yonggyo1125 Update README.md a8e3229 · last year History

Name	Name	Last commit date
..		
source/day18	스트림 강의 작성	2 years ago
README.md	Update README.md	last year

README.md

강의 동영상 링크

동영상 링크

람다식(Lamba expression)

람다식이란?

- 메서드를 하나의 '식(expression)'으로 표현한 것
- 람다식은 메서드를 간략하면서도 명확한 식으로 표현할 수 있게 해준다.
- 메서드를 람다식으로 표현하면 메서드의 이름과 반환값이 없어지므로, 람다식을 '익명 함수(anonymous function)' 이라고도 한다.

람다식 문법 살펴보기

매개변수 자료형과 괄호 생략하기

- 매개변수 자료형 생략 가능, 단 매개변수 타입이 있으면 괄호를 생략할 수 없다.
- 매개변수가 하나인 경우에는 괄호도 생략할 수 있음
- 매개변수가 2개 이상인 경우 괄호를 생략할 수 없다.

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

(x, y) -> { System.out.println(x + y) }
```

중괄호 생략하기

- 중괄호 안의 구현 부분이 한 문장인 경우 중괄호를 생략할 수 있다.

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

- 중괄호 안의 구현부분이 한 문장이더라도 return 문은 중괄호를 생략할 수 없다.

```
str -> return str.length(); // 잘못된 형식(오류 발생)
str -> str.length(); // 올바른 형식
```

return 생략하기

- 중괄호 안의 구현 부분이 return문 하나라면 중괄호와 return을 모두 생략하고 식만 쓴다.

```
(x,y) -> x+y
str -> str.length()
```

람다식 사용하기

람다식 구현 방법

1. 인터페이스를 만든다.
2. 구현할 메서드를 선언한다.

```
package day18;

public interface MyCalculator {
        int plus(int num1, int num2);
}
```

3. 이때 메서드는 한개만 정의한다.

```
package day18;

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

```
                MyCalculator calcu = (x, y) -> x + y;
                int result = calcu.plus(20, 30);

                System.out.println(result);
        }
}
```

4. 람다식을 구현할 때 되도록 생략할 수 있는 부분은 생략하여 구현

함수형 인터페이스(Functional Interface)

- 람다식은 메서드 이름이 없고 메서드를 실행하는 데 필요한 매개변수와 매개변수를 활용한 실행코드를 구현하는 것
- 따라서 람다식을 구현하기 위해서는 함수형 인터페이스를 만들고 인터페이스에 람다식으로 구현할 메서드를 선언 하는 것
- 람다식은 오직 하나의 추상 메서드만 선언한 인터페이스를 구현할 수 있다
    - 람다식은 이름이 없는 익명 함수로 구현하기 때문에 인터페이스에 추상 메서드가 여러 개 있다면 어떤 메서드를 구현한 것인지 모호해 지므로 하나의 메서드만 허용
    - 다만 구현체가 있는 static메서드와 default 메서드는 정의 할 수 있습니다(모호함이 없으므로)

    @FunctionalInterface 애노테이션

    - 람다식은 하나의 추상 메서드를 선언해야 하므로 실수하기 쉽다
    - 이러한 실수를 방지하기 위해 @FunctionalInterface 애노테이션을 사용합니다.
    - @FunctionalInterface 애노테이션을 선언하고 메서드를 하나 이상 선언하면 오류가 발생하므로 실수를 방지할 수 있다.

        ```
        package day18;

        @FunctionalInterface
        public interface MyCalculator {
                int plus(int num1, int num2);
        }
        ```

- 람다식은 익명 내부 클래스를 구현하는 것
    - 함수형 언어에서는 함수만 따로 호출할 수 있지만, 자바에서는 참조 변수 없이 메서드를 호출할 수 없다.
    - 람다식은 익명(지역) 내부 클래스를 구현하고 생성된 인스턴스에서 선언된 메서드를 호출하는 것과 같다.
    - 따라서 람다식은 익명 내부 클래스의 속성과 동일하다.
        - 람다식 내에서 사용되는 지역변수는 상수화 된다.

익명(지역) 내부 클래스는 메서드내에서 구현되고 익명(지역) 내부 클래스의 인스턴스 메서드가 호출되는데, 메서드내에 지역변수를
익명(지역) 내부클래스의 메서드가 참조하고 있다면 메서드 호출이 완료가 되도 제거가 되면 안된다.
(메서드는 호출이 종료가 되면 스택메모리 영역에서 자원이 제거가 된다)
따라서 제거가 되지 않기 위해 항상 유지가 되는 데이터 영역의 상수형태로 컴파일 하여 변환한다
(변수 앞에 final 키워드가 생략되어 있다라고 보면 된다.)

