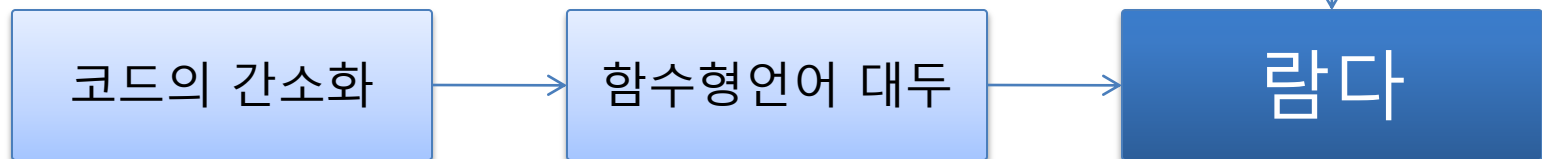
```
Runnable thread = () -> {  
    System.out.println("Hello there!");  
}  
thread.run();
```

2. 람다의 필요성

1. 대용량 데이터를 다루기 위한 병렬화 기술의 필요성



2. 코드 경량화 및 생산성에 대한 활발한 연구



3. Lambda Expression (람다표현식)

- 실행할 수 있도록 전달이 가능한 코드 블록

- 기본문법

`(Long val1, String val2) -> { val1 + val2.length(); }`

1. int 파라미터 a의 값을 콘솔에 출력하기 위한 람다표현식

`(int a) -> {System.out.println(a); }`

2) 파라미터 타입은 런타임 시에 대입되는 값에 따라 자동인식

`(a) -> {System.out.println(a); }`

3) 하나의 파라미터만 있다면 괄호 생략, 하나의 실행문만 있다면 중괄호 생략 가능

`a -> System.out.println(a);`

4) 파라미터가 없다면 빈 괄호를 반드시 사용

`() -> System.out.println(a);`

5) 중괄호를 실행하고 결과값을 리턴하는 경우

`(x, y) -> { return x + y; }`

6) 중괄호에 return문만 있을 경우

`(x, y) -> { x + y }`

4. Runnable 인터페이스로 보는 람다 01

1. 생성자와 method의 생략

```
Runnable thread = new Runnable() {  
    public void run() {  
        System.out.println("익명객체 코드에서 실행");  
    }  
};  
thread.run();
```

1. 객체내에 하나의 method만 존재할 경우 **method**를 생략할 수 있다

2. 객체내에 하나의 method만 존재할 경우 **객체자체**를 생략할 수 있다

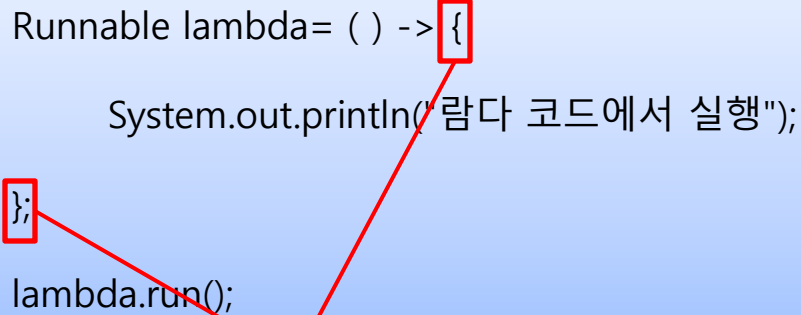


```
Runnable lambda = () -> {  
    System.out.println("람다 코드에서 실행");  
};  
lambda.run();
```

5. Runnable 인터페이스로 보는 람다 02

2. 코드 블록의 생략

```
Runnable lambda= ( ) -> {  
    System.out.println("람다 코드에서 실행");  
};  
lambda.run();
```



3. 코드블럭이 한줄일 경우 블럭을 생략할 수 있다



```
Runnable lambda= ( ) -> System.out.println("람다 코드에서 실행");  
lambda.run();
```

