JDK 1.8부터 추가된 람다식

1. 자바는 객체 지향 프로그래밍의 소프트웨어 개발의 주요 패러다임이었던 1990년대에 디자인 되었다. 최근 들어 함수적 프로그래밍이 다시 부각되고 있는데, 자바에서는 이벤트 지향 프로그래밍에 적합하기 때문이다. 그래서 객체 지향 프로그래밍과 함수적 프로그래밍을 혼합함으로써 더욱 효율적인 프로그래밍이 될 수 있도록 개발 언어가 변하고 있다.

2. 자바는 **함수적 프로그래밍을 위해 자바 8부터 람다식(Lambda Expressions)을 지원**하면서 기존의 코드 패턴이 많이 달라졌다.

3. 람다식의 도입으로 인해, 이제 자바는 **객체지향언어인 동시에 함수형 언어**가 되었다.

4. **람다식**은 간단히 말해서 **메서드를 하나의 식**으로 표현한 것이다.메서드를 람다식으로 표현하면 **메서드의 이름과 반환타입이 없어지므로 람다식을 '익명 함수(Anonymous function)' 라고 한다.**

5. 객체 지향언어에 익숙한 개발자들은 다소 혼란스러울 수 있지만, 자바는 람다식을 수용한 이유는 **자바 코드가 매우 간결**해지고, 컬렉션의 요소를 필터링하거나 매핑해서 원하는 결과를 쉽게 집계할 수 있기 때문이다. 그만큼 **코드 라인이 줄어들고 생략되어 진다.**

6. 람다식은 **메서드의 매개변수로 전달되어지는 것이 가능하고, 메서드의 결과로 반환**될 수 있다.

7. 람다식은 익명함수 답게 **메서드에서 이름과 반환타입을 제거**하고, **매개변수 선언부와 몸통{} 사이에 '->'를 추가**한다.

형식) 반환타입 메서드이름(매개변수 선언){
    몸체 문장;
    }

이 객체지향언어를 람다식으로 바꾸면

```
(매개변수 선언) -> {
  몸체 문장;
};
```

익명 구현객체를 람다식으로 변환한 예)

```
Runnable runnable= new Runnable(){
  @Override
  public void run(){
```

```
    실행문장;
  }
}; //익명 구현 객체
```

위의 익명 구현 객체 즉 익명클래스 문법을 람다식으로 변경하면 코드가 간결해지고 생략된다.

Runnable runnable = () -> { 실행문장;}; 벌써 내부적으로 run()메서드가 오버라이딩 되었다. 물론 해당 메서드는 생략된다.

람다식 기본 문법

1. 함수적 스타일의 람다식 작성방법

(타입 매개변수, ...) -> { 실행문; ..};

(타입 매개변수,....)는 **오른쪽 중괄호 {} 블록을 실행하기 위해 필요한 값을 제공**하는 역할을 한다. **매개변수명은 개발자가 자유롭게 줄 수 있다**. **-> 기호는 매개 변수를 이용해서 중괄호 { }를 실행한다는 뜻**이다.

예를 들어 int 매개 변수를 a의 값을 콘솔에 출력하기 위해 다음과 같이 람다식을 작성할 수 있다.

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

2. **매개 변수 타입은 런타임(실행)시 대입되는 값에 따라 자동으로 인식**될 수 있기 때문에 **람다식에서는 매개 변수 타입을 일반적으로 언급하지 않는다**. 즉 **생략할 수 있다**. 그래서 위 코드는 다음과 같이 변경할 수 있다.

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

3. **하나의 매개 변수만 있어면 소괄호 ( ) 를 생략**할 수 있다. **하나의 실행문장만 있다면 중괄호 { } 도 생략가능**하다. 그래서 위 코드를 다음과 같이 작성할 수 있다.

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

4. **매개 변수가 없다면 람다식에서는 매개 변수 자리가 없어지기 때문에 빈 소괄호 ( ) 를 꼭 해야 한다**.

```
() -> { 실행문; ... };
```

5. 중괄호 { } 를 실행하고 결과값을 리턴해야 한다면 다음과 같이 return문으로 결과값을 지정할 수 있다.

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

6. 중괄호 { } 에 return문만 있다면 람다식에서는 return문을 생략할 수 있다.

```
(x,y) -> x + y;
```

함수적 인터페이스와 람다식

1. **람다식은 인터페이스에 변수에 대입**된다. 형식은 다음과 같다.

인터페이스 변수 = 람다식;

람다식은 인터페이스의 익명 구현 객체를 생성한다는 뜻이 된다. 인터페이스는 직접 객체를 생성할 수 없기 때문에 구현 클래스가 필요한 데, 람다식은 익명 구현 클래스를 생성하고 객체화한다. 람다식은 대입될 인터페이스의 종류에 따라 작성 방법이 달라지기 때문에 **람다식에 대입될 인터페이스를 람다식의 타켓 타입**이라고 한다.

