(INTERMEDIATE) JAVA PROGRAMMING

10. Various forms of classes

Chapter 5

Review: 추상 메소드와 추상 클래스

- 추상 메소드(abstract method)
  - 선언되어 있으나 구현되어 있지 않은 메소드, abstract로 선언

```
public abstract String getName();
public abstract void setName(String s);
```

  - 추상 메소드는 서브 클래스에서 오버라이딩하여 구현해야 함

- 추상 클래스(abstract class)의 2종류
  1. 추상 메소드를 하나라도 가진 클래스
     - 클래스 앞에 반드시 abstract라고 선언해야 함
  2. 추상 메소드가 하나도 없지만 abstract로 선언된 클래스

Review: 추상 클래스의 용도

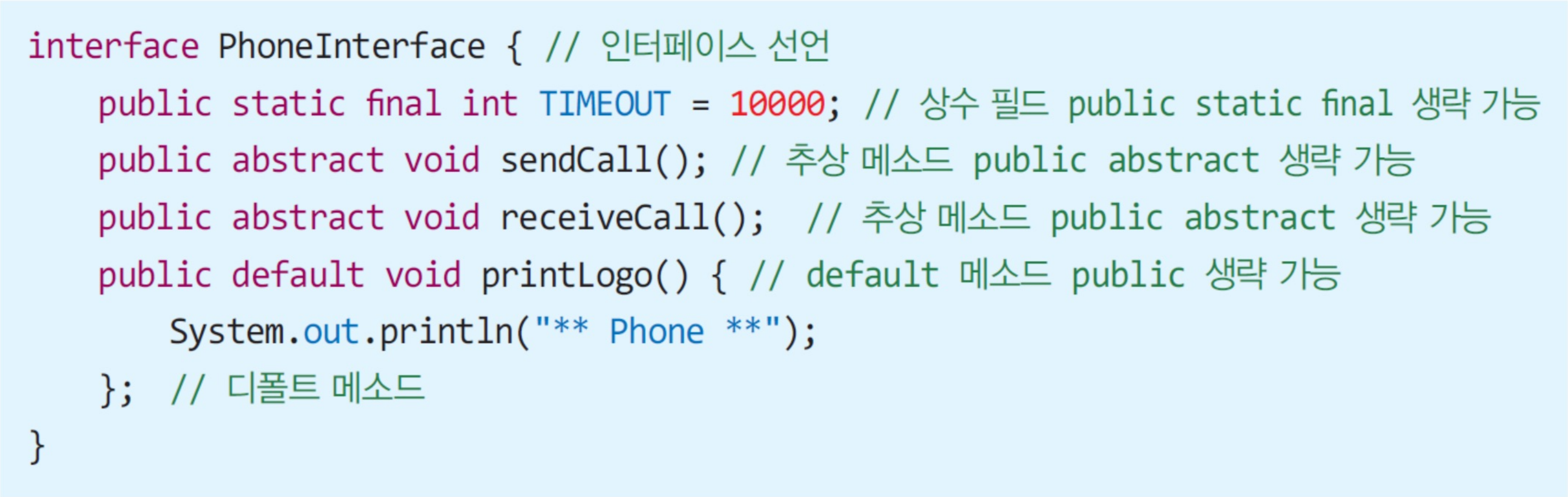
- 설계와 구현 분리
  - 슈퍼 클래스에서는 개념 정의
    - 서브 클래스마다 다른 구현이 필요한 메소드는 추상 메소드로 선언
  - 각 서브 클래스에서 구체적 행위 구현
    - 서브 클래스마다 목적에 맞게 추상 메소드 다르게 구현

- 계층적 상속 관계를 갖는 클래스 구조를 만들 때

Review: 인터페이스

- 자바의 인터페이스
  - 클래스가 구현해야 할 메소드들이 선언되는 추상형
  - 인터페이스 선언
    - **interface** 키워드로 선언
    - Ex) public **interface** SerialDriver {…}
- 자바 인터페이스에 대한 변화
  - Java 7까지
    - 인터페이스는 상수와 추상 메소드로만 구성
  - Java 8부터
    - 상수와 추상메소드 포함
    - default 메소드 포함 (Java 8)
    - private 메소드 포함 (Java 9)
    - static 메소드 포함 (Java 9)
  - 여전히 인터페이스에는 **필드(멤버 변수) 선언 불가**

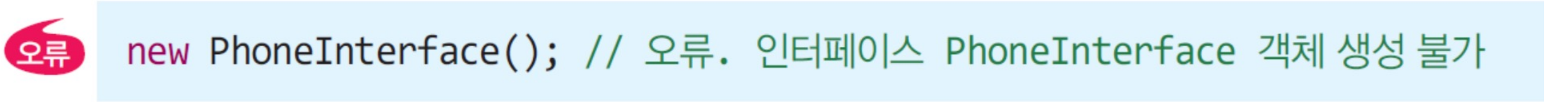
Review: 자바 인터페이스 예제



```
interface PhoneInterface { // 인터페이스 선언
    public static final int TIMEOUT = 10000; // 상수 필드 public static final 생략 가능
    public abstract void sendCall(); // 추상 메소드 public abstract 생략 가능
    public abstract void receiveCall();  // 추상 메소드 public abstract 생략 가능
    public default void printLogo() { // default 메소드 public 생략 가능
        System.out.println("** Phone **");
    }; // 디폴트 메소드
}
```