- 람다식으로 코드의 표현이 간단해 졌지만 근본적으로 달라진 것은 없다.

```
package day18;

public class MyCaculatorTest2 {
	public static void main(String[] args) {

		int num3 = 10;
		MyCalculator calcu = new MyCalculator() {
			@Override
			public int plus(int num1, int num2) {
				//num3 = 20; 익명(지역) 내부 클래스의 메서드에서 사용되는 지역 변수는 상수화 된다
				int result = num1 + num2 + num3;
				return result;
			}
		};


		int result = calcu.plus(20, 30);
		System.out.println(result);
	}
}
```

함수형 인터페이스 타입의 매개변수와 반환 타입

- 매개변수

```
package day18;

public class MyCalculatorParamTest {
	public static void main(String[] args) {
		calcuPlus((x, y) -> x + y);
	}

	public static void calcuPlus(MyCalculator calcu) {
		int result = calcu.plus(10, 20);
		System.out.println(result);
	}
}
```

- 반환타입

```
package day18;

public class MyCalculatorReturnTest {
	public static void main(String[] args) {
		MyCalculator calcu1 = calcuPlus();
		int result1 = calcu1.plus(10,20);
		System.out.println(result1);

		MyCalculator calcu2 = calcuPlus2();
		int result2 = calcu2.plus(10,20);
		System.out.println(result2);
	}

	public static MyCalculator calcuPlus() {
		MyCalculator calcu = (x, y) -> x + y;

		return calcu;
	}

	public static MyCalculator calcuPlus2() {
		return (x, y) -> x + y;
	}
}
```

java.util.function패키지

- 대부분의 메서드는 타입이 비슷하다
- 매개변수가 없거나 한 개, 두 개, 반환값이 없거나 한 개이다.
- 제네릭 메서드로 정의하면 매개변수나 반환 타입이 달라도 문제가 되지 않는다.
- java.util.function 패키지에 일반적으로 자주 쓰는 형식의 메서드를 함수형 인터페이스로 미리 정의해 놓았다.
- 매번 함수형 인터페이스를 정의하기 보다는 가능하면 이 패키지의 인터페이스를 활용하는 것이 좋다.

java.util.function 패키지의 주요 함수형 인터페이스

함수형 인터페이스	메서드	설명
java.lang.Runnable	void() run()	매개변수도 없고 반환값도 없음
Supplier	T get()	매개변수는 없고 반환값만 있음
Consumer	void accept(T t)	Supplier와 반대로 매개변수만 있고, 반환값이 없음
Function<T,R>	R apply(T t)	일반적인 함수. 하나의 매개변수를 받아서 결과를 반환

함수형 인터페이스	메서드	설명
Predicate	boolean test(T t)	조건식을 표현하는데 사용됨. 매개변수는 하나. 반환값은 boolean

참고) 타입문자 'T'는 'Type'을 'R'은 'Return Type'을 의미한다.

예제...

매개변수가 두 개인 함수형 인터페이스

매개변수가 두 개인 함수형 인터페이스는 이름 앞에 접두사 'Bi'가 붙는다.

함수형 인터페이스	메서드	설명
BiConsumer<T,U>	void accept(T t, U u)	두개의 매개변수만 있고 반환값이 없음
BiPredicate<T,U>	boolean test(T t, U u)	조건식을 표현하는데 사용됨. 매개변수는 둘, 반환값은 boolean
BiFunction<T,U,R>	R apply(T t, U u)	두 개의 매개변수를 받아서 하나의 결과를 반환

- 참고) Supplier는 매개변수는 없고 반환값만 존재하는데, 매서드는 두 개의 값을 반환할 수 없으므로 BiSupplier가 없다.
- 두 개 이상의 매개변수를 갖는 함수형 인터페이스가 필요하면 직접 만들어 써야 한다.

```
@FunctionalInterface
interface TriFunction<T,U,V,R> {
        R apply(T t, U u, V v);
}
```

예제...

UnaryOperator와 BinaryOperator

- Function의 변형이다
- 매개변수의 타입과 반환타입의 타입이 모두 일치한다.
- UnaryOperator와 BinaryOperator의 상위 인터페이스는 각각 Function, BiFunction이다.

