

NCP2103: Object-Oriented Programming (Java Programming)

Advanced Object Concepts Module 4

Errol John M. Antonio

Assistant Professor
Computer Engineering Department
University of the East – Manila Campus

Copyright belongs to Farrell, J. (2016). Java Programming. 8th Edition. Course Technology, Cengage Learning.



University of the East
Manila Campus

NCp2103: Object-Oriented Programming
erroljohn.antonio@ue.edu.ph

Learning Objectives

At the end of this module, students shall be able to:

- Understand blocks and scope
- Overload a method
- Avoid ambiguity
- Use constructors with parameters
- Use the `this` reference
- Use static variables
- Use constant fields



Learning Objectives

At the end of this module, students shall be able to:

- Use automatically imported, prewritten constants and methods
- Use composition
- Use nested and inner classes



Module Outline

- I. Understanding Blocks and Scope
- II. Overloading a Method
- III. Learning About Ambiguity
- IV. Using Constructors with Parameters
- V. Learning About the this Reference
- VI. Using static Variables
- VII. Using Automatically Imported,
Prewritten Constants and Methods
- VIII. Understanding Composition



I: Understanding Blocks and Scope

▪ Blocks

- Use opening and closing curly braces
- Can exist entirely within another block or entirely outside and separate from another block
- Cannot overlap
- Types
 - Outside or **outer blocks**
 - Inside or **inner blocks**
 - **Nested blocks**