Review: 인터페이스의 전체적인 특징

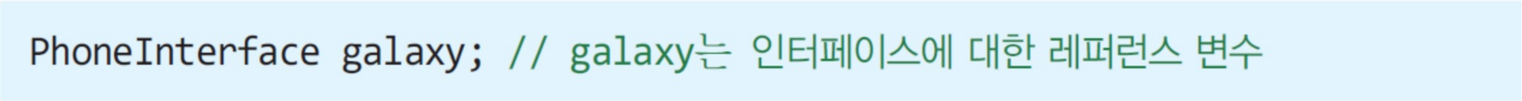
- 인터페이스의 객체 생성 불가



오류

```
new PhoneInterface(); // 오류. 인터페이스 PhoneInterface 객체 생성 불가
```

- 인터페이스 타입의 레퍼런스 변수 선언 가능

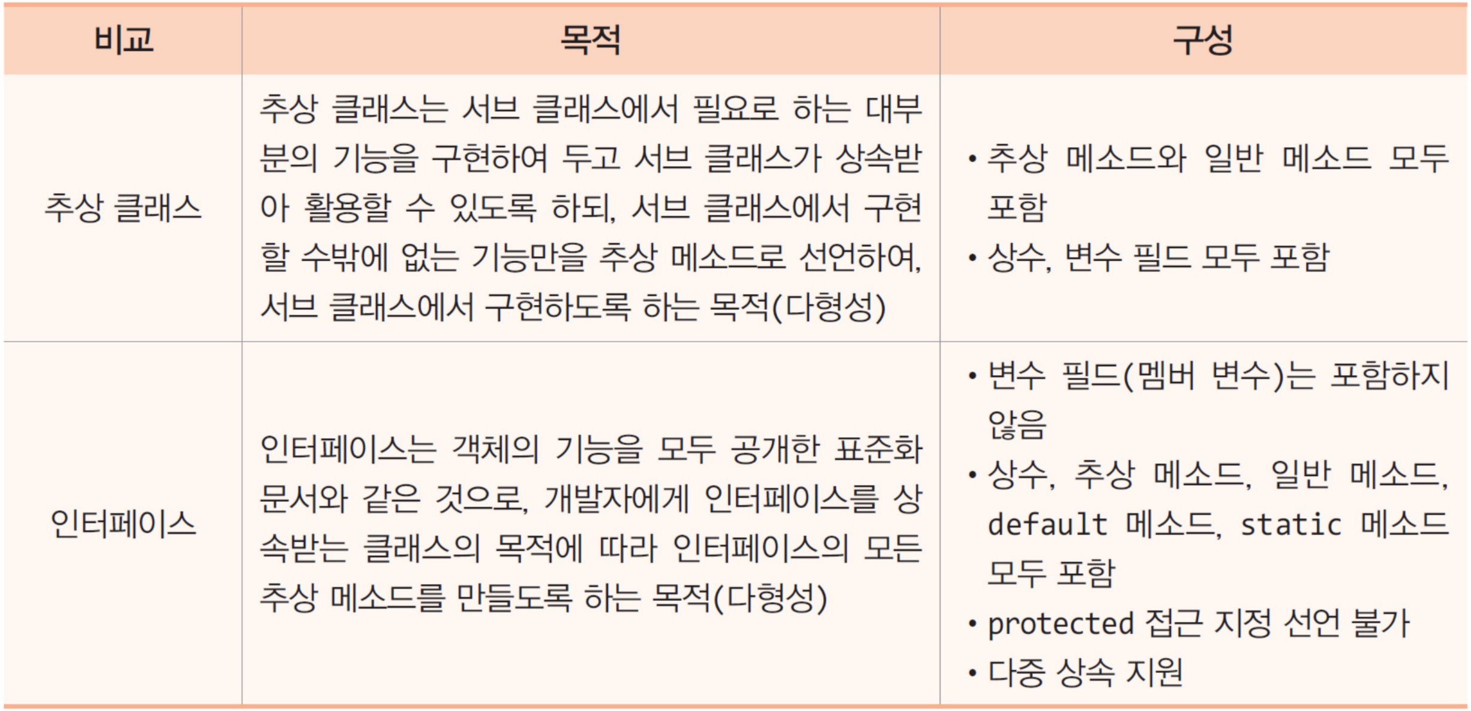


```
PhoneInterface galaxy; // galaxy는 인터페이스에 대한 레퍼런스 변수
```

- 인터페이스 구현
  - 인터페이스를 상속받는 클래스는 인터페이스의 모든 추상 메소드 반드시 구현
- 다른 인터페이스 상속 가능
- 인터페이스의 다중 상속 가능

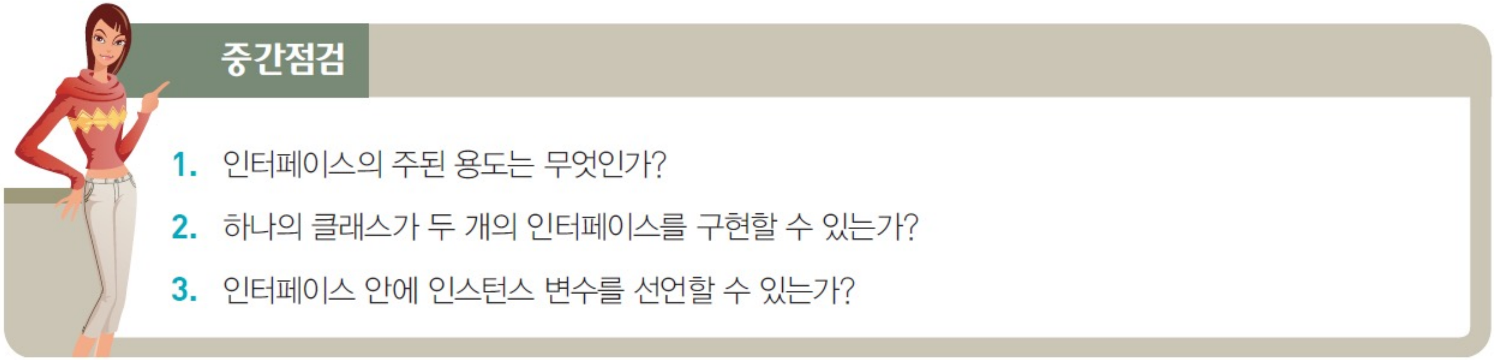
Review: 추상 클래스와 인터페이스 비교

- 유사점
  - 객체를 생성할 수 없고, 상속을 위한 슈퍼 클래스로만 사용
  - 클래스의 다형성을 실현하기 위한 목적
- 다른 점



비교	목적	구성
추상 클래스	추상 클래스는 서브 클래스에서 필요로 하는 대부분의 기능을 구현하여 두고 서브 클래스가 상속받아 활용할 수 있도록 하되, 서브 클래스에서 구현할 수밖에 없는 기능만을 추상 메소드로 선언하여, 서브 클래스에서 구현하도록 하는 목적(다형성)	• 추상 메소드와 일반 메소드 모두 포함 • 상수, 변수 필드 모두 포함
인터페이스	인터페이스는 객체의 기능을 모두 공개한 표준화 문서와 같은 것으로, 개발자에게 인터페이스를 상속받는 클래스의 목적에 따라 인터페이스의 모든 추상 메소드를 만들도록 하는 목적(다형성)	• 변수 필드(멤버 변수)는 포함하지 않음 • 상수, 추상 메소드, 일반 메소드, default 메소드, static 메소드 모두 포함 • protected 접근 지정 선언 불가 • 다중 상속 지원

