람다식(Lambda Expressions)

1. 람다식(Lambda Expressions) 이전 프로그램 구현 방식

람다식이 나오기 전과 후로 프로그램 구현 방식을 구분하자면 람다식이 소개되기 이전에 구현하던 방식을 '명령형 스타일', 람다식으로 구현하는 방식을 '함수형 스타일'이라고 한다. 명령형 스타일은 프로그램 구현 시 모든 작업 내용을 자바 코드로 자세하게 작성하는 방식이다. 그래서 구현해야 할 코드양이 많다. 그런데 함수형 스타일은 개발자가 핵심 내용만 구현하고 나머지는 자바 언어에서 자동으로 처리하는 방식이어서 코드가 간결하다.

(1) 명령형 스타일

다음은 명령형 스타일로 구현한 간단한 코드이다.

```
package exam_interface;
import java.util.Arrays;
import java.util.List;

public class Example01 {
	public static void main(String[] args) {
		// 명령형 스타일
		List<String> list = Arrays.asList("서울", "북경", "뉴욕", "방콕");
		boolean result = false;
		for(String city : list) {
			if(city.equals("서울")) {
				result = true;
				break;
			}
		}
		System.out.println(result);
	}
}
```

(2) 서술형 스타일

다음은 명령형 스타일의 코드를 서술형 스타일로 변경한 코드이다.

```
package exam_interface;

import java.util.Arrays;
import java.util.List;

public class Example01 {
	public static void main(String[] args) {
		//서술형 스타일
```

```
                List<String> list = Arrays.asList("서울", "북경", "뉴욕", "방콕");
                System.out.println(list.contains("서울"));
        }
}
```

서술형 스타일은 명령형 스타일처럼 변수와 로직을 구현해서 실행하는 것이 아니라, 처리 로직이 구현된 메서드를 호출해서 처리한다.

(3) 함수형 스타일

함수형 스타일은 서술형 스타일에 객체의 개념과 명령문을 추가하여 처리하는 방식이다. 이 방식을 지원하는 핵심 기능이 바로 람다식이다.

2. 인터페이스 구현 방법

구현하고자 하는 인터페이스는 다음과 같다.

```
interface MyInterface {
        public void print();
}
```

2.1. 방법1: implements 키워드로 클래스 선언

인터페이스를 구현하는 가장 일반적인 방법은 클래스를 선언할 때 implements 인터페이스명을 선언한 후 인터페이스에 선언된 추상 메서드를 오버라이딩하는 것이다.

```
class MyClass1 implements _____ {
        @Override
          _____ {
                System.out.println("MyClass1");
        }
}
```

인터페이스에 선언된 메서드를 실행할 때는 인터페이스를 구현한 클래스의 인스턴스를 생성한 후 참조 변수를 이용해 호출한다.

```
_____ mc1 = new _____ ;
mc1.print();
```

방법2: 익명 클래스 사용

인터페이스를 구현하는 방법 중 하나는 익명 클래스를 사용하는 것이다. 익명 클래스로 인터페이스를 구현할 때는 클래스를 별도로 선언할 필요없이 인터페이스 구현과 동시에 인스턴스를 생성할 수 있다.

```
MyInterface mi = _____
        @Override
        _____ {
                System.out.println("익명 클래스로 구현");
        }
```

```
};

mi.print(); // 익명 클래스로 구현
```

2.3. 방법3: 선언, 생성, 호출을 한번에 처리

인터페이스 구현과 동시에 객체가 생성되고 메서드를 호출이 실행되기 때문에 참조변수가 필요하지 않다.

```
(_____________________________
        @Override
        _____________________________{
                System.out.println("선언, 생성, 호출을 한번에 처리");
        }
___).______________; // 선언, 생성, 호출을 한번에 처리
```

2.4. 매개변수

인터페이스 타입으로 매개변수가 선언된 경우를 살펴보면 다음의 test() 메서드는 MyInterface 타입으로 매개변수가 선언되어 있다.

```
public static void test(MyInterface mi) {
        mi.print();
}
```

test() 메서드 호출 시 인터페이스를 구현한 클래스의 참조변수를 인자로 전달할 수 있다.

```
test(mc1); // MyClass1
test(mi); // 익명 클래스로 구현
```