Understanding Blocks and Scope

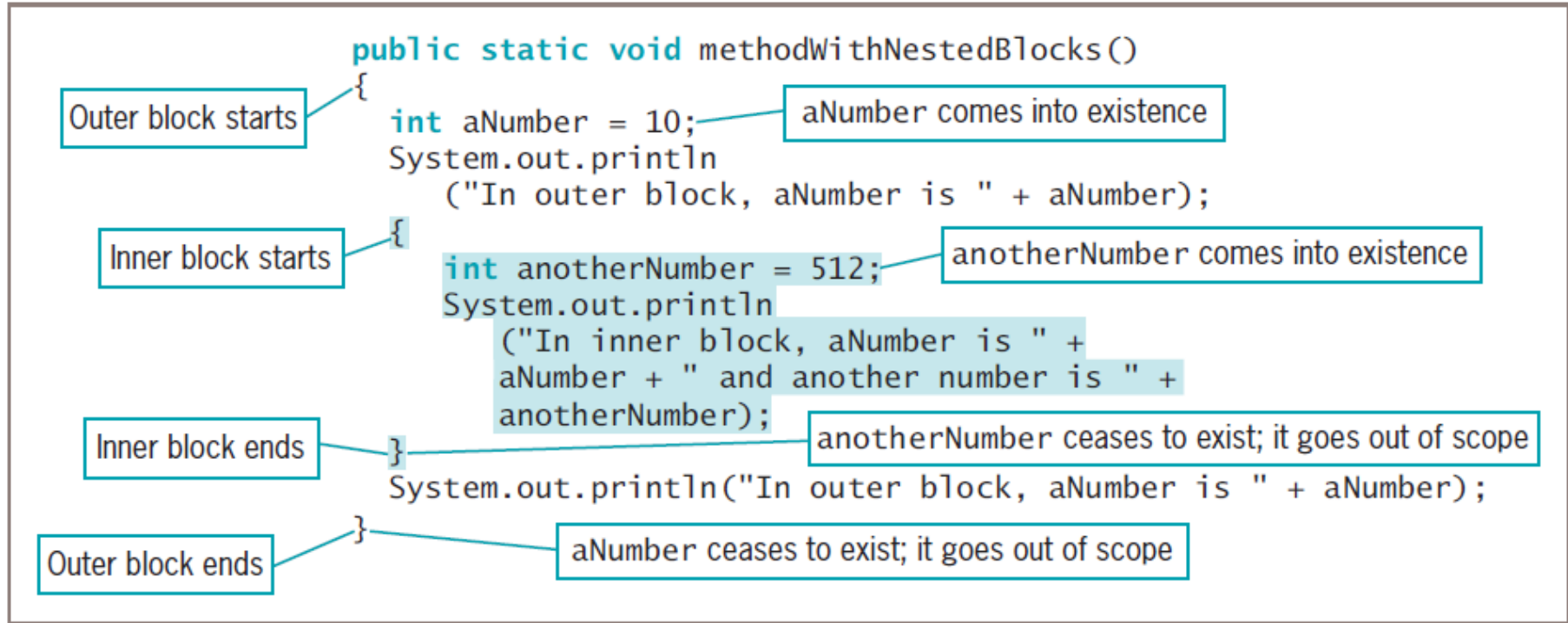


Figure 4-1 A method with nested blocks



Understanding Blocks and Scope



```
Command Prompt

C:\Java>java TestMethodWithNestedBlocks
In outer block, aNumber is 10
In inner block, aNumber is 10 and another number is 512
In outer block, aNumber is 10

C:\Java>_
```

Figure 4-2 Output produced by application that uses `methodWithNestedBlocks()`



Understanding Blocks and Scope

▪ Scope

- Portion of program within which you can refer to a variable
- Comes into scope
 - Variable comes into existence
- Goes out of scope
 - Variable ceases to exist



Understanding Blocks and Scope

▪ Scope

- Portion of program within which you can refer to a variable
- Comes into scope
 - Variable comes into existence
- Goes out of scope
 - Variable ceases to exist



Understanding Blocks and Scope

```
public static void methodWithInvalidStatements()  
{  
    aNumber = 75; Illegal statement; this variable has not been declared yet  
    int aNumber = 22;  
    aNumber = 6;  
    anotherNumber = 489; Illegal statement; this variable has not been declared yet  
    {  
        anotherNumber = 165; Illegal statement; this variable still has not been declared  
        int anotherNumber = 99;  
        anotherNumber = 2;  
    }  
    aNumber = 50; Illegal statement; this variable was declared in the inner block  
and has gone out of scope here  
    anotherNumber = 34;  
}  
aNumber = 29; Illegal statement; this variable has gone out of scope
```

Figure 4-3 The `methodWithInvalidStatements()` method



Understanding Blocks and Scope

- Redeclare variable
 - Cannot declare same variable name more than once within block
 - Illegal action



Understanding Blocks and Scope

Don't declare blocks for no reason. A new block starts here only to demonstrate scope.

```
public static void twoDeclarations()
{
    {
        int someVar = 7;
        System.out.println(someVar);
    }
    {
        int someVar = 845;
        System.out.println(someVar);
    }
}
```

This variable will go out of scope at the next closing curly brace.

This variable is totally different from the one in the previous block even though their identifiers are the same.

Figure 4-4 The `twoDeclarations()` method



Understanding Blocks and Scope

```
public static void invalidRedeclarationMethod()
```

```
{
```

```
    int aValue = 35;
```

```
    int aValue = 44;
```

```
{
```

```
    int anotherValue = 0;
```

```
    int aValue = 10;
```

```
}
```

```
}
```

Invalid redeclaration of aValue because it is in the same block as the first declaration

Invalid redeclaration of aValue; even though this is a new block, this block is inside the first block

Figure 4-5 The `invalidRedeclarationMethod()`



Understanding Blocks and Scope

- Override
 - Variable's name within method in which it is declared
 - Takes precedence over any other variable with same name in another method
 - Locally declared variables
 - Always mask or hide other variables with same name elsewhere in class



Understanding Blocks and Scope

```

public class OverridingVariable
{
    public static void main(String[] args)
    {
        int aNumber = 10;
        System.out.println("In main(), aNumber is " + aNumber);
        firstMethod();
        System.out.println("Back in main(), aNumber is " + aNumber);
        secondMethod(aNumber);
        System.out.println("Back in main() again, aNumber is " + aNumber);
    }
    public static void firstMethod()
    {
        int aNumber = 77;
        System.out.println("In firstMethod(), aNumber is "
            + aNumber);
    }
    public static void secondMethod(int aNumber)
    {
        System.out.println("In secondMethod(), at first "
            + aNumber);
        aNumber = 862;
        System.out.println("In secondMethod(), after an assignment "
            + aNumber);
    }
}

```

aNumber is declared in main().