중간 점검



중간점검

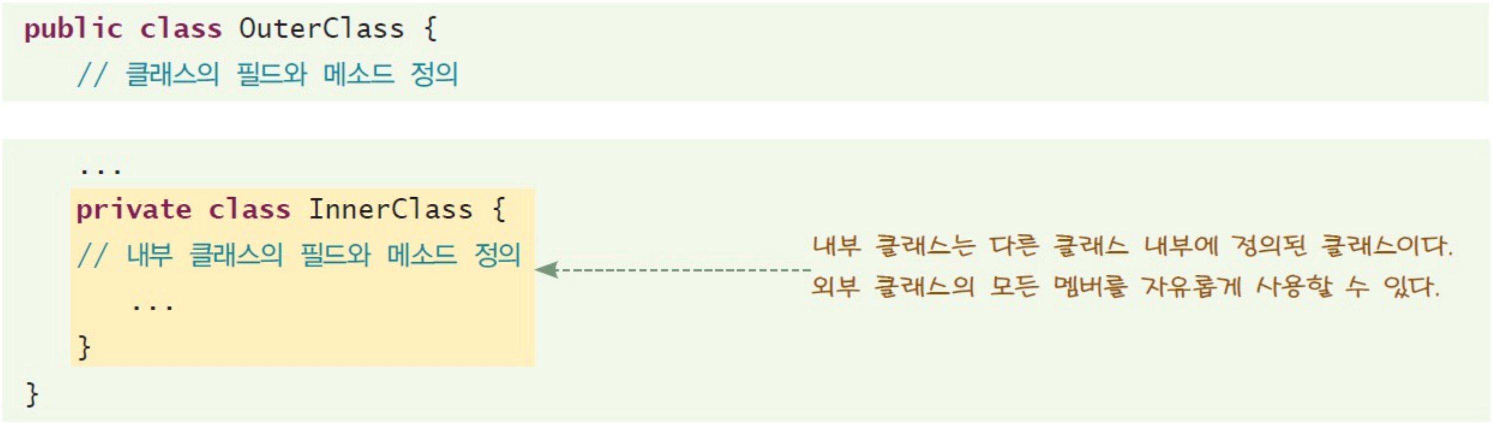
1. 인터페이스의 주된 용도는 무엇인가?
2. 하나의 클래스가 두 개의 인터페이스를 구현할 수 있는가?
3. 인터페이스 안에 인스턴스 변수를 선언할 수 있는가?

JAVA:

INNER CLASS

내부 클래스

- 내부 클래스(inner class): 클래스 안에 다른 클래스를 정의



```
public class OuterClass {
    // 클래스의 필드와 메소드 정의

    ...
    private class InnerClass {
    // 내부 클래스의 필드와 메소드 정의
        ...
    }
}
```

내부 클래스는 다른 클래스 내부에 정의된 클래스이다.
외부 클래스의 모든 멤버를 자유롭게 사용할 수 있다.

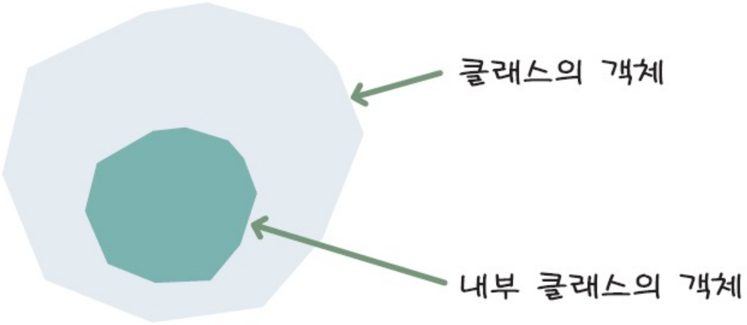
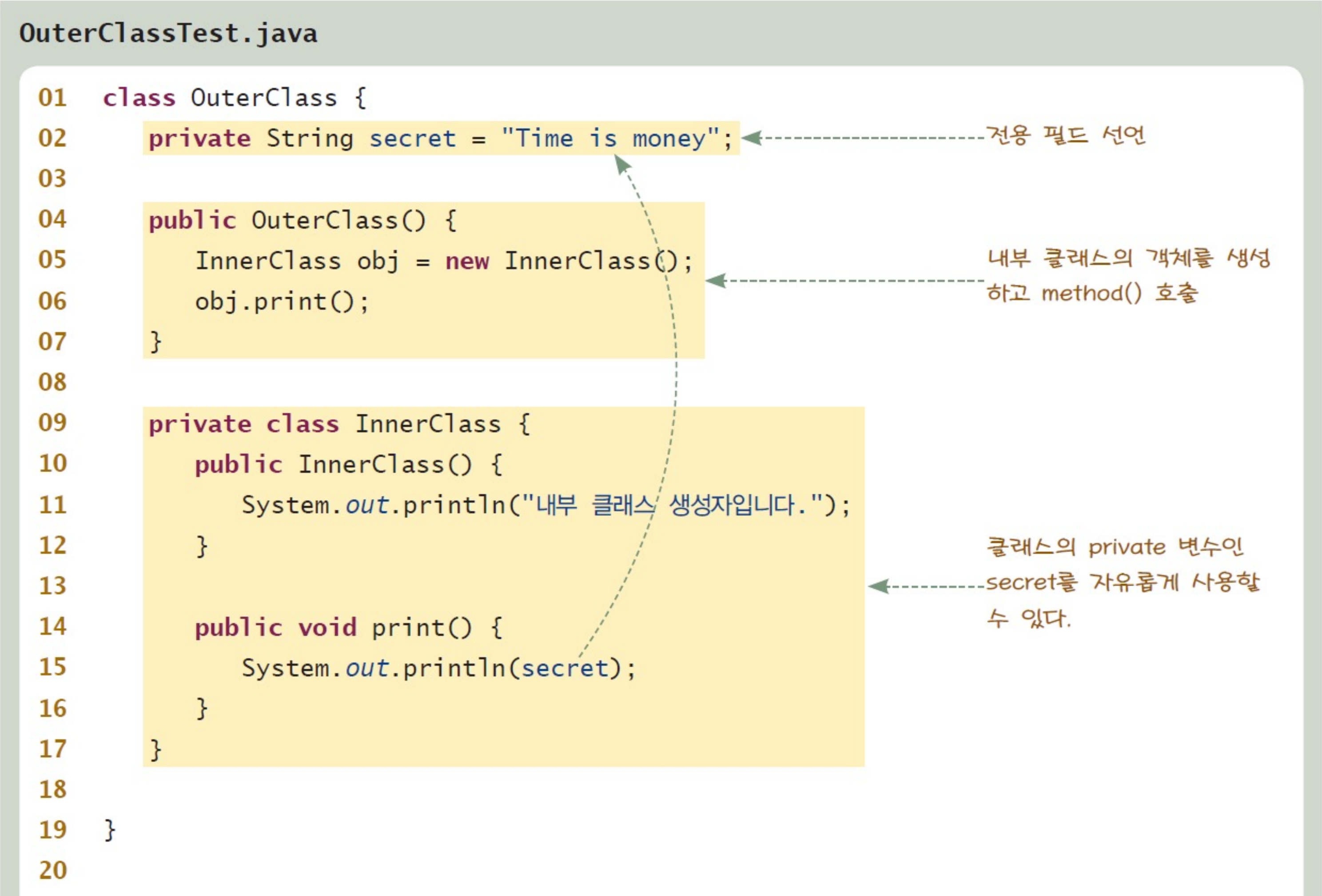