6. 함수형 인터페이스로 보는 람다 01

1. 함수형 인터페이스 @FunctionalInterface

```
@FunctionalInterface
interface LambdaFunction {
    public abstract int squareParameter(int number)
}
```

2. 함수형인터페이스를 통한 람다코드 작성

```
LambdaFunction rambda = (int num) -> { return num * num; };

int result = rambda.squareParameter(10);

System.out.println("result value : " + result);
```



result value : 100

7. 함수형 인터페이스로 보는 람다 02

3. 파라미터 타입이 추정가능 할때 파라미터 타입 생략

(int num) -> { **return** num * num; };



(num) -> { **return** num * num; };

4. 인자가 하나이고 자료형을 표기하지 않을 경우 괄호() 생략

(num) -> { **return** num * num; };



num -> { **return** num * num; };

5. 코드가 한줄인 경우 코드블럭 생략

함수형 인터페이스의 경우 코드블럭을 생략할경우 return 과 함께 생략해야만 한다

num -> { **return** num * num; };



num -> num * num;

8. 함수 파라미터로 보는 람다 01

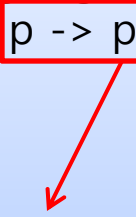
1. 함수형 인터페이스

```
@FunctionalInterface
interface LambdaFunction {
    public abstract int squareParameter(int number);
}
```

2. 파라미터로 사용하는 람다 코드 함수의 파라미터로 람다 코드를 받을 수 있다

```
public static void main(String args[]){
    LambdaFunction arg = p -> p * p;
    calc(arg);
}

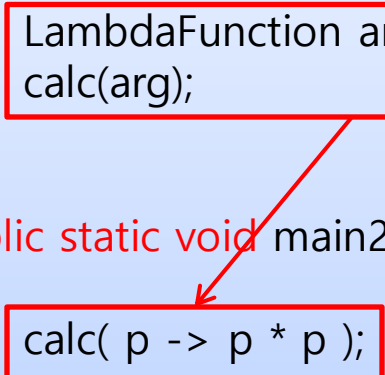
public void calc(LambdaFunction param){
    int result = param.squareParameter(7);
    System.out.println(result);
}
```



9. 함수 파라미터로 보는 람다 02

3. 파라미터로 람다코드를 받을 경우 람다코드를 직접 입력할 수 있다

```
public static void main1( ){  
    LambdaFunction arg = p -> p * p;  
    calc(arg);  
}  
  
public static void main2( ){  
    calc( p -> p * p );  
}  
  
public void calc(LambdaFunction param){  
    int result = param.squareParameter(7);  
    System.out.println(result);  
}
```

A red arrow points from the boxed lambda expression 'p -> p * p' in the 'main1' method to the boxed lambda expression 'p -> p * p' in the 'main2' method, illustrating how the lambda can be passed directly as a parameter.

10. 함수 return type 으로 보는 람다 01

1. 함수형 인터페이스

```
@FunctionalInterface
interface LambdaFunction {
    public abstract int squareParameter(int number);
}
```

2. 람다식을 리턴타입으로 갖는 함수를 사용할 수 있다 즉, 함수를 호출 시 람다 코드블럭 자체를 넘겨 받는다

```
public static void main(String args[]){
    LambdaFunction arg = calc( );
    int result = arg.squareParameter(6);
    System.out.println(result);
}

public static LambdaFunction calc( ){
    return num -> num * num;
}
```

11. Java 8 의 Functional Interface

1. 단항(입력값 1개) 인터페이스

Interface	Method	설명
Supplier<T>	T get()	입력값 없고, 반환값이 있다
Consumer<T>	accept(T t)	반환값이 없다. 자체사용
Function<T,R>	R apply(T t)	입력값 있고, 반환타입 지정
Predicate<T>	Boolean test(T t)	입력값에 대한 참거짓 판단
UnaryOperator<T>	T apply(T t)	입력과 반환타입이 동일하다