Whenever aNumber is used in main(), it retains its value of 10.

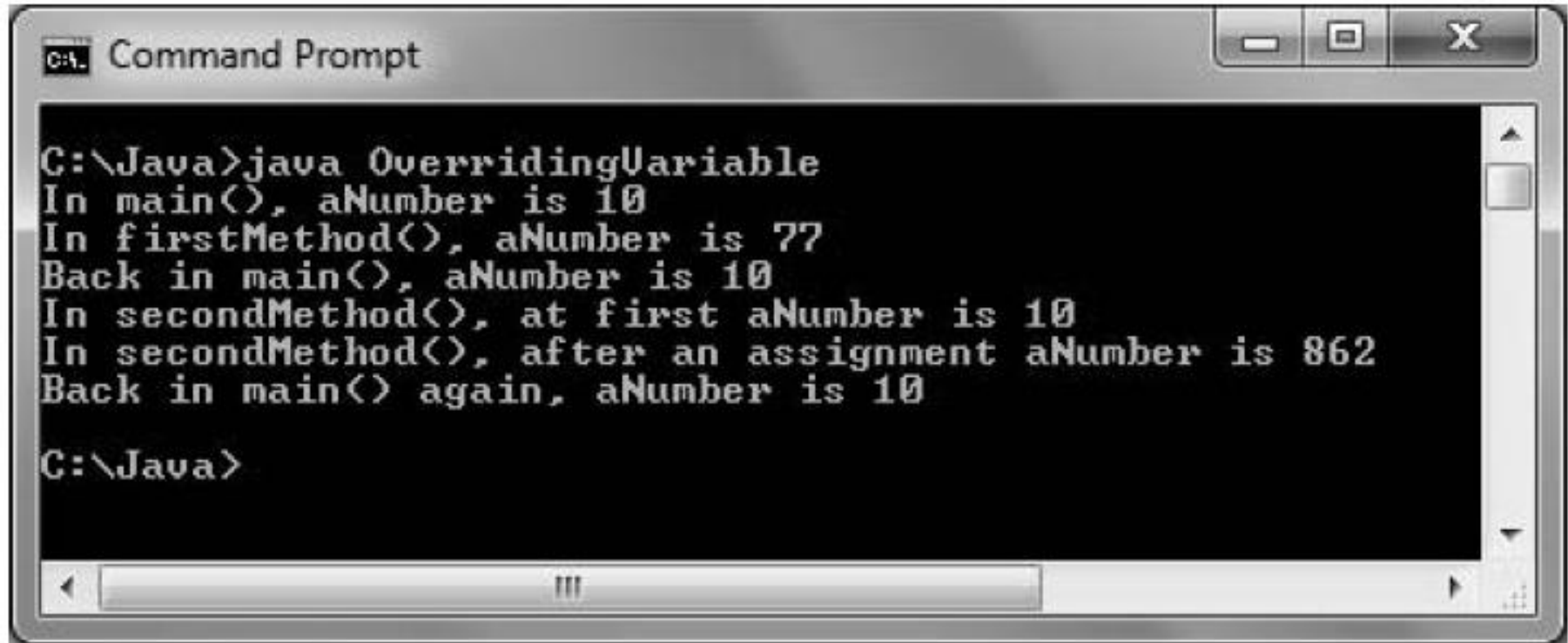
This aNumber resides at a different memory address from the one in main(). It is declared locally in this method.

This aNumber also resides at a different memory address from the one in main(). It is declared locally in this method.

Figure 4-6 The OverridingVariable class



Understanding Blocks and Scope



```
C:\Java>java OverridingVariable
In main(), aNumber is 10
In firstMethod(), aNumber is 77
Back in main(), aNumber is 10
In secondMethod(), at first aNumber is 10
In secondMethod(), after an assignment aNumber is 862
Back in main() again, aNumber is 10

C:\Java>
```

Figure 4-7 Output of the OverridingVariable application



TWO TRUTHS AND A LIE

1. A variable ceases to exist, or goes out of scope, at the end of the block in which it is declared.
2. You cannot declare the same variable name more than once within a block, even if a block contains other blocks.
3. A class's instance variables override locally declared variables with the same names that are declared within the class's methods.