그림12-10. 내부 클래스 객체는 객체 안에 위치한다.

왜 내부 클래스를 사용하는가?

- 멤버 변수를 private로 유지하면서 자유롭게 사용할 수 있다.
- 하나의 장소에서만 사용되는 클래스들을 한곳에 모을 수 있다
- 보다 읽기 쉽고 유지 보수가 쉬운 코드가 된다.

예제



OuterClassTest.java

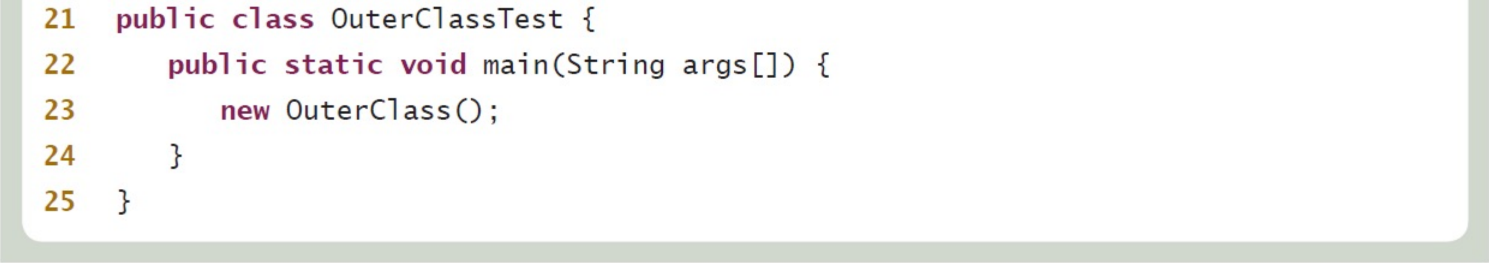
```
class OuterClass {
    private String secret = "Time is money";

    public OuterClass() {
        InnerClass obj = new InnerClass();
        obj.print();
    }

    private class InnerClass {
        public InnerClass() {
            System.out.println("내부 클래스 생성자입니다.");
        }

        public void print() {
            System.out.println(secret);
        }
    }

}

```

전용 필드 선언

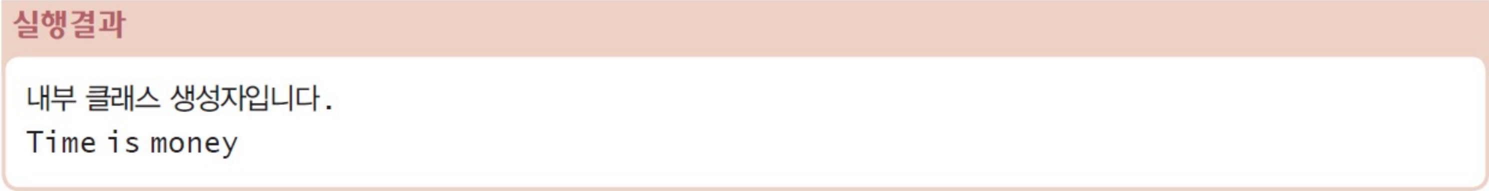
내부 클래스의 객체를 생성하고 method() 호출

클래스의 private 변수인 secret를 자유롭게 사용할 수 있다.

예제



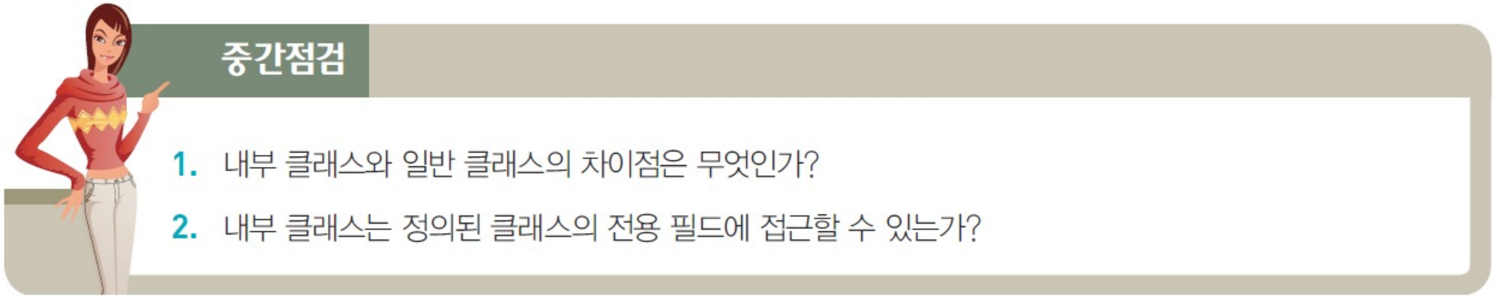
```java
public class OuterClassTest {
    public static void main(String args[]) {
        new OuterClass();
    }
}
```