2. **함수적 인터페이스인 @FunctionalInterface는 자바 8에서 추가**되었다. **함수형 인터페이스로 선언된 것은 추상메서드가 딱 하나만 올수 있고,람다식을 위한 인터페이스**이다. 이 함수형 인터페이스를 람다식 타켓 타입으로 사용하면 된다.
모든 인터페이스를 람다식 타켓 타입으로 사용할 수 없다.

3. **람다식이 하나의 메서드를 정의하기 때문에 2개 이상의 추상메서드가 정의된 인터페이스는 람다식을 이용해서 구현 객체를 생성할 수가 없다. 하나의 추상메서드가 선언된 인터페이스만이 람다식의 타켓 타입으로 될 수 있다.**

4. @FunctionalInterface로 선언된 함수형 인터페이스에 만약 추상메서드가 2개 이상 온다면 컴파일 에러가 발생한다. 이 애노테이션은 선택사항이다. 이 어노테이션이 없더라도 하나의 추상메서드만 있다면 모두 함수형 인터페이스이고 람다식의 타켓타입으로 사용할 수 있다.

5. 매개변수와 리턴값이 없는 함수형 인터페이스와 람다식

```
@FunctionalInterface
public interface MyFun{
   void m();
}
MyFun fi = () -> { 실행문장; };
fi.m();
```

람다식에 매개변수가 없는 이유는 m()에 매개변수를 가지지 않기 때문이다. 람다식에 대입된 인터페이스 참조변수 fi로 m()메서드를 호출할 수 있다.

6. 매개 변수가 있고 리턴값이 없는 람다식

```
@FunctionalInterface
public interface MyFun{
   void m(int x);
}
```

```
MyFun mf = (x) -> {...}; 또는 x -> {....};
mf.m(5);
```
람다식에서 매개변수가 한 개 인 이유는 m()메서드가 매개변수를 하나만 가지기 때문이다.

7. 매개변수가 있고 리턴값이 있는 람다식

```
@FunationalInterface
public interface MyFun{
   int m(int x,int y);
}
MyFun my = (x,y) -> {....;    return 값; };
my.m(10,5);
```

람다식에서 매개변수가 2개인 경우는 m()메서드의 매개변수가 2개이기 때문이다. 그리고 m()가 리턴타입이 있기 때문에 람다식 {}에는 return문이 있어야 한다.
만약 중괄호에 return문만 있고, return문 뒤에 연산식이나 메서드 호출이 오는 경우라면 다음과 같이 작성 할 수 있다.

```
MyFun my = (x,y) -> {          =>    MyFun my= (x,y) -> x+y;
   return x+y;
}
MyFun my = (x,y) -> {          => MyFun my = (x,y) -> sum(x,y);
   return sum(x,y);
}
int result= my.m(2,5);
```

위의 람다식에서 {}와 return문을 생략할 수 있다.

8. var 키워드 소개

Java 10에서 var 키워드는 지역 변수의 타입을 자동으로 추론할 수 있도록 도입되었다. 즉, 변수 선언 시 타입을 명시적으로 지정하지 않고 var 키워드를 사용하여 컴파일러가 타입을 추론하게 할 수 있다. 결국 자바스크립트처럼 저장되는 값에 의해서 타입이 결정된다는 의미이다. Java 11부터 var 키워드를 사용하여 생성자나 메서드 내의 지역 변수를 선언할 수 있다. 이는 Java 10에서 도입된 var 키워드의 기능을 계속 확장한 것이다.
var 키워드는 Java에서 생성자나 메서드 내의 지역 변수 선언에서만 사용할 수 있으며, 클래스나 인터페이스의 멤버 변수(필드) 선언에는 사용할 수 없다.

예제

다음은 Java 11에서 생성자와 메서드 내에서 var를 사용하는 예제이다.

```
public class Example {

    // 생성자 내에서 var 사용
    public Example() {
        var localVariable = "Hello, World!"; // String 타입으로 추론
```

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

    // 메서드 내에서 var 사용
    public void printMessage() {
        var message = "Hello, Java 11!"; // String 타입으로 추론
        var number = 42; // int 타입으로 추론
        var list = List.of("A", "B", "C"); // List<String> 타입으로 추론

        System.out.println(message);
        System.out.println(number);
        System.out.println(list);
    }

    public static void main(String[] args) {
        Example example = new Example();
        example.printMessage();
    }
}
```

따라서 매개변수가 있는 람다식에서 함수형 인터페이스의 추상메서드에 매개변수가 있는 경우 해당 매개변수를 선언할 때 구체적인 타입 대신 var키워드를 사용할 수 있다.