2.5. 리턴타입

인터페이스 타입으로 반환값이 선언된 메서드를 살펴보면 test2()메서드는 MyInterface 타입을 반환한다.

```
public static MyInterface test2() {
        MyInterface mi = new MyInterface() {
                @Override
                public void print() {
                        System.out.println("test2()메서드에서 반환된 MyInterface");
                }
        };
        return mi;
}
```

다음은 인터페이스를 반환하는 메서드를 호출하는 메서드 예제 코드이다.

```
MyInterface mi2 = test2();
mi2.print(); // test2() 메서드에서 반환된 MyInterface
```

3. 람다식 사용하기

3.1. 람다식 기본

람다식을 사용해 인터페이스를 구현하고 생성한 후 사용하는 함수형 스타일로 코드를 작성하고자 한다.

(1) 일반 인터페이스 구현

```
public static MyInterface test3() {
        return new MyInterface() {
                @Override
                public void print() {
                        System.out.println("Hello");
                }
        };
}
```

test3() 메서드에서 반환한 익명 클래스를 사용하려면 다음처럼 test3() 메서드를 호출하고, 반환되는 값을 저장한다. 이때 반환하는 익명 클래스가 구현한 MyInterface 타입으로 참조변수를 선언한 후 참조변수를 이용해 print() 메서드를 호출한다.

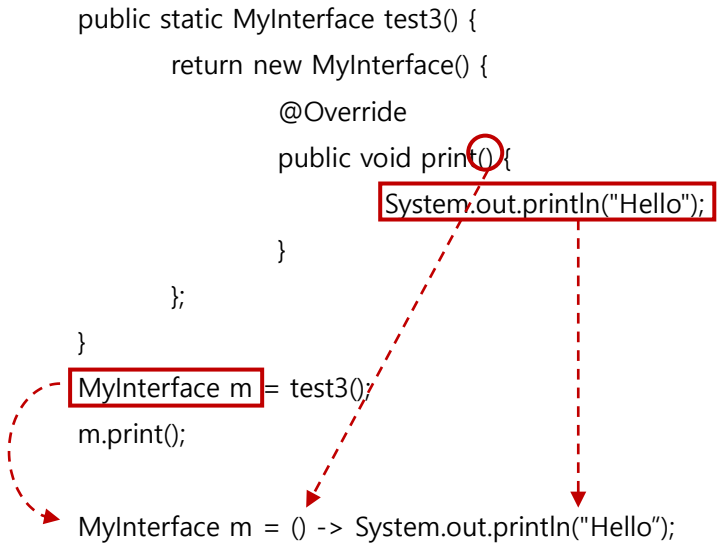
```
MyInterface m = test3();
m.print(); // test3() 메서드에서 반환된 MyInterface
```

(2) 람다식 구현

바로 앞에서 구현한 코드를 람다식으로 구현하면 다음과 같다.

```
MyInterface m = () -> System.out.println("Hello");
```

다음은 명령형 스타일과 함수형 스타일 코드를 비교한 것이다.



메서드의 매개변수들이 선언되는 괄호()는 그대로 표현하고, 인터페이스에 선언된 추상 메서드의 본문을 구현할 때는 화살표(->) 기호 다음에 작성한다. 본문을 구현할 때 명령문이 한 줄이면 위와 같이 작성하고, 여러 줄일 때는 중괄호 {}로 감싸준다. 이때 인터페이스에 선언된 추상 메서드가 여러 개일 때는 어

떤 메서드의 본문인지 구별할 방법이 없다. 따라서 람다식으로 구현하려는 인터페이스는 반드시 하나의 메서드만 선언되어야 한다. 그리고 **하나의 메서드만 선언된 인터페이스를 "함수형 인터페이스"라고 한다.** 정리하면 람다식으로 구현할 수 있는 인터페이스는 함수형 인터페이스만 가능하다.

(3) 람다식 문법

람다식의 문법은 다음과 같다.

```
() -> 명령문; ← 함수형 인터페이스의 추상 메서드 구현 시 명령문이 한 개일 때
() -> {       ← 함수형 인터페이스의 추상 메서드 구현 시 명령문이 여러 개일 때
    명령문1;
    명령문n;
};
```

LambdaExample01.java

```
package exam_interface;

interface Multiply {
	double getValue();
}

public class LambdaExample01 {
	public static void main(String[] args) {
		Multiply m;
		m = new Multiply() {
			@Override
			public double getValue() {
				return 3 + 6;
			}
		};
		System.out.println(m.getValue());

		m = () -> 5.0 * 2;
		System.out.println(m.getValue());

		m = () -> 10 / 3;
		System.out.println(m.getValue());
	}
}
```

(4) @FunctionalInterface 어노테이션

람다식으로 구현할 수 있는 인터페이스는 하나의 추상 메서드를 포함하는 함수형 인터페이스만 가능하다. 따라서 함수형 인터페이스를 작성할 때 두 개 이상의 추상 메서드가 선언되는 오류를 예방하고자

@FunctionalInterface 어노테이션을 사용할 수 있다.

