Chapter 5

# Defining Classes and Methods

- 5.1 Class and Method Definitions
- 5.2 Information Hiding and Encapsulation
- 5.3 Objects and Reference
- 5.4 GRAPHICS SUPPLEMENT (OPTIONAL)

# objectives

- Become familiar with the concepts of a class and of an object that instantiates the class
- Learn how to define classes in Java
- Learn to define and use methods (object actions) in Java
- Learn to create objects in Java
- Find out how parameters work in Java
- Learn about information hiding and encapsulation
- Become familiar with the notion of reference so that you can understand class variables and class parameters

# 5.1 Class and Method definition

- *Object – like automobiles, houses, employee records…*
- *Class* — definition of a kind of object
  - » Like an outline or plan for constructing specific objects
  - » see next slide or diagram in text (display 4.3)
- Example: an Automobile class
  - » Object that satisfies the `Automobile` definition
  - » ….. ***instantiates*** the `Automobile` class
- Class specifies what kind of data objects of that class have
  - » Each object has the same items but can have different
- Class specifies what methods each object will have
  - » All objects of the same class have the exact same methods

# Figure 5.1
# Class as an Outline

Class Definition

Objects that are [redacted] of the class

**Class Name:** Automobile
**Data:**
amount of fuel _____
speed _____
license plate _____
**Methods (actions):**
increaseSpeed:
**How:** Press on gas pedal.
stop:
**How:** Press on brake pedal.

**First Instantiation:**
**Object name:** patsCar
**amount of fuel:** 10 **gallons**
**speed:** 55 **miles per hour**
**license plate:** "135 XJK"

**Second Instantiation:**
**Object name:** suesCar
**amount of fuel:** 14 **gallons**
**speed:** 0 **miles per hour**
**license plate:** "SUES CAR"

**Third Instantiation:**
**Object name:** ronsCar
**amount of fuel:** 2 **gallons**
**speed:** 75 **miles per hour**
**license plate:** "351 WLF"

# Class Diagram

- Class Diagram
  - » UML Class Diagram
  - » the Java and UML Syntaxes for visibility

| Visibility | Java Syntax | UML Syntax |
|---|---|---|
| public | `public` | |
| protected | `protected` | |
| package | | |
| private | `private` | |

# Figure 5.2 A Class Outline as a UML Class Diagram

```
Automobile

- fuel: double
- speed: double
- license: String

+ accelerate(double pedalPressure): void
+ decelerate(double pedalPressure): void
```

