람다 표현식

람다 표현식

- 기능 하나가 필요한 상황을 위한 람다
  - 자바는 객체지향 언어이므로 코드 흐름의 대부분에 클래스와 인스턴스 존재
  - 코드 작성 시 기능 하나를 정의해서 전달해야 하는 상황을 자주 접하게 됨

```
public class SLenComp implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

```
public class SLenComparator {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Robot");
        list.add("Lambda");
        list.add("Box");
        Collections.sort(list, new SLenComp());
        for (String s : list)
            System.out.println(s);
    }
}
```

  - 위 코드에서 Collection.sort 메소드를 호출하면서 두 번째 인자로 정렬의 기준을 갖고 있는 인스턴스를 생성해서 전달
  - 인스턴스를 전달하는 형태이지만 내용을 보면, 메소드, 즉 기능을 전달하는 것에 해당

람다 표현식

- 매개변수가 있고 반환하지 않는 람다식

```
public interface Printable {
    void print(String s);
}
```

```
public class OneParamNoReturn {
    public static void main(String[] args) {
        // 줄임없는 표현
        Printable p1 = (String s) -> { System.out.println(s); };
        p1.print("Lambda exp one.");
        // 중괄호, 매개변수 형, 매개변수 소괄호 생략
        Printable p2 = s -> System.out.println(s);
        p2.print("Lambda exp one.");
    }
}
```

  - 메소드 몸체가 두 줄 이상의 문장으로 이루어져 있거나 매개변수의 수가 둘 이상인 경우에는 각각 중괄호 및 소괄호 생략 불가

```
public interface Calculate {
    void cal(int a, int b);
}
```

```
public class TwoParamNoReturn {
    public static void main(String[] args) {
        Calculate c1 = (a, b) -> System.out.println(a + b);
        c1.cal(4, 3);
        Calculate c2 = (a, b) -> System.out.println(a - b);
        c2.cal(4, 3);
        Calculate c3 = (a, b) -> System.out.println(a * b);
        c3.cal(4, 3);
    }
}
```

람다 표현식

- 매개변수가 있고 반환하는 람다식

```
public interface Printable {
    int print(String s);
}
```

```
public class TwoParamAndReturn {
    public static void main(String[] args) {
        Calculate c1 = (a, b) -> { return a + b; };
        System.out.println(c1.cal(4, 3));
        Calculate c2 = (a, b) -> a + b; ;
        System.out.println(c1.cal(4, 3));
    }
}
```

  - 메소드 정의가 한 줄이지만 return이 있으면 중괄호의 생략이 불가능하며, 이 경우 return을 생략할 수 있음
  - 즉, 메소드의 정의가 return문이 유일하면 return을 생략하는 것이 보편적인 방식

```
public interface HowLong {
    int len(String s);
}
```

```
public class OneParamAndReturn {
    public static void main(String[] args) {
        HowLong hl = s -> s.length();
        System.out.println(hl.len("I amd so happy."));
    }
}
```

람다 표현식

- 매개변수가 없는 람다식

```
public interface Generator {
    int rand();
}
```

```
public class NoParamAndReturn {
    public static void main(String[] args) {
        Generator gen = () -> {
            Random rand = new Random();
            return rand.nextInt(50);
        };
        System.out.println(gen.rand());
    }
}
```

- 매개변수 선언이 없는 관계로 매개변수 정보를 담는 소괄호는 빈칸
- 두 줄 이상의 문장으로 이루어진 람다식은 중괄호로 반드시 감싸야 하며, 값을 반환할 때에도 return문을 반드시 사용

람다 표현식

- 함수형 인터페이스(Functional Interfaces)와 어노테이션
  - 이전에 봤던 람다식 관련 코드에는 인터페이스에 추상 메소드가 딱 하나만 존재
  - 이러한 인터페이스를 가리켜 "함수형 인터페이스"라 하며, 람다식은 이러한 함수형 인터페이스를 기반으로만 작성할 수 있음
  - 함수형 인터페이스에 "@FunctionalInterface"라는 어노테이션을 붙여 사용
  - "@FunctionalInterface" 어노테이션은 함수형 인터페이스에 부합하는지 확인하기 위한 어노테이션 타입으로 인터페이스에 둘 이상의 추상 메소드가 존재하면, 이는 함수형 인터페이스가 아니기 때문에 컴파일 오류 발생
  - 단, static, default 선언이 붙은 메소드의 정의는 함수형 인터페이스의 정의에 영향을 미치지 않음

```
@FunctionalInterface
public interface Calculate {
    int cal(int a, int b);
    default int add(int a, int b) {
        return a + b;
    }
    static int sub(int a, int b) {
        return a - b;
    }
}
```

람다 표현식

- 람다식과 제네릭
  - 인터페이스는 제네릭으로 정의하는 것이 가능

```
@FunctionalInterface
public interface Calculate <T> {
    T cal(T a, T b);
}
```

```
public class LambdaGeneric {
    public static void main(String[] args) {
        Calculate<Integer> ci = (a, b) -> a + b;
        System.out.println(ci.cal(4, 3));
        Calculate<Double> cd = (a, b) -> a + b;
        System.out.println(cd.cal(4.32, 3.45));
    }
}
```

람다 표현식

1. 아래 코드에서 주석에 명시된 연산의 결과를 출력하기 위한 calAndShow 메소드의 호출문을 람다식을 기반으로 작성해보자.

```
@FunctionalInterface
public interface Calculate <T> {
    T cal(T a, T b);
}
```

```
public class CalculatorDemo {
    public static <T> void calAndShow(Calculate<T> op, T n1,
T n2) {
        T r = op.cal(n1, n2);
        System.out.println(r);
    }

    public static void main(String[] args) {
        // 3 + 4
        // 2.5 + 7.1
        // 4 - 2
        // 4.9 - 3.2
    }
}
```