MyInterface.java

```
1 package exam_interface;
2
3 @FunctionalInterface
4 interface MyInterface {
5     public void put();
6     public void get();
7 }
```

@FunctionalInterface 어노테이션을 지정하면 하나 이상의 추상 메서드가 선언될 때 위와 같이 컴파일 오류가 발생하도록 명시해 준다.

(5) 람다식 매개변수

【 매개변수가 있는 람다식 구현 】
(변수명) -> 명령문: ← 매개변수가 한 개일 때
(변수명1, ...변수명n) -> 명령문; ← 매개변수가 여러 개일 때

지정해 줄 때 변수명 앞에 타입을 지정하지 않는다는 것이다. 이처럼 변수의 타입을 지정하지 않으면 자바 언어에서는 자동으로 인터페이스에 선언된 추상 메서드의 매개변수 타입으로 처리해주므로 생략해도 상관없다. 이처럼 함수형 스타일은 개발자가 핵심적인 내용만 구현하고 부가적인 처리는 자바 언어에 맡기는 특징이 있다.

물론 괄호 ()안에 타입을 지정하고 싶다면 다음처럼 할 수 있다.

(타입 변수명) -> 명령문;

- 매개변수를 한 개 사용하는 람다식

LambdaExample02.java

```
package exam_lambda;

interface Verify {
	boolean check(int n);
}

public class LambdaExample02 {
	public static void main(String[] args) {
		Verify isEven = new Verify() {
			@Override
			public boolean check(int n) {
				return (n % 2) == 0;
			}
		};
		System.out.println("결과값: "+isEven.check(3));
```

```
                isEven = (n) -> (n % 2) == 0;
                System.out.println("결과값: "+isEven.check(10));

                Verify isOdd = (n) -> (n % 2) == 1;
                System.out.println("결과값: "+isOdd.check(9));
        }
}
```

- 매개변수를 여러 개 사용하는 람다식

LambdaExample03.java

```
package exam_lambda;

interface Verify2 {
        boolean check(int n, int d);
}

public class LambdaExample03 {
        public static void main(String[] args) {
                Verify2 vf = (n, d) -> (n % d) == 0;
                System.out.println("결과값: " + vf.check(24, 3));
        }
}
```

(6) 람다식 블록

람다식 문법에서 () -> 기호 다음 부분은 람다식 본문이다. 람다식 본문의 명령문이 한 줄이면 다음처럼 표현하며 이때는 return 문을 생략해도 명령문에서 처리된 값이 자동으로 반환된다. 람다식 본문에 명령문이 여러 줄로 구현하려면 다음처럼 블록{}으로 감싸야 한다.

[람다식 본문이 여러 줄일 때]

```
() -> {
    명령문1;
    명령문n;
    return 값;
}
```

람다식에서 return문은 명령문이 한 줄 일 때는 생략해도 되지만, 블록 {}을 사용하면 생략할 수 없다.

LambdaExample04.java

```
package exam_lambda;

interface NumberFunction {
        int sum(int n);
```

```
}

public class LambdaExample04 {
	public static void main(String[] args) {
		NumberFunction number = (n) -> {
			int result = 0;
			for(int i=1; i<=n; i++) {
				result += i;
			}
			return result;
		};

		System.out.println("1부터 10까지의 합: " + number.sum(10));
		System.out.println("1부터 100까지의 합: " + number.sum(100));
	}
}
```

(7) 제네릭 함수형 인터페이스

```
interface StringFunction {
	String modify(String str);
}
```

```
interface IntegerFunction {
	Integer modify(Integer in);
}
```

그래서 두 개의 인터페이스를 한번에 처리할 수 있는 제네릭 함수형 인터페이스(MyFunction)를 정의하면 된다.

```
interface _____ {

	_____
}
```