# Class. Type.

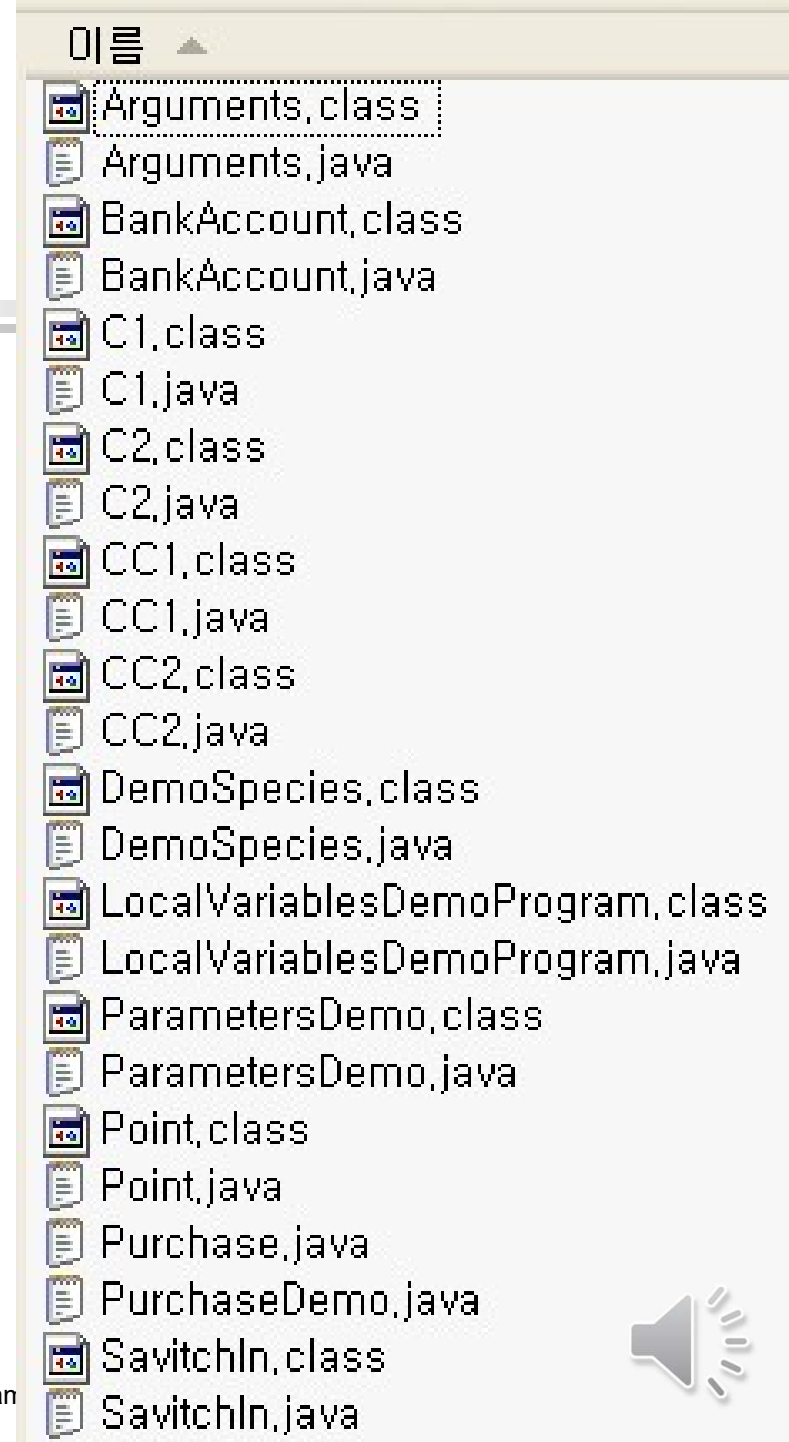
- A class
- The objects that instantiate the class
- Each object has a name
  - » the names are patsCar, suesCar, and ronsCar
- a class is a [redacted]
  - » object names(patsCar, suesCar, and ronsCar) would be <u>variables of type Automobile</u>.

# Class Files

- Each Java class definition should be a file
- Use the same name for the class and the file, except add ".java" to the file name
- Good programming practice:
  Start the class (and file) name a capital letter and capitalize inner words upper case
  - » e.g. MyClass.java for the class MyClass
- For now put all the classes you need to run a program in the same directory

# Class Files and Separate Compilation

- How a Java class definition is stored in a file
  - » Each Java class definition should be in a file by itself
  - » The name of the file should be the same as the name of class
  - » The file name should end in .java
- The compiled [illegible] for the class will be stored in a file of the same name, but ending in .class



# Listing 5.1 A Class Definition -- SpeciesFirstTry.java

```
// We will give a better version of this class later in this chapter
import java.util.Scanner;

public class SpeciesFirstTry
{
   public String name;
   public int population; // later in this chapter, public ==> private
   public double growthRate;

   public void readInput( )
   {
      Scanner keyboard = new Scanner(System.in);
      System.out.println("What is the species' name?");
      name = keyboard.nextLine( );

      System.out.println("What is the population of the species?");
      population = keyboard.nextInt( );

      System.out.println("Enter growth rate (% increase per year):");
      growthRate = keyboard.nextDouble( );
   }
```

```
public void writeOutput( )
   {
       System.out.println("Name = " + name);
       System.out.println("Population = " + population);
       System.out.println("Growth rate = " + growthRate + "%");
   }

   public int populationIn10( )
   {
      double populationAmount = population;
      int count = 10;
      while ((count > 0) && (populationAmount > 0))
      {
         populationAmount = (populationAmount +
                        (growthRate/100) * populationAmount);
         count--;
      }
      if (populationAmount > 0)
         return (int)populationAmount;  // type cast
      else
         return 0;
   }
}
```