II: Overloading a Method

▪ Overloading

- Using one term to indicate diverse meanings
- Writing multiple methods with same name but with different arguments
- Compiler understands meaning based on arguments used with method call
- Convenient for programmers to use one reasonable name



Overloading a Method

```
public static void calculateInterest(double bal, double rate)
{
    double interest;
    interest = bal * rate;
    System.out.println("Simple interest on $" + bal +
        " at " + rate + "% rate is " + interest);
}
```

Figure 4-11 The calculateInterest() method with two double parameters



Overloading a Method

Notice the data type
for rate.

```
public static void calculateInterest(double bal, int rate)
{
    double interest, rateAsPercent;
    rateAsPercent = rate / 100.0;
    interest = bal * rateAsPercent;
    System.out.println("Simple interest on $" +
        bal + " at " + rate + "% rate is " +
        interest);
}
```

Dividing by 100.0 converts rate
to its percent equivalent.

Figure 4-13 The `calculateInterest()` method with a `double` parameter and an `int` parameter



Overloading a Method

- When an application contains just one version of a method, you can call the method using a parameter of the correct data type or one that can be promoted to the correct data type.

```
public static void simpleMethod(double d)
{
    System.out.println("Method receives double parameter");
}
```

Figure 4-14 The `simpleMethod()` method with a `double` parameter



Overloading a Method

- When an application contains multiple versions of a method.

```
public class CallSimpleMethod
{
    public static void main(String[] args)
    {
        double doubleValue = 45.67;
        int intValue = 17;
        simpleMethod(doubleValue);
        simpleMethod(intValue);
    }
    public static void simpleMethod(double d)
    {
        System.out.println("Method receives double parameter");
    }
}
```

Either a double or an int can be sent to a method that accepts a double.

Figure 4-15 The CallSimpleMethod application that calls simpleMethod() with a double and an int



```

public static void calculateInterest(double bal, double rate)
{
    double interest;
    interest = bal * rate;
    System.out.println("Simple interest on $" + bal +
        " at " + rate + "% rate is " + interest);
}

```

Figure 4-12 The calculateInterest() method with two double parameters

```

public static void calculateInterest(double bal, int rate)
{
    double interest, rateAsPercent;
    rateAsPercent = rate / 100.0;
    interest = bal * rateAsPercent;
    System.out.println("Simple interest on $" +
        bal + " at " + rate + "% rate is " +
        interest);
}