함수형 인터페이스	메서드	설명
UnaryOperator	T apply(T t)	Function의 하위 인터페이스, Function과 달리 매개변수와 결과의 타입이 같다.

함수형 인터페이스	메서드	설명
BinaryOperator	T apply(T t, T t)	BiFunction의 하위 인터페이스, BiFunction과 달리 매개변수와 결과 타입이 같다.

예제...

컬렉션 프레임워크와 함수형 인터페이스

컬렉션 프레임워크의 인터페이스에 다수의 디폴트 메서드가 추가 되었고 그 중 일부는 함수형 인터페이스를 사용한다.

인터페이스	메서드	설명
Collection	boolean removeIf(Predicate filter)	조건에 맞는 요소를 삭제
List	void replaceAll(UnaryOperator operator)	모든 요소를 반환하여 대체
Iterable	void forEach(Consumer action)	모든 요소를 변환하여 대체
Map	V compute(K key, BiFunction<K,V,V> f)	지정된 키의 값에 작업 f를 수행
Map	V computeIfAbsent(K key, Function(K,V) f)	키가 없으면 작업 f 수행 후 추가
Map	V computeIfPresent(K key, BiFunction(K,V,V) f)	지정된 키가 있을 때 작업 f수행
Map	V merge(K key, V value, BiFunction(V, V, V) f)	모든 요소에 병합작업 f를 수행
Map	void forEach(BiConsumer<K,V> action)	모든 요소에 작업 action을 수행
Map	void replaceAll(BiFunction(K,V,V) f)	모든 요소에 치환작업 f를 수행

```
package day18;

import java.util.*;

public class CollectionExam {
	public static void main(String[] args) {
		ArrayList<Integer> list = new ArrayList<>();
		for(int i = 0; i < 10; i++) {
			list.add(i);
		}

		// list의 모든 요소를 출력
		list.forEach(i->System.out.print(i + ","));
		System.out.println();
```

```
                // list에서 2 또는 3의 배수를 제거한다.
                list.removeIf(x -> x % 2 == 0 || x % 3 == 0);
                System.out.println(list);

                list.replaceAll(i->i*10); // list의 각 요소에 10을 곱한다.
                System.out.println(list);

                Map<String, String> map = new HashMap<>();
                map.put("1",  "1");
                map.put("2",  "2");
                map.put("3",  "3");
                map.put("4",  "4");

                // map의 모든 요소를 {k, v} 형식으로 출력
                map.forEach((k, v) -> System.out.println("{" + k + "," + v +
"},"));

                System.out.println();
        }
}
```

기본형 사용하는 함수형 인터페이스

- 매개변수와 반환값의 타입이 모두 제네릭 타입이다. 기본형 값을 처리할 때도 래퍼(Wrapper)클래스를 사용해왔다.
- 기본형 대신 래퍼클래스를 사용하는 것은 비효율적이다.
- 따라서 보다 효율적으로 처리할 수 있도록 기본형을 사용하는 함수형 인터페이스를 제공한다.

함수형 인터페이스	메서드	설명
DoubleToIntFunction	int applyAsInt(double d)	**A**To**B**Function은 입력이 A타입, 출력이 B 타입
ToIntFunction	int applyAsInt(T value)	To**B**Function은 출력 타입이 B타입이다. 입력은 지네릭 타입
IntFunction	R apply(T t, U u)	**A**Function은 입력이 A타입이고 출력은 지네릭 타입
ObjIntConsumer	void accept(T t, U u)	Obj**A**Function은 입력이 T, A타입이고 출력은 없다.

```
Function<Integer, Integer> f = a -> a*2;  // 매개변수 타입과 반환 타입이 Integer

IntFunction<Integer> f = a -> a*2;  // 매개변수 타입과 반환타입이 Integer
```

```
예제...
```

Function의 합성과 Predicate의 결합

Function의 합성

```
default <V> Function<T, V> andThen(Function<? super R, ? extends V> after)
default <V> Function<V, R> compose(Function<? super V, ? extends T> before)
static <T> Function<T, T> identity()
```

- 두 람다식을 합성해서 새로운 람다식을 만들 수 있다.
- f.andThen(g) - 함수 f를 먼저 적용하고 그 다음에 함수 g를 적용한다.
- f.compose(g) - g를 먼저 적용하고 f를 적용한다.
- Function.identity() - 함수를 적용하기 이전과 동일한 항등함수, x -> x

f.andThen(g)

```
Function<String, Integer> f = (s) -> Integer.parseInt(s, 16);
Function<Integer, String> g = (i) -> Integer.toBinaryString(1);
Function<String, String> h = f.andThen(g);

System.out.println(h.apply("FF")); "FF" -> 255 -> "11111111"
```

f.compose(g)

```
Function<Integer, String> g = (i) -> Integer.toBinaryString(i);
Function<String, Integer> f = (s) -> Integer.parseInt(s, 16);
Function<Integer, Integer> h = f.compose(g);

System.out.println(h.apply(2)); // 2 -> "10" -> 16
```