[제네릭 인터페이스에 타입 인자 전달]

인터페이스<타입인자> 변수명 = () -> 명령문

LambdaExample05.java

```
package exam_lambda;

interface MyFunction<T> {
	T modify(T data);
```

```
}

public class LambdaExample05 {
	public static void main(String[] args) {
		MyFunction<String> mf1 = (str) -> str.toUpperCase() + ":" + str.length();
		System.out.println("결과값: " + mf1.modify("java"));
		System.out.println("결과값: " + mf1.modify("java programming"));

		MyFunction<Integer> mf2 = (n) -> n * 2;
		System.out.println("결과값: " + mf2.modify(22));
		System.out.println("결과값: " + mf2.modify(43));
	}
}
```

3.2 람다식 활용

(1) 람다식 인자

호출하는 메서드의 매개변수 타입이 인터페이스로 선언되었다면 인터페이스를 구현한 객체를 인자로 전달하면 된다. 람다식은 인터페이스 구현과 동시에 객체를 생성하는 방식이므로 람다식도 인자로 전달할 수 있다. 아래의 메서드가 존재한다 가정하여 보자.

```
static String test(StringFunction sf, String str)
```

```
StringFunction sf = (str) -> {
		String result = "";
		char c;
		for(int i=0; i<str.length(); i++) {
			c = str.charAt(i);
			if(c==',') {
				result += " ";
			} else {
				result += c;
			}
		}
		return result;
	};

String s1 = test(sf, str);
```

위 내용을 test(람다식, 문자열) 형식으로 test()를 호출한다.

```
String s2 = test((str) -> {
        int max = 0;
        int index = 0;
        String[] word = str.split(",");
        for(int i=0; i < word.length; i++) {
                if(max < word[i].length()) {
                        max = word[i].length();
                        index = i;
                }
        }
        return word[index];
}, str);
```

LambdaExample06.java

```
package exam_lambda;

interface StringFunction {
        String modify(String str);
}

public class LambdaExample06 {
        static String test(StringFunction sf, String str) {
                return sf.modify(str);
        }

        public static void main(String[] args) {
                String cityName = "Korea,Australia,China,Germany,Spain,Turkey";
                StringFunction sf1 = (s) -> {
                        String result = "";
                        char c;
                        for (int i = 0; i < s.length(); i++) {
                                c = s.charAt(i);
                                if (c == ',')
                                        result += " ";
                                else
                                        result += c;
                        }
                        return result;
                };
```

```
                String s1 = test(sf1, cityName);
                System.out.println("결과값: " + s1);

                String s2 = test((s) -> {
                        int max = 0;
                        int index = 0;
                        String[] word = s.split(",");
                        for (int i = 0; i < word.length; i++) {
                                if (max < word[i].length()) {
                                        max = word[i].length();
                                        index = i;
                                }
                        }
                        return word[index];
                }, cityName);
                System.out.println("결과값: " + s2);
        }
}
```

(2) 람다식 예외 처리

람다식을 구현할 때 예외가 발생하게 할 수 있다. 람다식 본문에서 예외가 발생하게 하려면 반드시 함수형 인터페이스의 추상메서드 선언에서 throws 키워드를 지정해주어야 한다.

LambdaExample07.java

```
package exam_lambda;

import java.util.Arrays;

class EmptyStringException extends Exception {
        private static final long serialVersionUID = 1L;

        public EmptyStringException() {
                super("빈 문자열");
        }
}

interface StringFunction2 {
        String[] modify(String str) throws EmptyStringException;
}

public class LambdaExample07 {
```

```
	public static void main(String[] args) {
		String cityName = "Korea,Australia,China,Germany,Spain,Turkey";

		try {
			StringFunction2 sf = (str) -> {
				if (str.length() == 0)
					throw new EmptyStringException();
				return str.split(",");
			};

			String result[] = sf.modify(cityName);
			System.out.println("결과값: " + Arrays.toString(result));

			String str2 = "";
			String result2[] = sf.modify(str2);
			System.out.println("결과값: " + Arrays.toString(result2));
		} catch (EmptyStringException e) {
			System.err.println(e.getMessage());
		}
	}
}
```

(3) 변수사용

자바에서 사용하는 변수는 크게 필드와 지역변수가 있다. 필드는 클래스의 멤버로 선언된 변수로서 인스턴스가 생성될 때마다 힙 영역에 생성된다. 지역변수는 메서드 안이나 특정한 블록에 선언된 변수로서 메서드나 블록이 실행될 때마다 스택 영역에 생성된다.