```

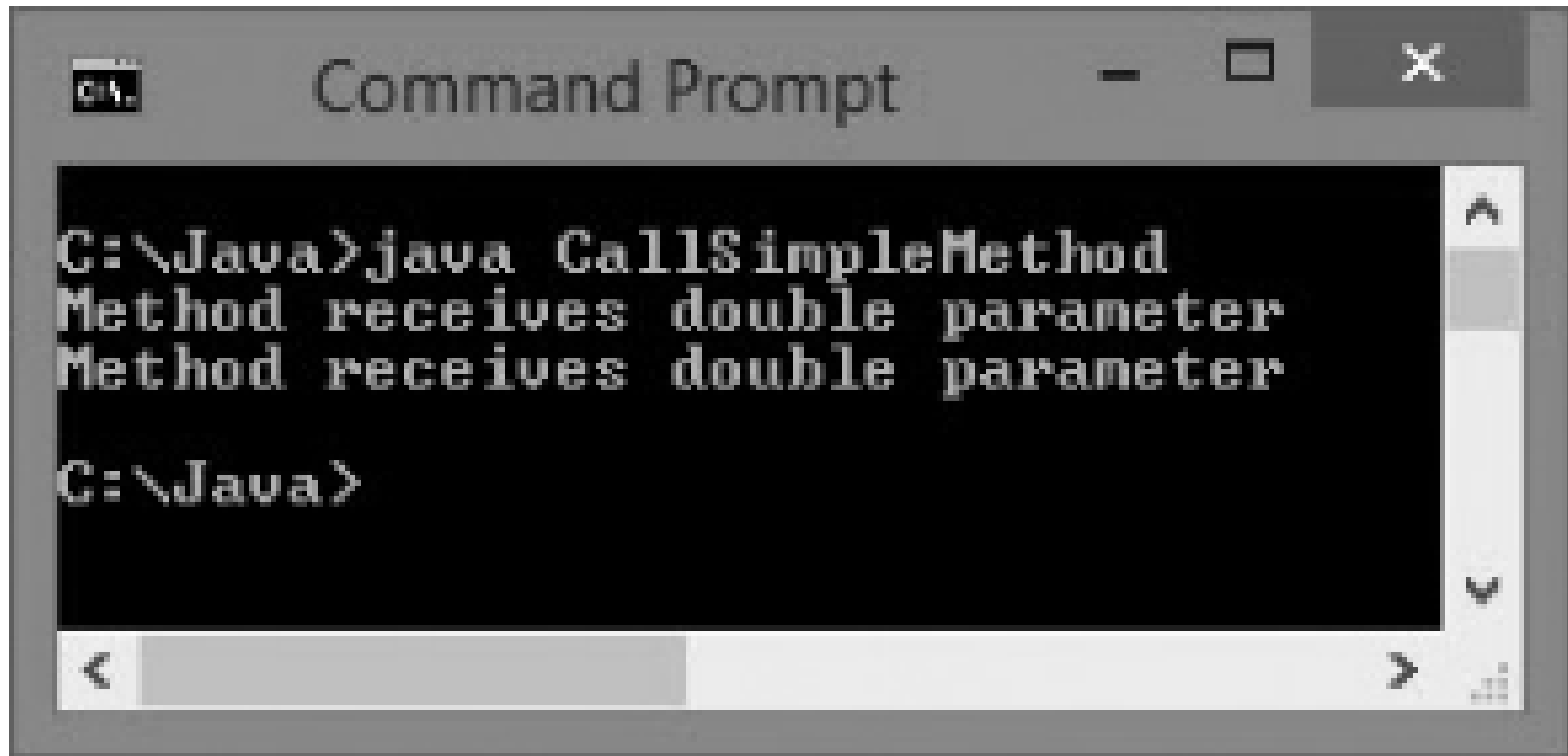
Notice the data type for rate.

Dividing by 100.0 converts rate to its percent equivalent.

Figure 4-13 The calculateInterest() method with a double parameter and an int parameter



Overloading a Method



```
C:\Java>java CallSimpleMethod
Method receives double parameter
Method receives double parameter
C:\Java>
```

Figure 4-16 Output of the `CallSimpleMethod` application



TWO TRUTHS AND A LIE

1. When you overload Java methods, you write multiple methods with a shared name.
2. When you overload Java methods, the methods are called using different arguments.
3. Instead of overloading methods, it is preferable to write methods with unique identifiers.



III: Learning About Ambiguity

- If application contains just one version of method:
 - Call method using parameter of correct data type; or
 - one that can be promoted to correct data type
- Ambiguous situation
 - When methods overloaded
 - May create situation in which compiler cannot determine which method to use



Learning About Ambiguity

- **Overload methods**

- Correctly provide different parameter lists for methods with same name

- **Illegal methods**

- Methods with identical names that have identical argument lists but different return types



Learning About Ambiguity

```
public static void computeBalance(double deposit)
```

```
public static void computeBalance(double withdrawal)
```

```
public static void calculateInterest(int bal, double rate)
```

```
public static void calculateInterest(double bal, int rate)
```



Learning About Ambiguity

- An overloaded method is not ambiguous on its own—it only becomes ambiguous if you create an ambiguous situation.