Function.identity()

```
Function<String, String> f = x -> x;
Function<String, String> f = Function.identity(); // 위 문장과 동일

System.out.println(f.apply("Hello")); // Hello가 그대로 출력됨
```

```
package day18;

import java.util.function.*;

public class FunctionComposeExam {
        public static void main(String[] args) {
                Function<String, Integer> f = (s) -> Integer.parseInt(s, 16);
                Function<Integer, String> g = (i) -> Integer.toBinaryString(i);

                Function<String, String> h = f.andThen(g);
                Function<Integer, Integer> h2 = f.compose(g);
```

```
                System.out.println(h.apply("FF")); // "FF" -> 255 -> "11111111"
                System.out.println(h2.apply(2)); // 2 -> "10" -> 16

                Function<String, String> f2 = x -> x; // 항등 함수(identity
function)
                System.out.println(f2.apply("Hello")); // Hello가 그대로 출력됨
        }
}
```

Predicate의 결합

```
default Predicate<T> and(Predicate<? super T> other)
default Predicate<T> or(Predicate<? super T> other)
default Predicate<T> negate()   // 조건식 전체가 부정이 된다.
static <T> Predicate<T> isEqual(Object targetRef)
```

- Predicate를 and(), or(), negate()로 연결해서 하나의 새로운 Predicate로 결합할 수 있다.
- Predicate의 끝에 negate()를 붙이면 조건식 전체가 부정이 된다.
- static 메서드인 isEqual()은 두 대상을 비교하는 Predicate를 만들때 사용한다.

```
package day18;

import java.util.function.*;

public class PredicateExam {
        public static void main(String[] args) {
                Predicate<Integer> p = i -> i < 100;
                Predicate<Integer> q = i -> i < 200;
                Predicate<Integer> r = i -> i % 2 == 0;
                Predicate<Integer> notP = p.negate(); // i >= 100

                Predicate<Integer> all = notP.and(q.or(r));
                System.out.println(all.test(150)); // true

                String str1 = "abc";
                String str2 = "abc";

                // str1과 str2가 같은지 비교한 결과를 반환
                Predicate<String> p2 = Predicate.isEqual(str1);
                boolean result = p2.test(str2);
                System.out.println(result);
        }
}
```

메서드 참조

- 람다식을 더욱 간결하게 표현할 수 있다.
- 람다식이 하나의 메서드만 호출하는 경우에는 메서드 참조(method reference)라는 방법으로 람다식을 간결하게 할 수 있다.

- 하나의 메서드만 호출하는 람다식은 **클래스이름::메서드이름** 또는 **참조변수::메서드이름**으로 바꿀 수 있다.

```
Function<String, Integer> f = (String s) -> Integer.parseInt(s);

Function<String, Integer> f = Integer::parseInt; // 메서드 참조
```

```
BiFunction<String, String, Boolean> f = (s1, s2) -> s1.equals(s2);

BiFunction<String, String, Boolean> f = String::equals; // 메서드 참조
```

- 이미 생성된 객체의 메서드를 람다식에서 사용한 경우에는 클래스 이름 대신 그 객체의 참조변수를 적어야 한다.

```
MyClass Obj = new MyClass();
Function<String, Boolean> f = (x) -> obj.equals(x); // 람다식
Function<String, Boolean> f2 = obj::equals; // 메서드 참조
```

생성자의 메서드 참조

```
Supplier<MyClass> s = () -> new MyClass(); // 람다식
Supplier<MyClass> s = MyClass::new; // 메서드 참조
```

```
Function<Integer, MyClass> f = (i) -> new MyClass(i); // 람다식
Function<Integer, MyClass> f2 = MyClass::new;  // 메서드 참조

BiFunction<Integer, String, MyClass> bf = (i, s) -> new MyClass(i, s);
BiFunction<Integer, String, MyClass> bf2 = MyClass::new;  // 메서드 참조
```

```
Function<Integer, int[]> f = x -> new int[x];
Function<Integer, int[]> f2 = int[]::new;
```