- The class name ➔
- The three data ➔
- Three Methods ➔

- SpeciesFirstTry
- Name, population size, a growth rate
- readInput, writeOutput and populationIn10

# Objects

- Objects are variables that are named [redacted] of a class
  - » the class is their [redacted]
  - Objects have both [redacted] and [redacted]
- Both the data items and methods of a class are [redacted] of the object
- Data items are also called [redacted] or [redacted]
- [redacted] a method means to *call* the method, i.e. execute the method
  - » Syntax for invoking an object's method: the dot operator
    `object_Variable_Name.method()`
  - » `object_Variable_Name` is the *calling object*

# Variables

- `SpeciesFirstTry` class has three instance variables: `name`, `population`, and `growthRate`:

```
public String name;

public int population;

public double growthRate;
```

- means that there are no restrictions on how these instance variables are used.
- Later we'll see that these should be declared `private` instead of `public`.

# ( ) Objects

Syntax:

*class_Name instance_Name = class_Name();*

- Note the keyword

- Ex) the text defines a class named `SpeciesFirstTry`

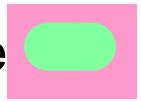
```
//instantiate an object of this class
    SpeciesFirstTry speciesOfTheMonth =
                        new SpeciesFirstTry();
     ; create an object of type SepciesFirstTry
     ; attaches the name speciesOfTheMonth to this object
```

# Listing 5.2 using Classes and Methods - SpeciesFirstTryDemo.java

```
// Listing 5.2 Using Classes and Methods
public class SpeciesFirstTryDemo
{
   public static void main(String[] args)
   {  SpeciesFirstTry speciesOfTheMonth = new SpeciesFirstTry( );
      int futurePopulation;

      System.out.println("Enter data on the Species of the Month:");
      speciesOfTheMonth.readInput( );              //
      speciesOfTheMonth.writeOutput();             //

      futurePopulation = speciesOfTheMonth.populationIn10( );
      System.out.println("In ten years the population will be "
                                  + futurePopulation);
      speciesOfTheMonth.name = "Klingon ox";
      speciesOfTheMonth.population = 10;
      speciesOfTheMonth.growthRate = 15;
      System.out.println("The new Species of the Month:");
      speciesOfTheMonth.writeOutput( );    //
      System.out.println("In ten years the population will be "
                     + speciesOfTheMonth.populationIn10( ));
   }
}
```

C:₩WINDOWS₩system32₩cmd.exe

```
Enter data on the Species of the Month:
What is the species' name?
Ferengie fur ball
What is the population of the species?
1000
Enter growth rate (percent increase per year):
-20.5
Name = Ferengie fur ball
Population = 1000
Growth rate = -20.5%
In ten years the population will be 100
The new Species of the Month:
Name = Klingon ox
Population = 10
Growth rate = 15.0%
In ten years the population will be 40
계속하려면 아무 키나 누르십시오 . . .
```

- Public instance variables can be accessed using the  operator:

```
SpeciesOfTheMonth.name = "Klingon ox";
```

- Each instance variables has a type
  - » The instance variable name is of type String
    **speciesOfTheMonth.name = "Klingon ox";**

- Each object of type SpeciesFirstTry has <u>its own three instance variables.</u>

**SpeciesFirstTry speciesOfTheMonth = new SpeciesFirstTry( );**
**SpeciesFirstTry speciesOfLastMonth = new SpeciesFirstTry( );**

  - » <u>Different instance variable..</u>

# Using Methods

- Methods are actions that an object can perform.
- To use a method you [redacted] or ***call*** it.

