13 Lambda Expression

Object Oriented Programming

Lambda Expression에 대해 강의하겠습니다.

Terminologies

- Function (타 언어 programming): 기능과 동작을 정의

```
void abc(){
  //기능 및 동작
}
```

- Method
  - Java 에서의 function
  - Class 또는 Interface 내에 정의
  - 사용하려면 object 생성 후, object.method 형태로 call

```
class A{
  void methodABC(){
  //기능 및 동작
  }
}
```

- Functional Interface
  - abstract method를 1개만 가지는 interface

```
interface A{
  public abstract void abc();
}
```

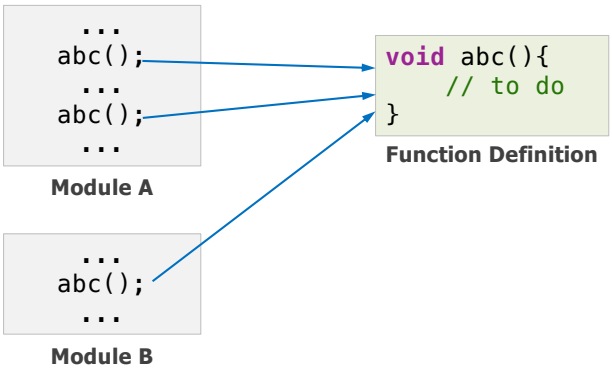

페이지 Two

여기서는 Lambda expression을 이해하는데 필요한 몇가지 용어들을 살펴보겠습니다.
일반적인 programming의 용어로 function은 기능 또는 동작을 정의한 일련의 명령 모음을 말합니다.
한편 Java에서 method는 class 또는 interface 내에 정의된 function이라 할 수 있습니다.
따라서 Java에서 어떤 function을 만들려면, 즉, 어떤 method를 사용하려면 항상 class object를 먼저 생성한 후 생성한 object로 method를 call해야 합니다.
Inheritance 같은 특수한 경우를 제외하면, 만일 모든 class에서 공통적으로 사용하는 기능이 있다고 해도, 모든 class마다 그 method를 정의해야 할 것입니다.

한편, interface 중에 abstract method를 단 1개만 가지는 경우를
functional interface라 부릅니다.
Labmda expression은 이 functional interface에만 적용될 수 있습니다.
자세한 것은 앞으로 차차 더 학습하겠습니다.

Functional Programming

- Function definition outside the execution module
- Function call

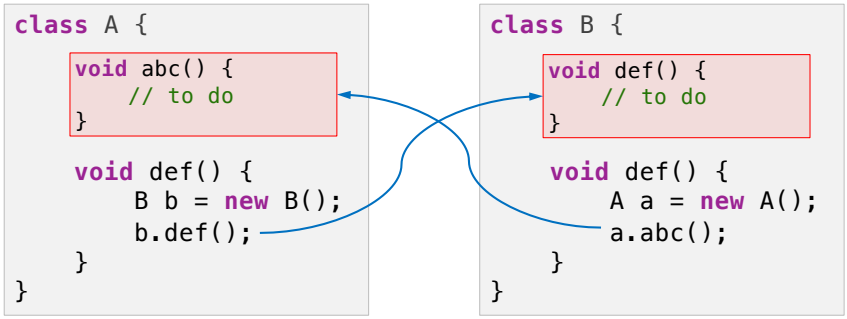


페이지 3

Object Oriented Programming이 아닌
일반적인 Functional Programming에서는
공통적인 기능을 하나의 function으로 독립적으로 정의한 후
각기 다른 module들에서 call하여 사용할 수 있습니다.
즉, 모든 module에서 그 공통 기능을 중복해서 각각 정의하는 대신,
독립적으로 정의된 function을 call하면
그 function이 제공하는 기능을
어떤 module에서라도 사용할 수 있다는 것입니다.
이와 같이 function call을 기반으로 하는 programming 방식을
functional programming이라 합니다.