람다식의 본문을 구현할 때 필드와 람다식이 구현된 메서드의 지역변수를 사용할 수 있다. 그런데 필드는 값을 사용하거나 수정할 수 있지만, 지역변수는 사용만 할 수 있고 수정은 할 수 없다. 람다식이 구현된 메서드의 지역변수는 마치 final이 선언된 변수처럼 사용되는 것이다.

LambdaExample08.java

```
package exam_lambda;

interface Calculator {
	int func();
}

public class LambdaExample08 {
	static int num1 = 10;
	int num2 = 20;

	public static void main(String[] args) {
```

```
        LambdaExample08 test = new LambdaExample08();
        //int num3 = 30;

        Calculator calc = () -> {
            int num4 = 40;
            LambdaExample08.num1++;
            test.num2++;
            //num3++; // 오류 발생
            num4++;
            return num4;
        };
        System.out.println("실행결과: " + calc.func());
    }
}
```

3.3. 메서드 참조

람다식 본문에 복잡한 로직을 구현할 때는 가독성이 떨어질 수 있다. 그래서 람다식을 메서드 형태로 구현하는 기능을 지원한다. LambdaExample06 클래스에서 람다식 본문을 메서드로 구현한 코드이다.

```
    static String func(String str) {
        String result = "";
        char c;
        for (int i = 0; i < str.length(); i++) {
            c = str.charAt(i);
            if (c == ','){
                result += " ";
            } else{
                result += c;
            }
        }
        return result;
    }
```

람다식을 구현하는 메서드의 리턴타입과 매개변수는 함수형 인터페이스의 추상메서드와 동일해야 한다. 그러나 메서드 이름은 상관없다. 왜냐하면 인터페이스의 추상 메서드를 구현하고자 선언된 메서드이기 때문이다. 이처럼 인터페이스의 추상 메서드를 구현하고자 선언된 메서드를 사용하여 인터페이스 객체를 생성하는 문법은 다음과 같다.

【 인터페이스 객체 생성 】
클래스명::메서드명 ← static 메서드로 선언했을 때
참조변수명::메서드명 ← 인스턴스 메서드로 선언했을 때

인터페이스 구현 시 메서드를 참조하기 때문에 이러한 구현 방식을 "메서드 참조(Method Reference)"라고 한다.

【 함수형 인터페이스를 구현한 후 객체를 생성하는 방법은 다음이 세가지이다. 】

1. 명령형 스타일
2. 람다식 ← () -> { }
3. 메서드 참조 ← 클래스명::메서드명

(1) static 메서드 참조

MethodReference01.java

```
package exam_methodRef;

interface StringFunction {
	String modify(String str);
}

public class MethodReference01 {
	static String func(String str) {
		String result = "";
		char c;
		for (int i = 0; i < str.length(); i++) {
			c = str.charAt(i);
			if (c == ',') {
				result += " ";
			} else {
				result += c;
			}
		}
		return result;
	}

	public static void main(String[] args) {
		StringFunction sf = MethodReference01::func;

		String cityName = "Korea,Australia,China,Germany,Spain,Turkey";
		String result = sf.modify(cityName);
		System.out.println("실행결과: " + result);

		String str2 = "서울,북경,도쿄,뉴욕,발리";
		result = sf.modify(str2);
		System.out.println("실행결과: " + result);
	}
}
```

(2) 인스턴스 메서드 참조

MethodReference01.java 소스파일 수정

```
package exam_methodRef;

interface StringFunction {
	String modify(String str);
}

public class MethodReference01 {
	String func(String str) { // static 제어자 삭제
		String result = "";
		char c;
		for (int i = 0; i < str.length(); i++) {
			c = str.charAt(i);
			if (c == ',') {
				result += " ";
			} else {
				result += c;
			}
		}
		return result;
	}

	public static void main(String[] args) {
		MethodReference01 mf = new MethodReference01();
		StringFunction sf = mf::func;

		String cityName = "Korea,Australia,China,Germany,Spain,Turkey";
		String result = sf.modify(cityName);
		System.out.println("실행결과: " + result);

		String str2 = "서울,북경,도쿄,뉴욕,발리";
		result = sf.modify(str2);
		System.out.println("실행결과: " + result);
	}
}
```