  Example of a method call:

```
speciesOfTheMonth.writeOutput()
```

***calling object***— tells which object will do the action

method name—tells which action the object will perform

parameter list in parentheses—parameters give info to the method, but in this example there are no parameters

- Two basic kinds of methods:
  - » methods that return [redacted] value
  - » [redacted] methods that do some action other than returning a value

# Calling object

- For certain special methods(static methods..) , you can use the [redacted] of the class instead of using an[redacted]f the class.
  - » ex) Class name .. SavitchIn
    - – SavitchIn.readLineInt()

```
public static String readLine( )
  {
      char nextChar;
      String result = "";
      boolean done = false;
     .............
 public static String readLineWord( )
  {
      String inputString = null,
            result = null;
      boolean done = false;

      while(!done)
      {
```

# Return Type of Methods

- All methods require that the return type be specified
- Return types may be:
  - » a data type, such as `char`, `int`, `double`, etc.
  - » a , such as `String`, `SpeciesFirstTry`, etc.
  - » `void` if no value is returned
- You can use a method anyplace where it is legal to use its return type, for example the `readLineInt()` method of `SavitchIn` returns an integer, so this is legal:

```
int next = SavitchIn.readLineInt();
```

```
public static int readLineInt( )
  {
     String inputString = null;
     int number = -9999;//To keep the compiler happy.
                        //Designed to look like a garbage value.
     boolean done = false;
```

# `void` Method Example

- The definition of the `writeOutput` method of `SpeciesFirstTry`:

```
public void writeOutput()
{
  System.out.println("Name = " + name);
  System.out.println("Population = " + population);
  System.out.println("Growth = " + growthRate + "%");
}
```

- Assuming instance variables `name`, `population`, and `growthRate` have been defined and assigned values, this method performs an action (writes values to the screen) but does not return a value

# Return Statement

- Methods that return a value must execute a `return` statement that includes the value to return

- For example:

```
public int count = 0;

public int getCount()
{
    return count;
}
```

# Return Statement

- Return expression;
  - » the Expression can be any expression that produces a value of the type specified in the heading of the method definition.

```
public int populationIn10( )
  {
    double populationAmount = population;
    [illegible] = 10;
    ……..
   if (populationAmount > 0)
      return populationAmount;
    else
      return 0;
  }
```

```
D:₩My Documents₩@@@@@@jv₩ch04₩SpeciesFirstTry.java:44: possible loss
found   : double
required: int
        return populationAmount;   / type cast
               ^

1 error

Tool completed with exit code 1
```

# Use of return in void Methods

- To end a method invocation early, such as when the method discovers some sort of problem

```
public void showLandPortion()
  {
        if (population == 0)
        {
                System.out.println("population is zero.");
                return; // Ends here to aviod division by zero.
        }
        double fraction;
        ..
        ..
        ..
  }
```

# The **`main`** Method

- A program written to solve a problem (rather than define an object) is written as a class with one method, `main`
- Invoking the class name invokes the `main` method
- See the text: `SpeciesFirstTryDemo`
- Note the basic structure:

```
public class SpeciesFirstTryDemo
{
   public static void main(String[] args)
   {
      <statements that define the main method>
   }
}
```