Object Oriented Programming

- Method definition inside the class
- Method call via class object

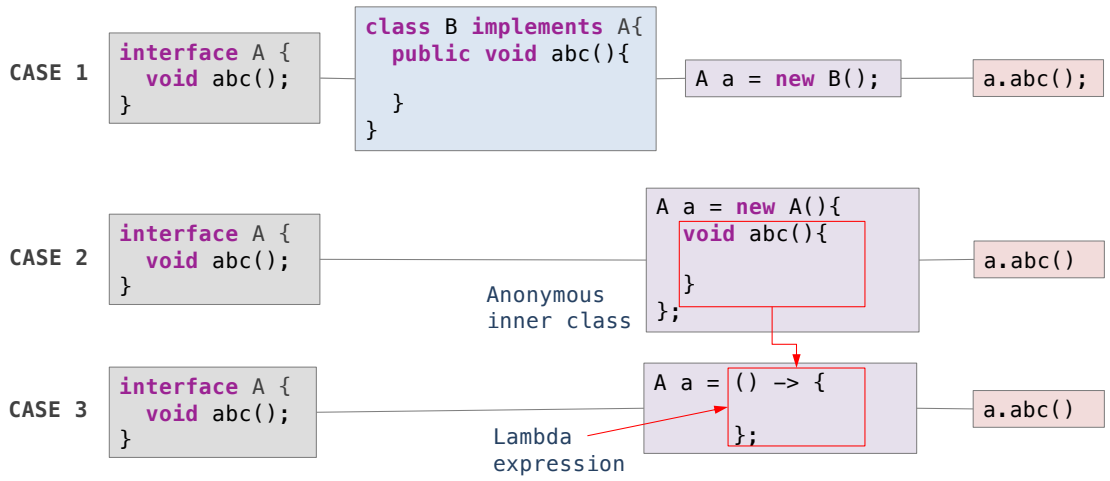


페이지 4

하지만 Java는 object oriented language 이므로
모든 method를 class 또는 interface 내부에 정의합니다.
즉, 외부에 독립적인 function를 구성할 수 없는 것입니다.
method는 항상 class 내부에 method로 존재해야 하고,
어떤 method를 사용하기 위해서는
class의 object를 먼저 생성한 후에 method를 call해야 합니다.
모든 method가 특정 class 안에 위치하고 있으므로
그 method의 기능을 사용하기 위해서는 어쩔 수 없는 과정입니다.
그림에서 보면 method abc가 class A 내부에 정의되어 있고
method def는 class B 내부에 정의되어 있습니다.
class A에서 method def를 call하려면
먼저 class B의 object인 small b를 생성하고

b.def() 로 call할 수 있습니다.
마찬가지로 class B 내부에서 abc()를 call하려면
먼저 class A의 object인 small a를 생성하고
a.abc() 로 call할 수 있습니다.
하지만 이는 외부에 어떤 기능을 가진 function를 정의하고,
이 function를 call함으로써 기능을 수행하는
본래의 functional programming과는 거리가 있다고 할 수 있고,
번거로와 보이기도 합니다.

Abstract Method Implementation and Call



페이지 5

앞에서의 문제를 해결하기 위해 나온 방법이 ' lambda expression ' 입니다.
Java는 새로운 function 문법을 정의하는 대신 ,
이미 있는 interface의 문법을 활용해 lambda expression을 표현합니다.
단 하나의 abstract method만을 포함하는 interface를 functional interface라 했던 것을 기억하고 있을 것입니다.
이 functional interface의 call 및 기능을 구현하는 방법을
새롭게 정의한 문법이 바로 lambda expression인 것입니다.
정리하면 lambda expression은 기존의 object oriented program 체계 안에서
functional programming을 가능하게 하는 기법이라

생각할 수 있습니다.

그렇다면, lambda expression을 문법적으로 정확하게 정의하기 전에
우선 Java의 object oriented 구조 내에서 lambda expression이 적용되는 과정을
간단히 살펴보겠습니다,
여기서 세 가지 case들은 functional interface를 상속받아 abstract method를 구현한 object를 생성한 후 method를 call하는 과정을
기존 방법과 lambda expression을 이용한 방법으로 비교하여 나타내고 있습니다.

첫 번째 case에서는 abstract method abc() 를 가진 interface A를 구현한 class B 를 생성합니다.
이후 class B의 생성자를 이용해 object를 생성한 후
object의 reference 변수 a를 통하여 a.abc()를 call합니다.

두 번째 case에서는 class의 명시적인 정의 없이
anonymous inner class를 사용해 object a를 생성하며,
이후 이 object를 통하여 a.abc() method를 call합니다.