(3) 제네릭 메서드 참조

함수형 인터페이스 구현시 제네릭 메서드를 참조할 때 다음처럼 타입 매개변수를 메서드명 앞에 지정한다

【 제네릭 메서드 참조 】

클래스명::<타입>메서드명

MethodReference02.java

```
package exam_methodRef;

interface MyFunction<T> {
	int func(T[] array, T value);
}

class MyUtil {
	static <T> int count(T[] array, T value) {
		int cnt = 0;
		for (int i = 0; i < array.length; i++) {
			if (array[i] == value)
				cnt++;
		}
		return cnt;
	}
}

public class MethodReference02 {
	static <T> int test(MyFunction<T> mf, T[] array, T value) {
		return mf.func(array, value);
	}

	public static void main(String[] args) {
		String[] list1 = { "blue", "red", "yellow", "blue", "red", "blue" };
		Integer[] list2 = { 10, 50, 10, 50, 40, 10, 90, 10, 20 };

		int cnt = 0;
		cnt = test(MyUtil::<String>count, list1, "blue");
		System.out.println("찾고자 하는 문자의 개수 : " + cnt);

		cnt = test(MyUtil::<Integer>count, list2, 10);
		System.out.println("찾고자 하는 숫자의 개수 : " + cnt);
	}
}
```

3.4 함수형 인터페이스 API

기본적인 함수형 인터페이스는 java.util.function 패키지에서 제공한다. 사용하려는 함수형 인터페이스의 매개변수와 리턴타입이 같다면 새로 선언하지 말고 API를 사용하는 것이 효율적이다.

종류	추상 메소드의 특징
Function	매개값이 있고 반환값이 있다. 주로 매개값을 반환값으로 매핑(타입 변환)
Predicate	매개값이 있고 반환 타입은 boolean, 매개값을 조사해서 true/false를 반환
Consumer	매개값이 있고 반환값이 없다.
Supplier	매개값이 없고 반환값이 있다.

java.util.function 패키지에서 제공하는 기본적인 함수형 인터페이스는 다음과 같다.

인터페이스명	추상 메소드	설 명
Function<T, R>	R apply(T t)	T 타입 인자 처리 후 R 타입 값 반환
Predicate<T>	boolean test(T t)	T 타입 인자 처리 후 boolean 값 반환
Consumer<T>	void accept(T t)	T 타입 인자 처리(반환값 없음)
Supplier<T>	T get()	인자 없이 처리 후 T 타입 값 반환.

(1) Function

인터페이스: Function<T, R> 메서드: R apply(T t)

Function 함수형 인터페이스에 선언된 메서드는 apply()이다. apply() 메서드는 매개변수 T와 리턴타입 R로 선언되어있다.

FunctionExample.java

```
package exam_lambda;

import java.util.function.Function;

public class FunctionExample {
	public static void main(String[] args) {
		Function<String, Integer> func = (s) -> {
			int cnt = 0;
			String[] word = s.split(" ");
			cnt = word.length;
			return cnt;
		};

		int wordCnt = func.apply("고개를 들어 별들을 보라 당신 발만 내다 보지말고...");
		System.out.println("단어 수 : " + wordCnt);
	}
}
```

매개변수 타입과 반환 타입에 따라서 다음과 같은 Function 함수적 인터페이스들이 있다.

인터페이스명	추상 메소드	설 명
Function<T, R>	R apply(T t)	객체 T를 매개변수로 객체 R를 반환
BiFunction<T, U, R>	R apply(T t, U u)	객체 T와U를 매개변수로 객체 R로 반환
DoubleFunction<R>	R apply(double value)	double을 객체 R로 반환
IntFunction<R>	R apply(int value)	int를 객체 R로 반환
IntToDoubleFunction	double applyAsDouble(int value)	int를 double로 반환
IntToLongFunction	long applyAsLong(int value)	int를 long으로 반환
LongToDoubleFunction	double applyAsDouble(long value)	long를 double로 반환
LongToIntFunction	int applyAsInt(long value)	long를 int로 반환
ToDoubleBiFunction<T, U>	double applyAsDouble(T t, U u)	객체 T와U를 double로 반환
ToDoubleFunction<T>	double applyAsDouble(T value)	객체 T를 double로 반환
ToIntBiFunction<T, U>	int applyAsInt(T t, U u)	객체 T와U를 int로 반환
ToIntFunction<T>	int applyAsInt(T value)	객체 T를 int로 반환
ToLongBiFunction<T, U>	long applyAsLong(T t, U u)	객체 T와U를 long으로 반환
ToLongFunction<T>	long applyAsLong(T value)	객체 T를 long으로 반환

(2) Predicate

인터페이스: Predicate<T>
메서드: boolean test(T t)

Predicate 함수형 인터페이스에 선언된 메서드 test()이다. test() 메서드는 매개변수 T와 리턴 타입 boolean으로 선언되었다.