```
(var 매개변수,...) -> {
   실행문;
   실행문;
 };
```

```
package 람다식;

//함수형 인터페이스 정의
@FunctionalInterface
interface Greeting {
        void greet(String name);
}
public class LambdaVarExample {
        public static void main(String[] args) {
                // 람다식 구현
                Greeting greeting = (var name) ->{
                        System.out.println("Hello, " + name);
                };
```

```
                greeting.greet("World");
        }
}
```

(var 매개변수, ...) -> 실행문

```
package 람다식;

//함수형 인터페이스 정의
@FunctionalInterface
interface Greeting {
        void greet(String name);
}
public class LambdaVarExample {
        public static void main(String[] args) {
                // 람다식 구현
                Greeting greeting = (var name) ->System.out.println("Hello, " +
name);
                greeting.greet("World");
        }
}
```

내장 함수형 인터페이스 종류

종류	특징
Consumer	- 매개값이 있고 리턴값이 없다.
Supplier	- 매개값이 없고, 리턴값이 있다.
Function	- 매개값도 있고,리턴값도 있다.주로 매개값을 리턴값으로 매핑 즉 타입변환을 해준다.
Operator	- 매개값도 있고,리턴값도 있다. 주로 매개값을 연산하고 동일한 타입으로 결과를 리턴한다.
Predicate	- 매개값이 있고, 리턴 타입은 boolean - 매개값을 조사해서 true/false를 리턴

1. 디폴트,정적 메서드는 추상메서드가 아니기 때문에 함수형 인터페이스에 선언되어도 함수형 인터페이스의 성질을 잃지 않는다.

여기서 함수형 인터페이스 성질이란 하나의 추상메서드를 가지고 있고, 람다식으로 구현된 익명 구현 객체를 생성할 수 있는 것을 말한다.

2. 함수형 인터페이스는 하나 이상의 디폴트 또는 정적메서드를 가지고 있다.

Predicate 함수형 인터페이스의 디폴트 메서드 특징

1. and() 디폴트 메서드는 논리 연산자의 && 와 대응된다. 즉 두 Predicate가 모두 true 이어야 최종 결과값도 true이다.

2. or() 디폴트 메서드는 논리 연산자의 || 와 대응된다. 즉 두 Predicate중 하나라도 true 이면 최종 결과값은 true 이다.

3. negate() 디폴트 메서드는 논리 연산자 ! 와 대응된다. 즉 원래 Predicate가 true이면 최종 결과값은 false이고, 반대로 false이면 결과값은 true가 된다.

메서드 참조

1. 메서드 참조는 메서드를 참조해서 매개 변수의 정보 및 리턴 타입을 알아 내어, 람다식에서 불필요한 매개 변수를 제거하는 것이 목적이다.

2. 두 개의 값을 받아 큰 수를 리턴하는 Math 수학클래스의 max() 정적 메서드를 호출하는 람다식은 다음과 같다.

```
(left,right) -> Math.max(left,right);
```

이 **람다식을 정적 메서드 참조로 바꾸면 다음과 같이 변경**할 수 있다.

```
Math :: max;  즉 클래스명 :: 정적메서드명;
```

메서드 참조도 람다식과 마찬가지로 인터페이스의 익명 구현 객체로 생성된다.

3. **인스턴스 메서드 참조는 먼저 객체를 생성한 다음 객체명 :: 인스턴스메서드명;**
으로 기술한다.

(x,y) -> obj.m(x,y); 람다식을 인스턴스 메서드 참조로 바꾸면 다음과 같이 간단히 표현할 수 있다.

```
obj ::  m ; 여기서 obj는 외부 클래스 객체명이 된다.
```

4 . **매개변수 메서드 참조**

람다식에서 제공하는 a 매개변수의 메서드를 호출해서 b매개변수를 매개값으로 사용하는 경우도 있다.

```
(a,b) -> {a.인스턴스메서드(b);};
```

이 람다식을 a매개변수 메서드 참조로 바꾸면 다음과 같다.

a클래스명뒤에 :: 기호를 붙이고 인스턴스 메서드명을 기술한다.

a가 String클래스의 매개변수라면 **String :: 인스턴스메서드명;**

5. 생성자 참조

생성자 참조도 메서드 참조에 포함된다. 생성자를 참조한다는 것은 객체를 생성한다는 의미이다. 객체를 생성하고 리턴하도록 구성된 람다식은 생성자 참조로 바꿀수 있다.
다음 코드 람다식은 객체 생성후 리턴만 한다.

(a,b) -> { return new 클래스명(a,b);};

위 람다식을 생성자 참조로 변경하면 **클래스명 뒤에 :: 기호를 붙이고 new 연산자를 기술**하면 된다.

클래스명 :: new;

생성자가 오버로딩 된 경우 컴파일러는 함수형 인터페이스의 추상메서드와 동일한 매개변수 타입과 개수를 가진 생성자를 찾아 실행한다.