실행결과

```
내부 클래스 생성자입니다.
Time is money
```

중간 점검



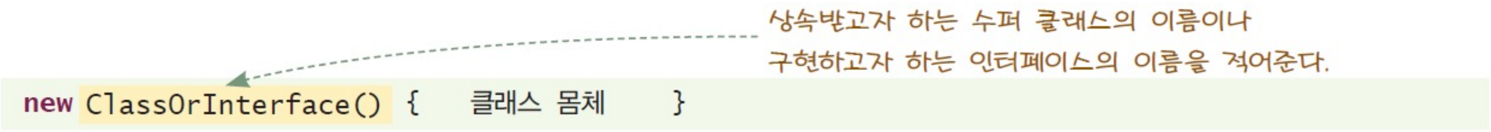
중간점검

1. 내부 클래스와 일반 클래스의 차이점은 무엇인가?
2. 내부 클래스는 정의된 클래스의 전용 필드에 접근할 수 있는가?

JAVA:
ANONYMOUS CLASS

무명 클래스

- 무명 클래스(anonymous class): 클래스 몸체는 정의되지만 이름이 없는 클래스

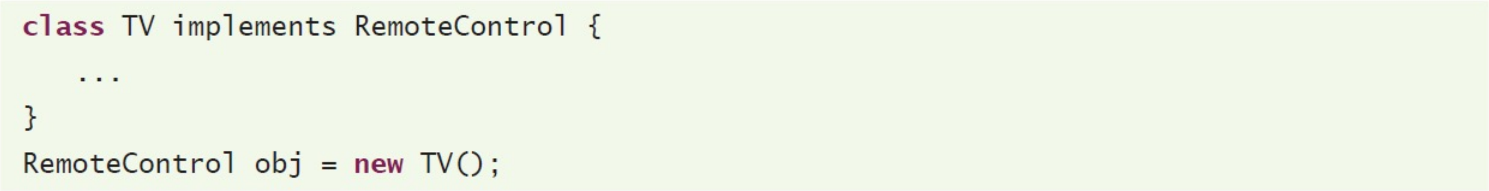


상속받고자 하는 수퍼 클래스의 이름이나
구현하고자 하는 인터페이스의 이름을 적어준다.

```
new ClassOrInterface() {    클래스 몸체    }
```

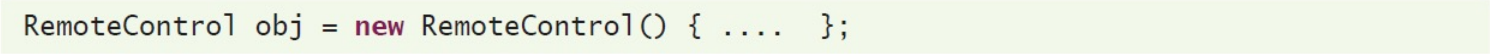
무명 클래스의 예

- 이름이 있는 클래스의 경우



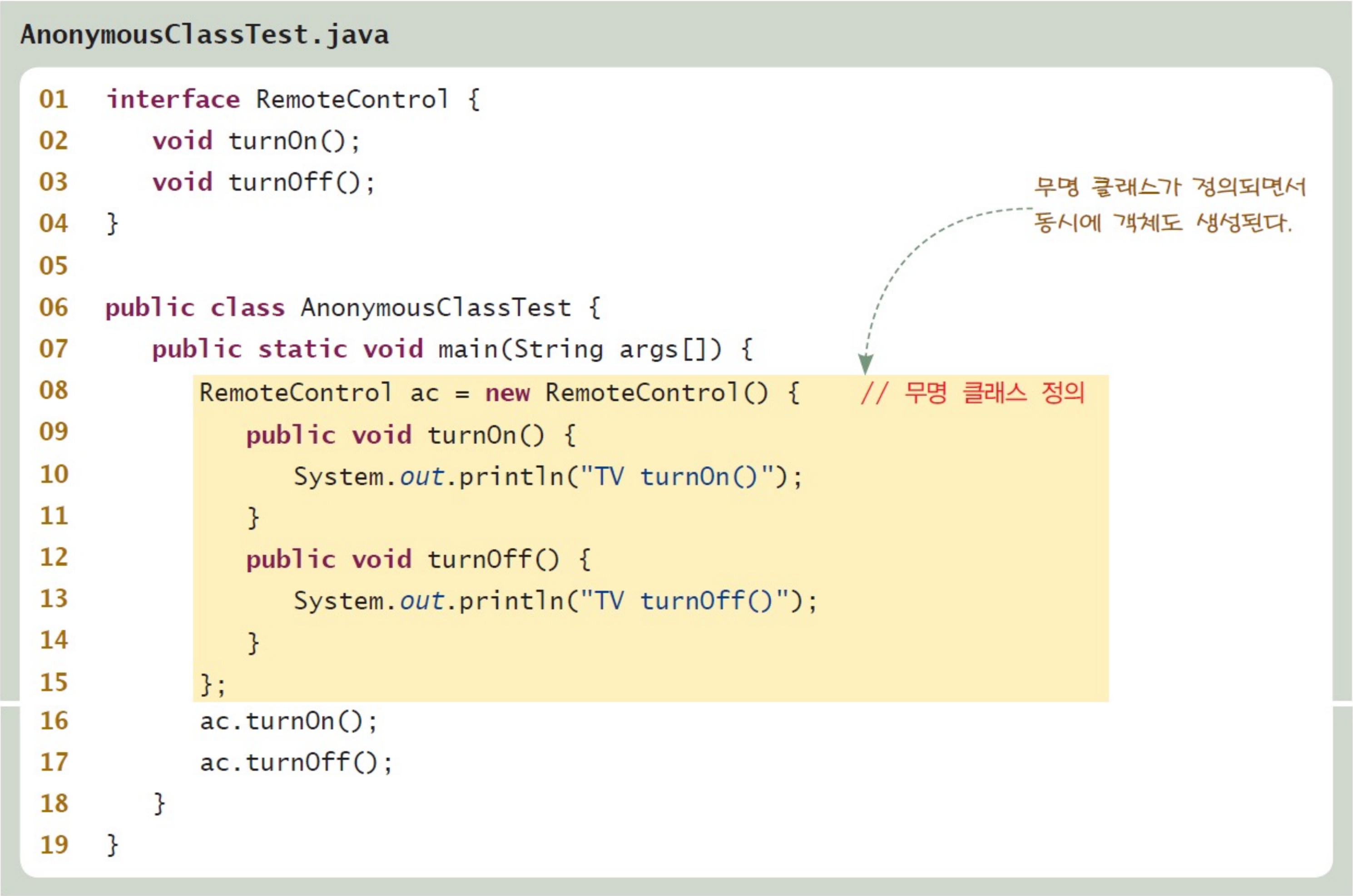
```
class TV implements RemoteControl {
    ...
}
RemoteControl obj = new TV();
```