마지막 세번째 case는 lambda expression을 활용한 방법으로
anonymous inner class의 method에서
method의 이름은 제외하고 body 부분만 가져와
lambda expression으로 method를 정의하고 call합니다.
즉, 문법적인 의미만 고려할 때
lambda expression은 anonymous inner class의 약식 표현이라 할 수 있습니다.

Example: OOPvsFP (1/2)

```
interface A {
    void abc();
}

class B implements A {
    @Override
    public void abc() {
        System.out.println(”method 1");
    }
}

public class OOPvsFP {
    public static void main(String[] args) {
        // CASE 1: object oriented 문법 1 (class implementation 사용)
        A a1 = new B();
        a1.abc();
```


페이지 6

이것은 앞의 세가지 case의 구현 방법을 보여주는
example program 입니다.
interface A가 abstract method abc() 를 가지고 있고
case 1에서처럼 class B는 interface A를
implements합니다.
Main class인 OOPvsFP 의 main method에서
먼저 case 1의 경우
class B의 object a1을 생성하고
이 object를 이용하여 method abc()를 call합니다.

Example: OOPvsFP (2/2)

```
        // CASE 2: object oriented 문법 2 (Anonymous class 사용)
        A a2 = new A() {
            @Override
            public void abc() {
                System.out.println("method 2");
            }
        };
        a2.abc();

        // CASE 3: Functional Programming 문법 (Lambda expression 사용)
        A a3 =  () -> {System.out.println("method 3");};
        a3.abc();
    }
}
```

OUTPUT:
method 1
method 2
method 3

페이지 7

Case 2에서는 interface를 명시적인 class로 implement하기 보다는
anonymous inner class를 사용하여 object를 생성합니다.
a2는 anonymous inner class의 object로 생성되었으며
a2를 이용하여 abc()를 call합니다.

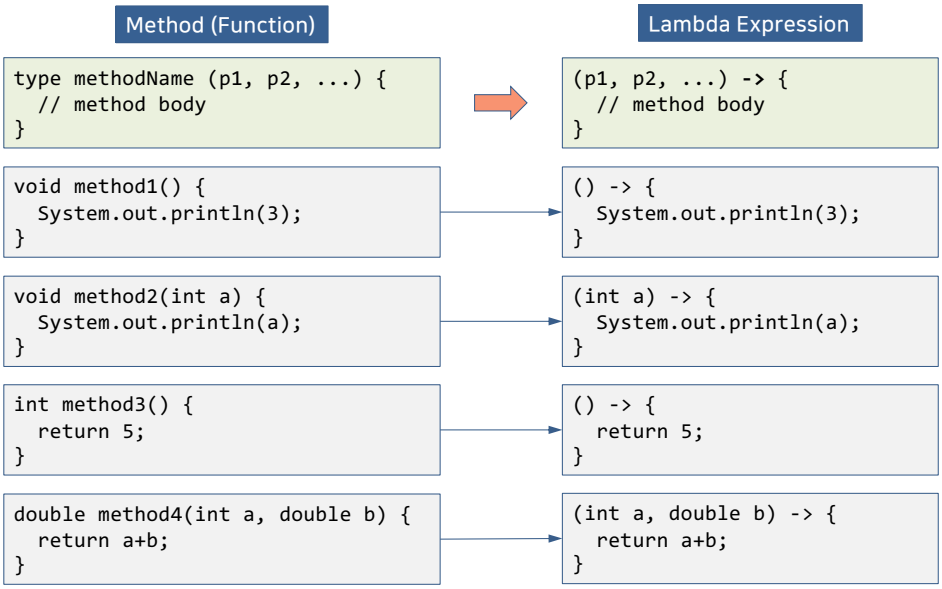
Case 3에서는 functional programming 구조를 따르기 위해
lambda expression을 사용하여 object a3를 생성하였습니다.
이 program의 실행 결과도 잘 출력되었습니다.

lambda expression은 anonymous inner class 정의 내부에 있는 method명을 포함해

이전 부분을 삭제한 형태입니다.
즉 , lambda expression은 anonymous inner class의
축약된 형태라고 볼 수 있습니다.

여기까지 이해했다면 앞에서 말한 것처럼
내부에 단 하나의 abstract method만을 가진 functional interface만
lambda expression으로 표현할 수 있는 이유를 유추할 수 있을 것입니다.
lambda expression은 내부 method명을 생략하므로
만일 구현해야할 abstract method가 두 개 이상이라면
어떤 method를 구현한 것인지 구분할 수 없기 때문입니다.