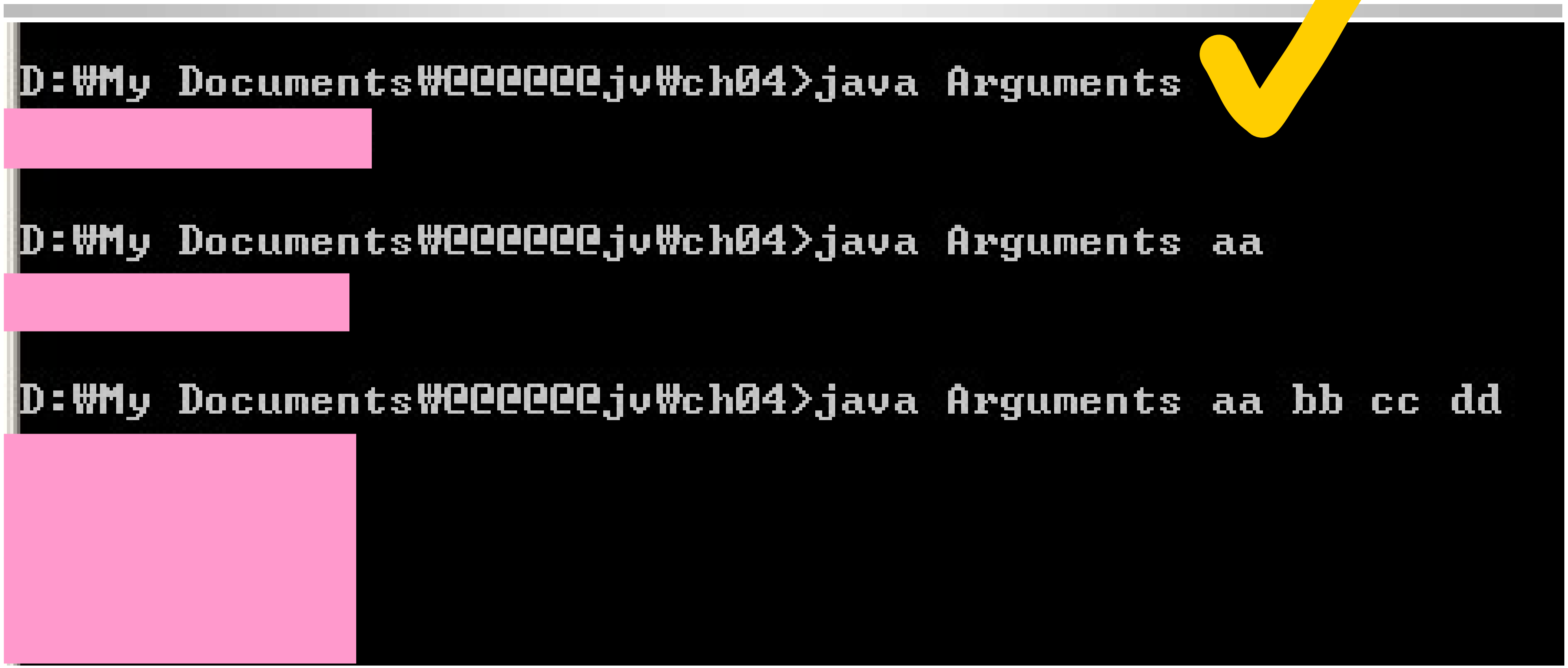
# Static & public of main()

- main 메소드
  - » 자바에서 프로그램을 시작하는 메소드
- Static
  - » 그렇기 때문에 main 메소드는 그 어떤 클래스가 로딩되기 전에 그 어떤 오브젝트가 만들어 지기 전에도 쓸수 있어야 하기 때문에 static 해야함
  - » 즉 되기전에 써야 하기 때문에 static 해야함 .
- Public
  - » 외부에서 써야 하기 때문에 public 해야함

# Arguments - The `main` Method

- Examples : Arguments.java

```
/**
 *  This program prints out the command line arguments
 */
public class Arguments {

  public static void main(String[] args) {
    if (args.length > 0) {
      for (int i = 0; i < args.length; i++) {
        System.out.println("args[" + i + "]: " + args[i]);
      }
    } else {
      System.out.println("No arguments.");
    }
  }

}
```



```
D:₩My Documents₩@@@@@@jv₩ch04>java Arguments
[redacted]

D:₩My Documents₩@@@@@@jv₩ch04>java Arguments aa
[redacted]

D:₩My Documents₩@@@@@@jv₩ch04>java Arguments aa bb cc dd
[redacted]
```

# The Reserved Word **`this`**

- The word `this` has a special meaning for objects
  - » a [redacted] word
    - – you should not use it as an identifier for a variable, class or method
    - – Ex) `int`, `char`, `main`, etc.
- `this` stands for the name of the [redacted] object
- Java allows you to omit `this` .
  - » It is automatically understood that an instance variable name without the keyword `this` refers to the calling object

# Example Using **this**

- Using the same example as for the `void` method, but including the keyword `this`:

```
public void writeOutput()
{
    System.out.println("Name = " + this.name);
    System.out.println("Population = " +
   this.population);
    System.out.println("Growth rate = " +
   this.growthRate + "%");
}
```

- `this` refers to the name of the calling object that invoked the `writeOutput` method