- 무명 클래스의 경우



```
RemoteControl obj = new RemoteControl() { ....  };
```

예제

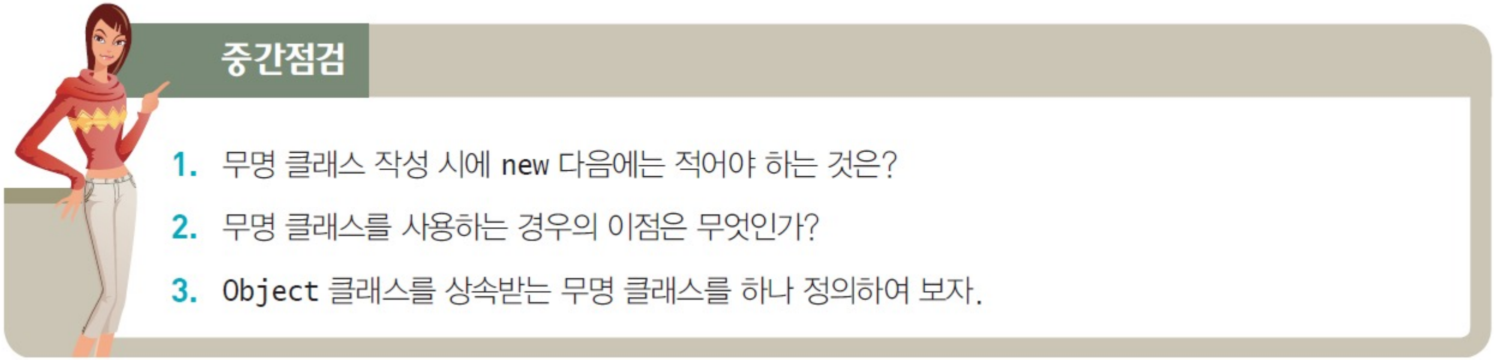


AnonymousClassTest.java

```java
interface RemoteControl {
    void turnOn();
    void turnOff();
}

public class AnonymousClassTest {
    public static void main(String args[]) {
        RemoteControl ac = new RemoteControl() {     // 무명 클래스 정의
            public void turnOn() {
                System.out.println("TV turnOn()");
            }
            public void turnOff() {
                System.out.println("TV turnOff()");
            }
        };
        ac.turnOn();
        ac.turnOff();
    }
}
```

무명 클래스가 정의되면서
동시에 객체도 생성된다.

중간 점검



중간점검

1. 무명 클래스 작성 시에 new 다음에는 적어야 하는 것은?
2. 무명 클래스를 사용하는 경우의 이점은 무엇인가?
3. Object 클래스를 상속받는 무명 클래스를 하나 정의하여 보자.

JAVA:

OPTIONAL: LAMBDA EXPRESSIONS

Lambda Expressions

- Added by JDK 8 - lambda expressions significantly enhance Java
  - add new syntax elements - increase the expressive power of the language
  - new capabilities - API library
    - ability to more easily take advantage of the parallel processing capabilities of multi-core environments
    - especially as it relates to the handling of for-each style operations
    - the new stream API, which supports pipeline operations on data.
- also provided the catalyst for other new Java features,
  - the default method - define default behavior for an interface
  - the method reference

Outline

- Introducing Lambda Expressions
  - Lambda Expression Fundamentals
  - Functional Interfaces
  - Some Lambda Expression Examples

Introducing Lambda Expressions

- Key to understanding Java’s implementation of lambda expressions
  - lambda expression
  - functional interface
- A lambda expression - an anonymous (unnamed) method
  - not executed on its own
  - Instead - implement a method defined by a functional interface.
  - results in a form of anonymous class

Introducing Lambda Expressions (cont.)

- A *functional interface* is an interface that contains one and only one abstract method.
- specifies the intended purpose of the interface.
- typically represents a single action.
- Furthermore, defines the *target type* of a lambda
- expression.
- a lambda expression can be used only in a context in which its target type is specified.
- *SAM type* - Single Abstract Method.

Lambda Expression Fundamentals

- The lambda expression - new syntax element and operator
- the *lambda operator* or the *arrow operator*, is −>.
- divides a lambda expression into two parts
- The left side specifies any parameters required
- by the lambda expression. (If no parameters are needed, an empty parameter list is used.)
- The right side is the *lambda body,* which specifies the actions of the lambda expression.
- −> verbalized as “becomes” or “goes to.”
- Java defines two types of lambda bodies
  - single expression
  - block of code

Examples

- It evaluates to a constant value:

```
() -> 123.45
```

- takes no parameters
- returns the constant value 123.45
- similar to the method:

```
double myMeth() { return 123.45; }
```

- the method defined by a lambda expression does not have a name

```
() -> Math.random() * 100
```

- This lambda expression obtains a pseudo-random value from Math.random( ), multiplies it by 100, and returns the result
- does not require a parameter.

Examples (cont.)

- a lambda expression with a parameter

```
(n) -> (n % 2)==0
```

- returns true if the value of parameter n is even. Although it is
- possible to explicitly specify the type of a parameter
- its type can be inferred.
- a lambda expression can specify many parameters

Functional Interfaces

- a functional interface is an interface that specifies only one abstract method
- with JDK 8 – default behavior for a method declared in an interface called a *default method.*
- an interface method is abstract only if it does not specify a default implementation
- nondefault interface methods are implicitly abstract
  - no need to use the abstract modifier

Functional Interfaces (cont.)

```
interface MyNumber {
  double getValue();
}
```

- the method getValue( ) is implicitly abstract
- the only method defined by MyNumber
- MyNumber is a functional interface
- its function is defined by getValue( ).

Functional Interfaces (cont.)

- how a lambda expression can be used in an
- assignment context

```
// Create a reference to a MyNumber instance.
MyNumber myNum;
// Use a lambda in an assignment context.
myNum = () -> 123.45;
```

Functional Interfaces (cont.)