PredicateExample.java

```
package exam_lambda;

import java.util.function.Predicate;

public class PredicateExample {
	public static void main(String[] args) {
		Predicate<Integer> func = (n) -> n % 2 == 0;
		if (func.test(123))
			System.out.println("짝수입니다.");
		else
			System.out.println("홀수입니다.");
	}
}
```

매개변수의 타입과 수에 따라서 다음과 같은 Predicate 함수적 인터페이스들이 있다.

인터페이스명	추상 메소드	설 명
Predicate<T>	boolean test(T t)	객체 T를 매개변수로 받아 boolean으로 반환
BiPredicate<T, U>	boolean test(T t, U u)	객체 T와 U를 매개변수로 받아 boolean으로 반환
DoublePredicate	boolean test(double value)	double 값을 매개변수로 받아 boolean으로 반환
IntPredicate	boolean test(int value)	int 값을 매개변수로 받아 boolean으로 반환
LongPredicate	boolean test(long value)	long 값을 매개변수로 받아 boolean으로 반환

(3) Consumer

인터페이스: Consumer<T> 메서드: void accept(T t)

Consumer 함수형 인터페이스에 선언된 메서드는 accept()이다. accept() 메서드는 매개변수 T와 리턴타입은 void로 선언되었다.

ConsumerExample.java

```
package exam_lambda;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.function.Consumer;

public class ConsumerExample {
	public static void main(String[] args) {
		Consumer<Date> date = (d) -> {
			String s = new SimpleDateFormat("YYYY-MM-dd").format(d);
			System.out.println(s);
		};
		date.accept(new Date());
	}
}
```

매개변수의 타입과 수에 따라서 다음과 같은 Consumer 함수적 인터페이스들이 있다.

인터페이스명	추상 메소드	설 명
Consumer<T>	void accept(T t)	객체 T를 매개변수로 반환값이 존재하지 않음.
BiConsumer<T, U>	void accept(T t, U u)	객체 T와 U를 매개변수로 반환값이 존재하지 않음.
DoubleConsumer	void accept(double value)	double 값을 매개변수로 반환값이 존재하지 않음.
IntConsumer	void accept(int value)	int 값을 매개변수로 반환값이 존재하지

		않음.
LongConsumer	void accept(long value)	long 값을 매개변수로 반환값이 존재하지 않음.
ObjDoubleConsumer<T>	void accept(T t, double value)	객체 T와 double값을 매개변수로 반환값이 존재하지 않음.
ObjIntConsumer<T>	void accept(T t, int value)	객체 T와 int 값을 매개변수로 반환값이 존재하지 않음.
ObjLongConsumer<T>	void accept(T t, long value)	객체 T와 long 값을 매개변수로 반환값이 존재하지 않음.

(4) Supplier

인터페이스: Supplier<T> 메서드: T get()

Supplier 함수형 인터페이스에 선언된 메서드는 get()이다. get() 메서드는 전달받는 인자가 없고 리턴타입은 T로 선언되었다.

SupplierExample.java

```
package exam_lambda;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.function.Supplier;

public class SupplierExample {
	public static void main(String[] args) {
		Supplier<String> day = () -> new SimpleDateFormat("E요일").format(new Date());
		String result = day.get();
		System.out.println(result);
	}
}
```

매개변수의 타입과 수에 따라서 다음과 같은 Suppier 함수적 인터페이스들이 있다.

인터페이스명	추상 메소드	설 명
Supplier	T get()	T 객체를 반환
BooleanSupplier	boolean getAsBoolean()	Boolean 값을 반환
DoubleSupplier	double getAsDouble()	double 값을 반환
IntSupplier	int getAsInt()	int 값을 반환
LongSupplier	long getAsLong()	long 값을 반환