# Modify Listing *.* for this

- SpeciesFirstTryThis.java
- SpeciesFirstTryDemoThis.java

```
speciesOfTheMonth.writeOutput();
```

```
 public void writeOutput( )
  {
      System.out.println("Name = " +  speciesOfTheMonth.name);
      System.out.println("Population = " +
speciesOfTheMonth.population);
      System.out.println("Growth rate = " +
speciesOfTheMonth.growthRate + "%");
  }
```

```
public void writeOutput( )
  {
      System.out.println("Name = " + this.name);
      System.out.println("Population = " + this.population);
      System.out.println("Growth rate = " + this.growthRate + "%");
  }
```

# Modify Listing *.* for this

```
speciesOfTheMonth.name = "Klingon ox";
speciesOfTheMonth.population = 10;
speciesOfTheMonth.growthRate = 15;
```

```
//speciesOfTheMonth.name = "Klingon ox";
this.name = "Klingon ox";
speciesOfTheMonth.population = 10;
speciesOfTheMonth.growthRate = 15;
```

# This

- This
  - » 1) Passing this as a [redacted]
  - » 2) Accessing [redacted] Fields

# Shadowed Fields

```
public class MyClass {
   int var; //                    variable

   void method1() {
      float var; //                    shadows the instance variable
      // ...
   }

   void method2(int var) {
      //                    also shadows the instance variable
      // ...
   }

}
```

This.var : hidden variable can be accessed as this.var

# *Local* Variables and *Blocks*

- A *block* (a [redacted] *statement)*
  - » the set of statements between a pair of matching braces (curly brackets)
- A variable declared inside a block
  - » known only inside that block
  - » *local* to the block, therefore it is called *a local variable*
  - » when the block finishes executing, local variables disappear
  - » references to it outside the block cause a compile error

# *Local* Variables and *Blocks*

- Some programming languages (e.g. C and C++) allow the variable name to be reused outside the local block
  - » it is confusing and not recommended, nevertheless, it is allowed
- However, a variable name in Java can be declared only *once for a method*
  - » although the variable does not exist outside the block, other blocks in the same method cannot reuse the variable's name
- You cannot have two variables with the same name inside of a single method definition.

# When and Where to Declare Variables

- Declaring variables outside all blocks but within the method definition makes them available within all the blocks

<u>Good programming Practice:</u>

- declare variables just before you use them
- initialize variables when you declare them
- do not declare variables inside loops
  - » it takes time during execution to create and destroy variables, so it is better to do it just once for loops)
- it is ok to declare loop counters in the *`Initialization`* field of `for` loops, e.g.
  `for(int i=0; i <10; i++)…`
  - » the *`Initialization`* field executes only once, when the `for` loop is first entered

# Declaring variables in a for Statement

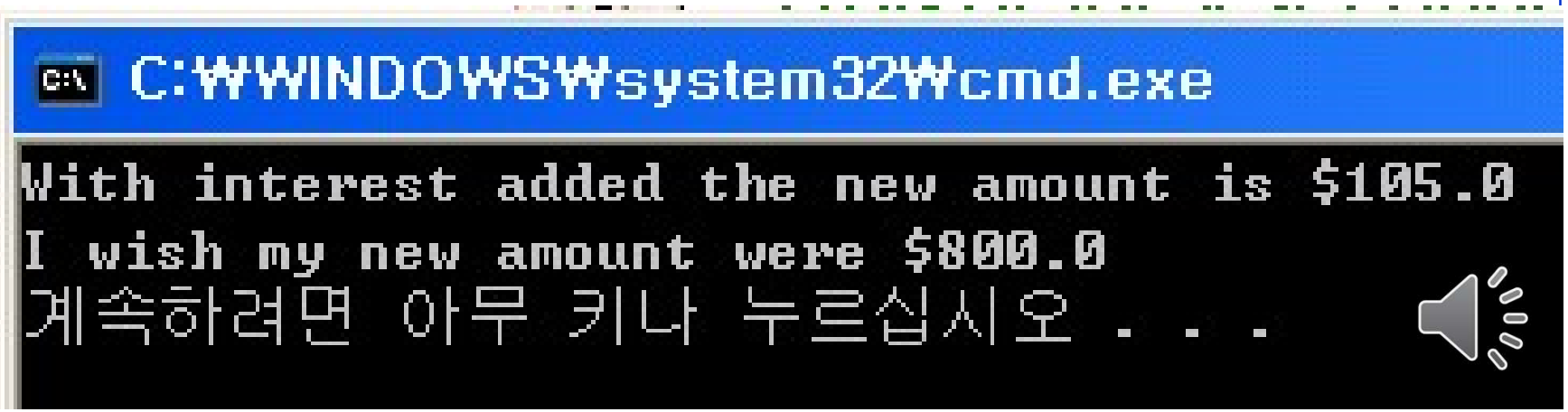
```
public class VarInFor
{
    public static void main(String[] args)
    {
        int sum = 0;
        for (int n=1; n<=10; n++)
            sum = sum + n*n;
        System.out.println(n);

    } // ???
}
```