TWO TRUTHS AND A LIE

When it is part of the same program as
`void myMethod(int age, String name),`
the following method would be ambiguous:

1. `void myMethod(String name, int age)`
2. `String myMethod(int zipCode, String address)`
3. `void myMethod(int x, String y)`



IV: An Introduction to Using Constructors

- Default constructor provides specific initial values to object's data fields
 - Numeric fields
 - Set to 0 (zero)
 - Character fields
 - Set to Unicode `'\u0000'`
 - Boolean fields
 - Set to `false`
 - Nonprimitive object fields
 - Set to `null`



An Introduction to Using Constructors

```
Employee chauffeur = new Employee();
```

- Actually calling method named `Employee()`

- Default constructor

- Requires no arguments
- Created automatically by Java compiler
 - For any class
 - Whenever you do not write constructor



An Introduction to Using Constructors

```
public Employee()  
{  
    empSalary = 300.00;  
}
```

Figure 3-24 The Employee class constructor



TWO TRUTHS AND A LIE

1. In Java, you cannot write a default constructor; it must be supplied for you automatically.
2. The automatically supplied default constructor sets all numeric fields to 0, character fields to Unicode '\u0000', Boolean fields to false, and fields that are object references to null.
3. When you write a constructor, it must have the same name as the class it constructs, and it cannot have a return type.



IV: Using Constructors with Parameters

- Java automatically provides constructor method
 - When class-created default constructors do not require parameters
- Write your own constructor method
 - Ensures that fields within classes are initialized to appropriate default values
 - Constructors can receive parameters
 - Used for initialization purposes



Using Constructors with Parameters

- Write constructor for class
 - No longer receive automatically provided default constructor
- If class's only constructor requires argument
 - Must provide argument for every object of class

*(Source Files: Constructor.java and
ConstructorParameter.java)*



Overloading Constructors

- Use constructor parameters
 - To initialize field values
 - Or any other purpose
- If constructor parameter lists differ
 - No ambiguity about which constructor method to call



Overloading Constructors

```
public class Employee
{
    private int empNum;
    Employee(int num)
    {
        empNum = num;
    }
    Employee()
    {
        empNum = 999;
    }
}
```

Figure 4-20 The Employee class that contains two constructors



TWO TRUTHS AND A LIE

1. A default constructor is one that takes arguments.
2. When you write a constructor, it can be written to receive parameters or not.
3. If a class's only constructor requires an argument, you must provide an argument for every object of the class that you create.



V: Learning About the `this` Reference

- Instantiate object from class
 - Memory reserved for each instance field in class
 - Not necessary to store separate copy of each variable and method for each instantiation of class
- In Java
 - One copy of each method in class stored
 - All instantiated objects can use one copy



Learning About the `this` Reference

- Reference
 - Object's memory address
 - Implicit
 - Automatically understood without actually being written



Learning About the `this` Reference

- **`this`** reference
 - Reference to object
 - Passed to any object's non-static class method
 - Reserved word in Java
 - Don't need to use `this` reference in methods you write
 - In most situations

(Source Files: `ThisReference.java` and `ThisReferenceTest.java`)



Learning About the `this` Reference

```
public int getEmpNum()  
{  
    return empNum;  
}
```

The `this` reference is sent into this nonstatic method as a parameter automatically; you do not (and cannot) write code for it. You do not need to use `this` with `empNum`.

```
public int getEmpNum()  
{  
    return this.empNum;  
}
```

However, you *can* explicitly use the `this` reference with `empNum`. The two methods in this figure operate identically.

Figure 4-24 Two versions of the `getEmpNum()` method, with and without an explicit `this` reference



Learning About the `this` Reference

```
public class Student
{
    private int stuNum;
    private double gpa;
    public Student (int stuNum, double gpa)
    {
        stuNum = stuNum;
        gpa = gpa;
    }
    public void showStudent()
    {
        System.out.println("Student #" + stuNum +
            " gpa is " + gpa);
    }
}
```

Don't Do It

All four variables used in these two statements are the local versions declared in the method's parameter list. The fields are never accessed because the local variables shadow the fields. These two assignment statements accomplish nothing.