From Method to Lambda Expression



8

페이지 8

그럼 이제 본격적으로 class 내부에서 구현한
functional interface의 method 정의를
lambda expression으로 변환하는 문법에 대해
알아보겠습니다.
구현된 abstract method를 lambda expression으로
표현할 때는
method 이름 이후의 소괄호 ( )와 중괄호 { } 만을 차례대로
포함하며,
이들 사이에는 lambda expression 기호인 화살표 (->)가
들어갑니다.
소괄호는 input parameter , 중괄호는 method의 body를
나타냅니다.

실제 lambda expression으로의 변환의 예를 보면
첫번째는 parameter가 없는 void method의 경우 입니다.

두번째는 하나의 parameter가 있는 void method의 경우 입니다.
이때 lambda expression에도 parameter의 type을 명시할 수 있습니다.

세번째는 parameter가 없고 return value가 있는 경우인데, lambda expression에서는 return type을 명시하지 않으며 다만 body가 return 문으로 끝나게 됩니다.

네번째는 두 개의 parameter를 가지고
double value를 return하는 경우 입니다.

Shorthand Representation

Statement가 하나인 경우 중괄호 생략가능

```
A a = ()  -> {System.out.println("test");};   →   A a = () -> System.out.println("test");
```

Parameter's type 생략 가능

```
A a = (int a) -> { ... };   →   A a = (a) -> { ... };
```

Parameter 가 한 개인 경우 ( ) 생략 (type은 반드시 함께 생략)

```
A a = (int a) -> { ... };   →   A a = a → { … };
```

Statement로 return만 있는 경우 return 생략 가능 (중괄호도 반드시 함께 생략)

```
A a = (int a, int b) → { return a + b; };   →   A a = (a, b) -> a + b;
```


페이지 9

lambda expression은 anonymous inner class를 활용한 object 생성 방법의 축약 형태입니다.
이런 lambda expression을 특정 조건에서 더욱 축약해 표현할 수 있습니다.

먼저 중괄호 안의 실행문이 한 개일 때 중괄호는 생략할 수 있습니다.
두 줄 이상의 코드를 포함하고 있을 때는 당연히 생략할 수 없습니다.

두번째로 input parameter의 type은 생략할 수 있습니다.
이 생략이 가능한 이유는 functional interface에 포함된 abstract method에
input parameter type이 이미 명시되어 있기 때문에
컴파일러가 parameter의 type을 알아낼 수 있기

때문입니다.

input parameter가 한 개일 때는 소괄호도 생략할 수 있습니다.
단, 소괄호를 생략할 때는 parameter의 자료형을 반드시
생략해야 합니다.

마지막으로 method가 return 문 하나만으로 이뤄져 있을 때는
return 도 생략할 수 있습니다.
다만 return 을 생략할 때 중괄호를 반드시 함께 생략해야 합니다.

Three Uses for Lambda Expressions

- Type 1: Shorthand representations of anonymous inner class implementation methods
- Type 2: Method reference
  - Importing functionality that already exists, rather than new implementation of the method
- Type 3: Constructor reference
  - The implementation method is fixed by the object creation code alone

페이지 10

이제부터는 Lambda expression의 용도를
세가지 Type으로 나누어 설명하겠습니다.

Type 1에서는 지금까지 살펴 보았듯,
anonymous inner class 내부 구현 method의
약식 표현에 사용됩니다.