- When a lambda expression occurs in a target type context
- an instance of a class is automatically created that implements the functional interface, with the lambda expression
- defining the behavior of the abstract method declared by the functional interface
- When that method is called through the target, the lambda expression is executed
- a lambda expression gives a way to transform a code segment into an object

Functional Interfaces (cont.)

- In the preceding example, the lambda expression becomes the implementation for the getValue( ) method
- As a result, the following displays the value 123.45:

```
 // Call getValue(), which is implemented by the previously assigned
// lambda expression.
  System.out.println("myNum.getValue());
```

- as the lambda expression assigned to myNum returns the value 123.45,

Functional Interfaces (cont.)

- the type of the abstract method and the type of the lambda expression must be compatible.
- For example, if the abstract method specifies two int parameters, then the lambda must specify two
- parameters whose type either is explicitly int or can be implicitly inferred as int by the context
- In general:
  - the type and number of the lambda expression's parameters must be compatible with the method's parameters
  - the return types must be compatible
  - any exceptions thrown by the lambda expression must be acceptable to the method

Example

- puts together the pieces shown in
- the foregoing section

interface MyNumber, class LambdaDema

```
// Demonstrate a simple lambda expression.
// A functional interface.
interface MyNumber {
   double getValue();
}

class LambdaDemo {
   public static void main(String args[])
   {
      MyNumber myNum; // declare an interface reference
      // Here, the lambda expression is simply a constant expression.
      // When it is assigned to myNum, a class instance is
      // constructed in which the lambda expression implements
      // the getValue() method in MyNumber.
      myNum = () -> 123.45;
```

class LambdaDemo

```
      // Call getValue(), which is provided by the previously assigned
      // lambda expression.
      System.out.println(“A fixed value: ‟ + myNum.getValue());

      // Here, a more complex expression is used.
      myNum = () -> Math.random() * 100;

      // These call the lambda expression in the previous line.
      System.out.println("A random value: " + myNum.getValue());
      System.out.println("Another random value: " + myNum.getValue());

      // A lambda expression must be compatible with the method
      // defined by the functional interface.
       // Therefore, this won't work:
      // myNum = () -> "123.03"; // Error!
   }
}
```

Output

```
A fixed value: 123.45
A random value: 88.90663650412304
Another random value: 53.00582701784129
```

Example: parameter with a lambda expression

- the use of a parameter with a lambda expression

interface NumericTest, class LambdaDemo2

```
// Demonstrate a lambda expression that takes a parameter.
// Another functional interface.
interface NumericTest {
   boolean test(int n);
}

class LambdaDemo2 {
   public static void main(String args[])
   {

      // A lambda expression that tests if a number is even.
      NumericTest isEven = (n) -> (n % 2)==0;
      if(isEven.test(10)) System.out.println("10 is even");
      if(!isEven.test(9)) System.out.println("9 is not even");

      // Now, use a lambda expression that tests if a number
      // is non-negative.
      NumericTest isNonNeg = (n) -> n >= 0;
      if(isNonNeg.test(1)) System.out.println("1 is non- negative");
      if(!isNonNeg.test(-1)) System.out.println("-1 is negative");
   }
}
```

Output

```
10 is even
9 is not even
1 is non-negative
-1 is negative
```

Explanations

```
(n) -> (n % 2)==0
```

- the type of n is not specified - inferred from the context
  - inferred from the parameter type of test( ) as defined by the NumericTest
- interface - int
- It is also possible to explicitly specify the type of a parameter in a lambda expression

```
(int n) -> (n % 2)==0
```

- with one parameter - not necessary to surround the parameter name with parentheses

```
n -> (n % 2)==0
```

Explanations (cont.)

- A functional interface reference can be used to execute any lambda expression that is compatible with it.
- the program defines two different lambda expressions that are compatible with the test( ) method of the functional interface NumericTest.
- The first - isEven
  - determines if a value is even
- The second - isNonNeg
  - checks if a value is non-negative.
- each lambda expression is compatible with test( ), each can be executed through a NumericTest reference

Example: a lambda expression with two parameters

- a lambda expression that takes two parameters.
- the lambda expression tests if one number is a factor of another

interface NumericalTest2, class LambdaDemo3

```
// Demonstrate a lambda expression that takes two parameters.
interface NumericTest2 {
   boolean test(int n, int d);
}

class LambdaDemo3 {
   public static void main(String args[])
   {
      // This lambda expression determines if one number is
      // a factor of another.
      NumericTest2 isFactor = (n, d) -> (n % d) == 0;
      if(isFactor.test(10, 2))
         System.out.println("2 is a factor of 10");
      if(!isFactor.test(10, 3))
         System.out.println("3 is not a factor of 10");
   }
}
```

Output

```
2 is a factor of 10
3 is not a factor of 10
```

An important point

- about multiple parameters in a lambda expression:
- If you need to explicitly declare the type of a parameter, then all of the parameters must have declared types
- For example, this is legal:

```
(int n, int d) -> (n % d) == 0
```

- But this is not:

```
(int n, d) -> (n % d) == 0
```

Block Lambda Expressions

- The body of the lambdas consisting of a single expressions are referred to as *expression bodies*
- lambdas that have expression bodies are sometimes called *expression lambdas.*

Block Lambda Expressions (cont.)

- Java supports a second type of lambda expression
- the code on the right side of the lambda operator consists of a block of code that can contain more than one statement.
- called a *block body.*
- Lambdas that have block bodies are sometimes referred to as *block lambdas*.