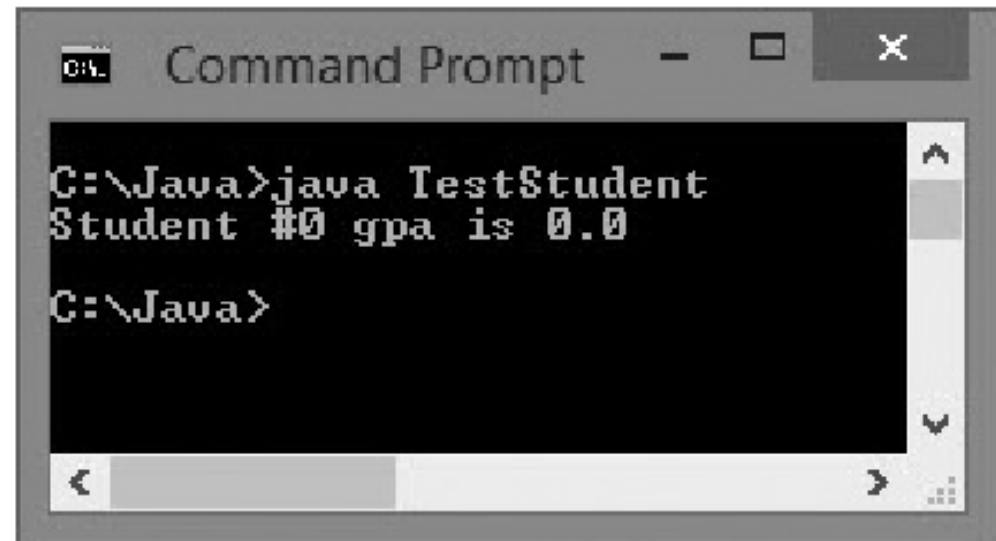
Figure 4-25 A Student class whose constructor does not work



Learning About the `this` Reference

```
public class TestStudent
{
    public static void main(String[] args)
    {
        Student aPsychMajor = new Student(111, 3.5);
        aPsychMajor.showStudent();
    }
}
```

Figure 4-26 The TestStudent



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command prompt displays the following text:

```
C:\Java>java TestStudent
Student #0 gpa is 0.0
C:\Java>
```



Learning About the `this` Reference

```
public class Student
{
    private int stuNum;
    private double gpa;

    public Student(int stuNum, double gpa)
    {
        this.stuNum = stuNum;
        this.gpa = gpa;
    }

    public void showStudent()
    {
        System.out.println("Student #" +
            stuNum + " gpa is " + gpa);
    }
}
```

These are the Student fields.

These parameters are locally declared in the Student constructor.

Because these identifiers are preceded by `this`, they refer to the fields in the Student class.

These identifiers, without `this`, refer to the locally declared variables and not the fields.

Command Prompt

```
C:\Java>java TestStudent
Student #111 gpa is 3.5
C:\Java>
```

Figure 4-28 The Student class using the explicit `this` reference



Using the `this` Reference to Make Overloaded Constructors More Efficient

- Avoid repetition within constructors
- Constructor calls other constructor
 - `this()`
 - More efficient and less error-prone



Using the this Reference to Make Overloaded Constructors More Efficient

- Suppose you create a Student class with data fields for a student number and a grade point average. Further suppose you want four overloaded constructors as follows:
 - A constructor that accepts an int and a double and assigns them the student number and grade point average, respectively
 - A constructor that accepts a double and assigns it to the grade point average, but initializes every student number to 999



Using the this Reference to Make Overloaded Constructors More Efficient

- A constructor that accepts an int and assigns it to the student number, but initializes every grade point average to 0.0
- A default constructor that assigns 999 to every student number and 0.0 to every grade point average



Using the `this` Reference to Make Overloaded Constructors More Efficient

```
public class Student
{
    private int stuNum;
    private double gpa;
    Student(int num, double avg)
    {
        stuNum = num;
        gpa = avg;
    }
    Student(double avg)
    {
        stuNum = 999;
        gpa = avg;
    }
    Student(int num)
    {
        stuNum = num;
        gpa = 0.0;
    }
    Student()
    {
        stuNum = 999;
        gpa = 0.0;
    }
}
```

Each constructor contains similar statements.

