

# Java 8의 인터페이스 구문

추상 메서드 외에도 디폴트 메서드와 스테틱 메서드 정의가 가능하다.

```
interface 인터페이스명 {  
    //상수  
    타입 상수명 = 값;  
    //추상 메소드  
    타입 메소드명(매개변수,...);  
    //디폴트 메소드  
    default 타입 메소드명(매개변수,...) {...}  
    //정적 메소드  
    static 타입 메소드명(매개변수) {...}  
}
```

## 디폴트 메서드 선언

자바8에서 추가된 인터페이스의 새로운 멤버

```
[public] default 리턴타입 메소드명(매개변수, ...) { ... }
```

실행 블록을 가지고 있는 메서드

default 키워드를 반드시 붙여야 함

public 접근 제한이 자동으로 붙음

## 스테틱 메서드 선언 :

```
[public] static 리턴타입 메소드명(매개변수, ...) { ... }
```

# Java 8의 인터페이스 구문

## 익명 구현 객체

- 구현 클래스 작성을 생략하고 바로 구현 객체를 얻는 방법
  - 이름 없는 구현 클래스 선언과 동시에 객체 생성

```
인터페이스 변수 = new 인터페이스() {  
    //인터페이스에 선언된 추상 메소드의 실제 메소드 선언  
};
```

- 인터페이스의 추상 메서드들을 모두 재정의 해야 함
- 추가적으로 필드와 메서드 선언 가능하지만 익명 객체 안에서만 사용가능 하며 인터페이스 변수로는 접근 불가

## 디폴트 메서드 사용

인터페이스로 직접 사용 불가

구현 객체가 인터페이스 변수에 대입되어야 호출할 수 있는 인스턴스 메서드

모든 구현 객체가 가지고 있는 기본 메서드로 사용

필요에 따라 구현 클래스가 디폴트 메서드를 재정의할 수 있음

## 정적 메서드 사용

인터페이스로 바로 호출 가능

# 람다식이란?

## 자바 8부터 함수적 프로그래밍 위해 람다식 지원

### ■ 람다식(Lambda Expressions)을 언어 차원에서 제공

- 람다식 : "식별자없이 실행가능한 함수"

- 익명 함수(anonymous function)를 생성하기 위한 식

### ■ 자바에서 람다식을 수용한 이유

- 코드가 매우 간결해진다.
- 컬렉션 요소(대용량 데이터)를 필터링 또는 매핑해서 쉽게 집계할 수 있다.

가  
가  
가  
functional interface .  
가 ..

### ■ 자바는 람다식을 함수적 인터페이스의 익명 구현 객체로 취급

람다식 → 매개변수를 가진 코드 블록 → 익명 구현 객체

- 어떤 인터페이스를 구현할지는 대입되는 인터페이스에 달려있음

Runnable runnable = () -> { ... }; 람다식

# 람다식이란

람다식(lambda expression)이란 간단히 말해 메서드를 하나의 식으로 표현한 것이다.

//메서드

```
int min(int x, int y) {  
    return x < y ? x : y;  
}
```

가

.

//람다 표현식

**$(x, y) \rightarrow x < y ? x : y;$**

위의 예제처럼 메서드를 람다식으로 표현하면, 클래스를 작성하고 객체를 생성하지 않아도 메서드로 사용할 수 있으며, 다른 메서드 호출시 아규먼트로 전달될 수도 있고, 메서드의 결과값으로 반환될 수도 있다.

## Lambda Syntax

- No arguments: `() -> System.out.println("Hello")`

가

- One argument: `s -> System.out.println(s)`

- Two arguments: `(x, y) -> x + y`

- With explicit argument types:

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

- Multiple statements:

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

# 람다식 기본 문법

## 함수적 스타일의 람다식 작성법

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

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

- 매개변수의 타입은 실행시에 대입 값 따라 자동 결정 → 생략 가능
- 하나의 매개변수만 있을 경우에는 괄호 ( ) 생략 가능
- 하나의 실행문만 있다면 중괄호 { } 생략 가능
- 매개변수 없다면 괄호 ( ) 생략 불가
- 리턴값이 있는 경우, return 문 사용
- 중괄호 { }에 return 문만 있을 경우, 중괄호 생략 가능

# 타겟 타입과 함수적 인터페이스

## 타겟 타입(target type)

- 람다식이 대입되는 인터페이스
- 익명 구현 객체를 만들 때 사용할 인터페이스

인터페이스 변수 = 람다식;

## 함수적 인터페이스(functional interface)

- 하나의 추상 메서드만 선언된 인터페이스
- **@FunctionalInterface** 어노테이션 정의

하나의 추상 메서드만을 가지는지 컴파일러가 체크하여 두 개 이상의 추상 메서드가 선언되어 있으면 컴파일 오류 발생

# 타겟 타입과 함수적 인터페이스

## 매개변수와 리턴값이 없는 람다식

```
@FunctionalInterface
public interface MyFunctionalInterface {
    public void method();
}
```

```
MyFunctionalInterface fi = () -> { ... }
```

```
fi.method();
```

## 매개변수가 있는 람다식

```
@FunctionalInterface
public interface MyFunctionalInterface {
    public void method(int x);
}
```

```
MyFunctionalInterface fi = (x) -> { ... } 또는 x -> { ... }
```

```
fi.method(5);
```



# 타겟 타입과 함수적 인터페이스

## 리턴값이 있는 람다식

```
@FunctionalInterface
public interface MyFunctionalInterface {
    public int method(int x, int y);
}
```

```
MyFunctionalInterface fi = (x, y) -> { ...; return 값; }
```

```
int result = fi.method(2, 5);
```

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



```
MyFunctionalInterface fi = (x, y) -> x + y;
```

```
MyFunctionalInterface fi = (x, y) -> {
    return sum(x, y);
}
```



```
MyFunctionalInterface fi = (x, y) -> sum(x, y);
```

# 클래스 멤버와 로컬 변수 사용

## 클래스의 멤버 사용

- 람다식 실행 블록에는 클래스의 멤버인 필드와 메서드를 제약 없이 사용 가능
- 람다식 실행 블록 내에서 this는 **람다식을 실행한 객체를 참조**

```
public class ThisExample {  
    public int outterField = 10;  
  
    class Inner {  
        int innerField = 20;  
  
        void method() {  
            //람다식  
            MyFunctionalInterface fi= () -> {  
                System.out.println("outterField: " + outterField);  
                System.out.println("outterField: " + ThisExample.this.outterField + "\n");  
  
                System.out.println("innerField: " + innerField);  
                System.out.println("innerField: " + this.innerField + "\n");  
            };  
            fi.method();  
        }  
    }  
}
```

바깥 객체의 참조를 얻기  
위해서는 클래스명.this 를 사용

람다식 내부에서 this 는 Inner 객체를 참조

# 클래스 멤버와 로컬 변수 사용

## 로컬 변수의 사용

- 람다식은 함수적 인터페이스의 익명 구현 객체를 생성한다.
- 람다식에서 사용하는 외부 로컬 변수는 final 특성을 갖는다.

```
public class UsingLocalVariable {  
    void method(int arg) { //arg는 final 특성을 가짐  
        int localVar = 40; //localVar는 final 특성을 가짐
```

```
//arg = 31; //final 특성 때문에 수정 불가  
//localVar = 41; //final 특성 때문에 수정 불가
```

```
//람다식
```

```
MyFunctionalInterface fi= () -> {
```

```
    //로컬변수 사용
```

```
    System.out.println("arg: " + arg);
```

```
    System.out.println("localVar: " + localVar + "\n");
```

```
};
```

```
fi.method();
```

```
}
```

```
}
```