Note that

- use a return statement to return a value. This is necessary because a block lambda body does not represent a single expression

Example

- This example uses a block lambda to compute and return the factorial of an int value

interface NumericFunc, class BlockLambdaDemo

```
// A block lambda that computes the factorial of an int value.
interface NumericFunc {
   int func(int n);
}

class BlockLambdaDemo {
   public static void main(String args[])
   {
      // This block lambda computes the factorial of an int value.
      NumericFunc factorial = (n) -> {
         int result = 1;
         for(int i=1; i <= n; i++)
            result = i * result;
         return result;
      };

      System.out.println("The factoral of 3 is " + factorial.func(3));
      System.out.println("The factoral of 5 is " + factorial.func(5));
   }
}
```

Output

```
The factorial of 3 is 6
The factorial of 5 is 120
```

Exdplanations

- When a return statement occurs within a lambda expression, it simply causes a return from the lambda.
- It does not cause an enclosing method to return

Example

- reverses the characters in a string

class BlockLambdaDemo

```
// A block lambda that reverses the characters in a string.
interface StringFunc {
   String func(String n);
}

class BlockLambdaDemo2 {
   public static void main(String args[])
   {
      // This block lambda reverses the characters in a string.
      StringFunc reverse = (str) -> {
         String result = "";
         int i;
         for(i = str.length()-1; i >= 0; i--)
            result += str.charAt(i);
         return result;
      };
      System.out.println("Lambda reversed is " +
      reverse.func("Lambda"));
      System.out.println("Expression reversed is " +
      reverse.func("Expression"));
   }
}
```

Output

- Lambda reversed is adbmaL
- Expression reversed is noisserpxE

Passing Lambda Expressions as Arguments

- a lambda expression can be passed as an argument
  - a common use of lambdas
- the type of the parameter receiving the lambda expression argument must be of a functional interface type compatible with the lambda

Example

- illustrates the use of lambda expressions as an argument to a method

interface StringFunc, class LambdasAsArgumentsDemo

```
// Use lambda expressions as an argument to a method.
interface StringFunc {
   String func(String n);
}

class LambdasAsArgumentsDemo {

   // This method has a functional interface
   // as the type of its first parameter.
   // Thus, it can be passed a reference to
   // any instance of that interface,
   // including the instance created by a lambda expression.
   // The second parameter specifies the string to operate on.
   static String stringOp(StringFunc sf, String s) {
      return sf.func(s);
   }
```

class LambdasAsArgumentsDemo (cont.)

```
public static void main(String args[])
{
   String inStr = "Lambdas add power to Java";
   String outStr;
   System.out.println("Here is input string: " + inStr);
   // Here, a simple expression lambda
   // that uppercases a string
   // is passed to stringOp( ).
   outStr = stringOp((str) -> str.toUpperCase(), inStr);
   System.out.println("The string in uppercase: " + outStr);


   // This passes a block lambda that removes spaces.
   outStr = stringOp((str) -> {
                   String result = "";
                   int i;
                   for(i = 0; i < str.length(); i++)
                      if(str.charAt(i) != ' ')
                        result += str.charAt(i);
                   return result;
                }, inStr);

   System.out.println("The string with spaces removed: " + outStr);
```

class LambdasAsArgumentsDemo (cont.)

```
      // Of course, it is also possible
      // to pass a StringFunc instance
      // created by an earlier lambda expression.
      // For example, after this declaration executes,
      // reverse refers to an instance of StringFunc.
      StringFunc reverse = (str) -> {
         String result = "";
         int i;
         for(i = str.length()-1; i >= 0; i--)
            result += str.charAt(i);
         return result;
      };

      // Now, reverse can be passed
      // as the first parameter to stringOp()
      // since it refers to a StringFunc object.
      System.out.println("The string reversed: " +
                         stringOp(reverse, inStr));
   }
}
```

Output

```
Here is input string: Lambdas add power to Java
The string in uppercase: LAMBDAS ADD POWER TO JAVA
The string with spaces removed: LambdasaddpowertoJava
The string reversed: avaJ ot rewop dda sadbmaL
```

Lambda Expressions and Variable Capture

- Variables defined by the enclosing scope of a lambda expression are accessible within the lambda expression.
- For example, a lambda expression can use an instance or static variable defined by its enclosing class.
- A lambda expression also has access to this (both explicitly and implicitly),
- which refers to the invoking instance of the lambda expression’s enclosing class.
- Thus, a lambda expression can obtain or set the value of an instance or static variable
- and call a method defined by its enclosing class.

Lambda Expressions and Variable Capture (cont.)

- local variables used in lambda expression from its enclosing scope - *variable capture*
- local variables - *effectively final*.
  - whose value does not change after it is first assigned
- no need to declare such a variable as final, although doing so would not be an error.
- The this parameter of an enclosing scope is automatically effectively final
- and lambda expressions do not have a this of their own.

Example

- the difference between effectively final and mutable local variables

interface MyFunc, class VarCapture

```java
// An example of capturing a local variable from the enclosing scope.
interface MyFunc {
   int func(int n);
}

class VarCapture {
   public static void main(String args[])
   {
      // A local variable that can be captured.
      int num = 10;
      MyFunc myLambda = (n) -> {
         // This use of num is OK. It does not modify num.
         int v = num + n;
         // However, the following is illegal because it attempts
         // to modify the value of num.
         // num++;
         return v;
      };
      // The following line would also cause an error, because
      // it would remove the effectively final status from num.
      // num = 9;
   } // end main
} // end class
```

Explanations

- As the comments indicate, num is effectively final and can, therefore, be used inside
- myLambda.
- However, if num were to be modified, either inside the lambda or outside of it,
- num would lose its effectively final status.
- This would cause an error, and the program would not compile.

Explanations (cont.)

- a lambda expression can use and modify an instance variable from its invoking class.
- It just can’t use a local variable of its enclosing scope unless that variable is effectively final.

SUMMARY: Lambda Expressions

- Lambda expression
  - anonymous method
  - shorthand notation for implementing a functional interface.
- The type of a lambda is the type of the functional interface that the lambda implements.
- Can be used anywhere functional interfaces are expected.

SUMMARY: Lambda Expressions (Cont.)

- A lambda consists of a parameter list followed by the arrow token and a body, as in:
  - (*parameterList*) -> {*statements*}
- For example, the following lambda receives two `int`s and returns their sum:
  - (`int` x, `int` y) -> {`return` x + y;}
- This lambda's body is a statement block that may contain one or more statements enclosed in curly braces.
- A lambda's parameter types may be omitted, as in:
  - (x, y) -> {`return` x + y;}
- in which case, the parameter and return types are determined by the lambda's context.

SUMMARY: Lambda Expressions (Cont.)

- A lambda with a one-expression body can be written as:
  - `(x, y) -> x + y`
  - In this case, the expression's value is implicitly returned.
- When the parameter list contains only one parameter, the parentheses may be omitted, as in:
  - `value -> System.out.printf("%d ", value)`
- A lambda with an empty parameter list is defined with `()` to the left of the arrow token (`->`), as in:
  - `() -> System.out.println("Welcome to lambdas!")`
- There are also specialized shorthand forms of lambdas that are known as method references.