2. 이항(입력값 2개) 인터페이스

Interface	Method	설명
BiConsumer<T,U>	accept(T t,U u)	Consumer에 두개의 입력값
BiFunction<T,U,R>	R apply(T t, U u)	Function 에 두개의 입력값
BiPredicate<T,U>	Boolean test(T t,U u)	Predicate에 두개의 입력값
BinaryOperator<T,T>	T apply(T t,T t)	동일타입의 두개의 입력값

12. Java 8 의 Functional Interface 예제

1. 단항(입력값 1개) 인터페이스 예제

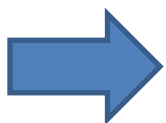
```
Supplier<Integer> sup = ()->180+20;  
System.out.println("Supplier:"+sup.get());
```

```
Consumer<Integer> con = a -> System.out.println("Consumer:"+a);  
con.accept(5);
```

```
Function<Integer,Double> fun = a -> a * 0.5;  
System.out.println("Function:"+fun.apply(5));
```

```
Predicate<Integer> pre = a -> a != 50;  
System.out.println("Predicate:"+pre.test(5));
```

```
UnaryOperator<Integer> una = a -> a * 50;  
System.out.println("UnaryOperator:"+una.apply(5));
```



```
Supplier:200  
Consumer:5  
Function:2.5  
Predicate:true  
UnaryOperator:250
```

13. Java 8 의 Functional Interface 예제

2. 이항(입력값 2개) 인터페이스 예제

단항 인터페이스에서 입력값만 두개로 바뀌는데

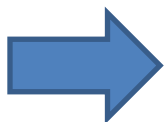
Binary의 경우 입력타입이 동일하므로 generic 은 한개만 선언해 준다

```
BiConsumer<Integer,Integer> con = (a,b) -> System.out.println("BiConsumer:"+(a*b));  
con.accept(5,7);
```

```
BiFunction<Integer,Integer,Double> fun = (a,b) -> a * b * 0.5;  
System.out.println("BiFunction:"+fun.apply(5,3));
```

```
BiPredicate<Integer,Double> pre = (a,b) -> a > b;  
System.out.println("BiPredicate:"+pre.test(5,5.1));
```

```
BinaryOperator<Integer> una = (a,b) -> a * b;  
System.out.println("BinaryOperator:"+una.apply(5,8));
```



BiConsumer:35
BiFunction:7.5
BiPredicate:false
BinaryOperator:40

14. Method Reference

- 코드블럭의 method가 입력값이 1개 일경우 method를 reference형태로 호출할 수 있다

1. 객체 Stream 에서의 일반 람다코드를 통한 반복문 처리

```
String objectArray[] = {"A","B","C","DX","E","F","G","H","I","J","K"};  
Arrays.stream(objectArray).forEach(a->System.out.println(a));
```



2. Method Reference 형태의 람다코드 처리

```
String objectArray[] = {"A","B","C","DX","E","F","G","H","I","J","K"};  
Arrays.stream(objectArray).forEach( System.out::println );
```

인자의 개수가 추측가능할 경우 **객체 :: 메서드** 형태로 호출할 수 있다

15. 람다의 예외처리

- 람다표현식에서 예외가 발생하면 호출자에게 전달된다. 일반적인 경우 해당 예외가 호출자에게 전달되는것이 바람직하지만 비동기 처리의 경우 아래와 같이 Handler를 사용해서 해당 예외를 처리하는 것이 권장된다.

```
async(a,b,(e)->{System.out.println("Here!! :"+e);}); // 호출
```

```
public static void async(Runnable a, Runnable b, Consumer<Throwable> handler) {  
    Thread t = new Thread( ){  
        public void run( ){  
            try{  
                a.run();  
                b.run();  
            }catch(Throwable e){  
                handler.accept(e);  
            }  
        }  
    };  
    t.start( );  
}
```