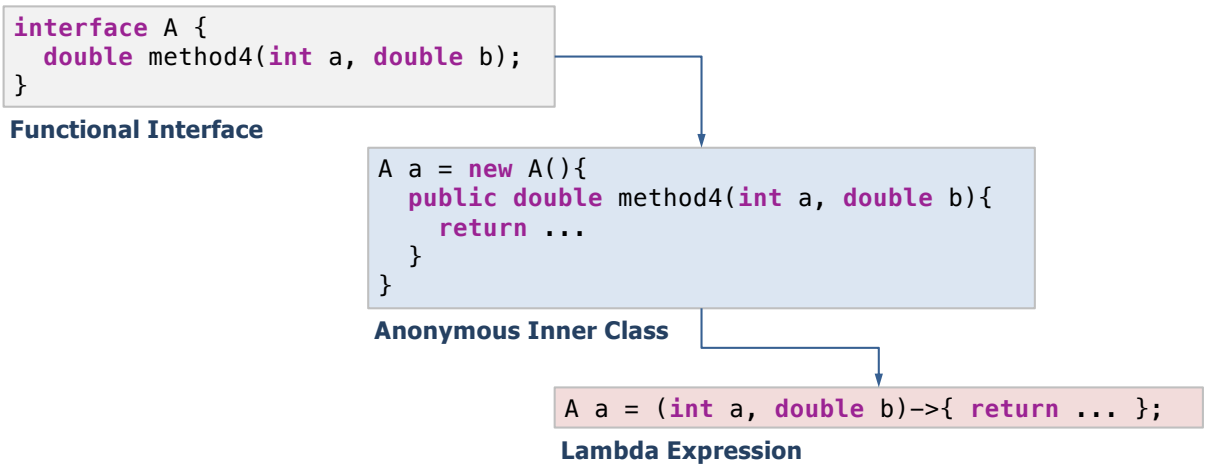
Type 2로는 lambda expression이 method
reference에 사용됩니다.
여기서 reference한다는 의미는
functional interface의 method를 구현하는 데 있어
직접 구현하는 대신,
이미 있는 기능을 가져다 쓰겠다는 의미입니다.
즉, anonymous inner class의 약식 표현에서
functional interface의 abstract method를 직접

구현하는 대신
이미 있는 method로 대체하는 것이 Type 2입니다.

마지막으로 Type 3에서 lambda expression은 생성자 reference에 사용됩니다.
생성자 reference는 구현 method의 내용이
object 생성 코드만으로 고정돼 있을 때 입니다.

결국 lambda expression은 functional interface의 abstract method를
어떤 방식으로 구현하느냐에 따라
이 세가지 형태를 띠게 됩니다.

Type 1: Shorthand Implementation of Method



페이지 11

Method implementation의 약식표현은
그 동안 우리가 계속 보아 오던 것들입니다.
먼저, Functional interface가 존재하는데
이전에 언급했듯이 functional interface에는
abstract method가 단 하나만 존재해야 합니다.
이 example에서는 interface A에
abstract method인 method4가 존재합니다.

이 functional interface를 implement하기 위해
기존에 우리가 사용했던 가장 간단한 방법은
Anonymous inner class를 사용하는 것이었습니다.
이 example에서는 Interface A의 object인 a를
생성하기위해
Anonymous inner class를 사용하였습니다.
method4도 이 anonymous class 안에 정의되었습니다.

이 Anonymous class를 lambda expression을 이용하여
더 축약할 수 있습니다.
method4라는 method name도 생략되고
parameter list와 body만 존재합니다.
사실 이 example에서는 body 안의 statement가
return 하나이기 때문에
중괄호와 return keyword도 생략될 수 있으며,
parameter list의 parameter type들도 생략될 수 있습니다.

Examples) FunctionToLambdaExpression2

```
interface Bftl2 {
    void method2(int a);
}
interface Dftl2 {
    double method4(int a, double b);
}

public class FunctionToLambdaExpression2 {
    public static void main(String[] args) {

        Bftl2 b2 = (int a)->{System.out.println("a = " + a);};
        b2.method2(3);

        Dftl2 d4 = (a, b)-> a + b;
        System.out.println(d4.method4(2,3));

    }
}
```

```
OUTPUT:
a = 3
5.0
```


페이지 12

이제 실제로 Lambda expression이 사용되는 example code를 보겠습니다.
Functional interface Bftl2는 abstract method인 method2를 가지고 있습니다.
method2는 void method이며 parameter는 int type 하나 입니다.
Functional interface Dftl2가 가진 method4는 int와 double 두개의 parameter를 가지며 return type은 double 입니다.
main method에서 Bftl2 interface의 object b2는 lambda expression으로 생성되었습니다.
b2.method2(3) 을 실행함으로써 output a = 3을 print합니다.
Dftl2의 object d4는 많이 축약된 버전의 lambda expression으로

parameter type과 return, 그리고 중괄호가 생략되어 있습니다.
d4.method4(2, 3) 을 call하면 output 5.00이 print됩니다.