# Listing 5.3 Local Variable -

BankAccount.java, LocalVariablesDemoProgram.java

```
// Listing 5.3 Local Variables

/**
 This class is used in the program LocalVariablesDemoProgram.
*/
public class BankAccount          //BankAccount.java
{
    public double amount;
    public double rate;

    public void showNewBalance( )
    {

        // newAmount
        double newAmount = amount + (rate/100.0)*amount;
        System.out.println("With interest added the new amount is $"
                                + newAmount);
    }
}
```

```
//Lisitng 5.3 Local Variable

/**
 A toy program to illustrate how local variables behave.
*/

public class LocalVariablesDemoProgram
{
   public static void main(String[] args)
   {
      BankAccount myAccount = new BankAccount( );
      myAccount.amount = 100.00;
      myAccount.rate = 5;

                        // newAmount
      double newAmount = 800.00;
      myAccount.showNewBalance( );
                        // does not change newAmount in main.
      System.out.println("I wish my new amount were $" + newAmount);
   }
}
```



# Passing Values to a Method: Parameters

- Formal parameter(??)
- Actual parameter(??)

# Passing Values to a Method: Parameters

- Some methods can be more flexible (therefore useful) if we pass them input values
- Input values for methods are called *passed values* or *parameters*
- Parameters and their data types must be specified inside the parentheses of the heading in the method
  - » these are called *formal* parameters
- The calling object must put values of the same data type, in the same order, inside th of the method invocation
  - » these are called *arguments*, or *actual* parameters

# Parameter Passing Example

```
//Definition of method to double an integer
public int doubleValue(int numberIn)
{
   return 2 * numberIn;
}
//Invocation of the method... somewhere in main...
...
int next = Scanner.nextInt();
System.out.println("Twice next = " +
doubleValue(next));
```

- What is the formal parameter in the method definition?
  - »
- What is the argument in the method invocation?
  - »

# (??): Data Types as Parameters

- When the method is called, the *value* of each argument is (assigned) to its corresponding formal parameter
- The number of arguments must be the same as the number of formal parameters
- The data types of the arguments must be the same as the formal parameters and in the same order
- Formal parameters are to the values passed
- Formal parameters are to their method
- Variables used as arguments cannot be changed by the method
  - » the method only gets a copy of the variable's value

# Class parameter

- Parameters of a class type behave differently from the parameters of a primitive type.

# Pass by Value – 1) Primitive type as a parameter – c1.java, cc1.java

```
//
public class C1
{
        public void inc(int i) {i++;}
}
```

```
//
public class CC1  {

   public static void main(String[] args)
   {
      C1 c1= new C1();
      int k = 1;
      c1.inc(k);

      System.out.println("k= " + k );
   }
}
```

# – 2) class type as a parameter – c2.java, cc2.java,point.java

```
//
public class C2
{
        public void pointInc(Point p)
        {
                p.x++; p.y++;
        }
}
```

```
//
public class CC2  {

    public static void main(String[] args)
    {
        C2 c2= new C2();
        Point p = new Point(10.0, 10.0);

        c2.pointInc(p);

        System.out.println("("+p.x+"," +p.y+")");
    }
}
```

끝