Type 2: Instance Method Reference

```java
interface A {
    void abc();
}
class B {
    void bcd() {
        System.out.println("method");
    }
}
```

```java
// Lambda Expression
A a2 = () -> {
    B b = new B();
    b.bcd();
};

// Instance Method Reference
B b = new B();
A a3 = b::bcd;
```

- a2를 생성하는 Lambda expression의 역할은 b.bcd() 를 call하는 것 뿐
- 이 경우, A a3 = b::bcd; 와 같이 축약된 method reference를 사용 가능
- a3.abc() call을 b.bcd() call로 대체하라는 뜻

페이지 13

interface A에 abc() abstract method가 존재하고
class B에 bcd() method가 정의되어 있을 때
A의 object인 a2를 생성하는 lambda expression이 하는 역할은
B class의 object인 b를 생성하고
b.bcd() 를 call하는 것 뿐입니다.
이럴 경우 lambda expression을 더 축약하여
A a3 = b::bcd; 와 같이 나타낼 수 있습니다.
물론 이 경우 B의 object인 b는 미리 생성해 두어야 합니다.
이 표현은 a3.abc() call을 b.bcd() call로 대체하라는 뜻이 됩니다.

Type 2: Static Method Reference

```
interface A {
    void abc();
}
class B {
    static void bcd() {
        System.out.println("method");
    }
}
```

```
// Lambda Expression
A a2 = () -> {
    B.bcd();
};

// Static Method Reference
A a3 = B::bcd;
```

- a2를 생성하는 Lambda expression의 역할은 class B의 static method인 B.bcd()를 call하는 것 뿐
- 이 경우, A a3 = B::bcd; 와 같이 축약된 method reference를 사용 가능
- a3.abc() call을 B.bcd() call로 대체하라는 뜻

페이지 14

interface A에 abc() abstract method가 존재하고
이번에는 class B에 정의된 bcd()가 static method인 경우 입니다.
A의 object인 a2를 생성하는 lambda expression이 하는 역할은
B class의 static method인 B.bcd() 를 call하는 것 뿐입니다.
이럴 경우 lambda expression을 더 축약하여
A a3 = B::bcd; 와 같이 나타낼 수 있습니다.
이 표현은 a3.abc() call을 B.bcd() call로 대체하라는 뜻입니다.
bcd() 가 static method이기 때문에
단순히 method call을 위해 B의 object를 생성할 필요는 없습니다.

Type 3: Array Constructor Reference

```
interface A {
    int[] abc(int len);
}

// Lambda Expression
A a = (len) -> {  new int[len];  };

// Array Constructor Reference
A a = int[]::new;
```

- Lambda expression이 하는 역할은 length가 len인 int array를 생성하는 것 뿐
- 이런 경우 A a = int[]::new; 로 축약하여 표현 가능함

페이지 15

이번에는 interface A가 가진 abstract method의
return type이
array일 경우입니다.
이 interface A의 object small a를 생성하는
lambda expression은 new int[len]; 이 됩니다.
이 lambda expression을 더 축약하여 표현하면
A a = int[] :: new; 입니다.
이 표현은 a.abc(len) call을 new int[len]; call로
대체합니다.

Type 3: Class Constructor Reference

```
interface A {
    B abc(int);
}
class B {
    B() { }
    B(int k) { }
}
```

```
// Lambda Expression
A a = (k) -> new B(k);

// Class Constructor Reference
A a = B::new;
```

- 여기서 lambda expression의 역할은 B의 constructor B(int)를 call하여 reference a의 object를 생성하는 것
- Class constructor reference의 축약표현 A a = B::new; 는 B의 default constructor B() 를 call하는 것이 아니라 B(int) 를 call 하는 것임 (interface A의 정의로 부터 유추 가능함)

페이지 16

이 example에서는 Lambda expression의 역할이
B class 의 constructor인 B(int k) 만들 실행하는
것입니다.
이 때, 이 lambda expression의 축약된 표현은
A a = B :: new; 입니다.
여기서 new 뒤에 무슨 표현이 없기 때문에
B :: new 라는 표현이 B의 default constructor call을
의미한다고
오해하기 쉽습니다.
하지만 원래 interface A에 정의된 method abc가 int
parameter를 받고 있고
따라서 B :: new 라는 표현은 constructor B (int k) 를
의미한다고 해석될 수 있습니다.