Figure 4-30 Student class with four constructors

```
public class Student
{
    private int stuNum;
    private double gpa;
    Student(int num, double avg)
    {
        stuNum = num;
        gpa = avg;
    }
    Student(double avg)
    {
        this(999, avg);
    }
    Student(int num)
    {
        this(num, 0.0);
    }
    Student()
    {
        this(999, 0.0);
    }
}
```

Each of these calls to `this()` calls the two-parameter version of the constructor.

Figure 4-31 The Student class using `this` in three of four constructors



TWO TRUTHS AND A LIE

1. Usually, you want each instantiation of a class to have its own non-static data fields, but each object does not need its own copy of most methods.
2. When you use a non-static method, the compiler accesses the correct object's field because you implicitly pass an object reference to the method.
3. The this reference is supplied automatically in classes; you cannot use it explicitly.



Remember:

▪ Class variables

- Static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.

▪ Instance variables

- Instance variables are declared in a class, but outside a method.

▪ Local variables

- Local variables are declared in methods, constructors, or blocks.



Remember:

▪ Instance Methods

- Instance method are methods which require an object of its class to be created before it can be called.

▪ Static Methods

- Static methods are the methods in Java that can be called without creating an object of class. They are referenced by the class name itself or reference to the Object of that class.



VI: Using `static` Variables

- Static methods **do not have a `this` reference** because they have no object associated with them
- **Class methods**
 - Do not have `this` reference
 - Have no object associated with them
- **Class variables**
 - Shared by every instantiation of class
 - Only one copy of `static` class variable per class



VI: Using static Variables

- Methods declared as **static cannot access instance variables**, but **instance methods can access both static and instance variables**.

(Source Files: BaseballPlayer.java and TestPlayer.java)



Using static Variables

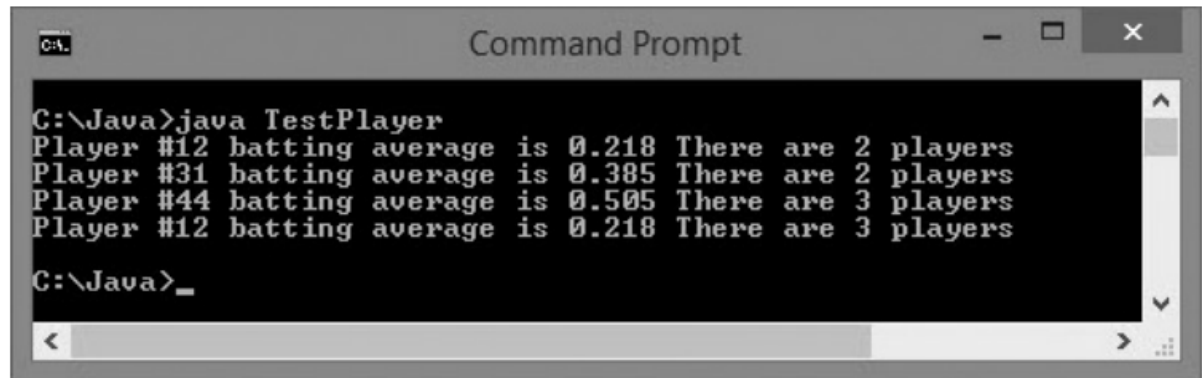
```
public class BaseballPlayer
{
    private static int count = 0;
    private int number;
    private double battingAverage;
    public BaseballPlayer(int id, double avg)
    {
        number = id;
        battingAverage = avg;
        count = count + 1;
    }
    public void showPlayer()
    {
        System.out.println("Player #" + number +
            " batting average is " + battingAverage +
            " There are " + count + " players");
    }
}
```

Figure 4-32 The BaseballPlayer class



Using static Variables

```
public class TestPlayer
{
    public static void main(String[] args)
    {
        BaseballPlayer aCatcher = new BaseballPlayer(12, .218);
        BaseballPlayer aShortstop = new BaseballPlayer(31, .385);
        aCatcher.showPlayer();
        aShortstop.showPlayer();
        BaseballPlayer anOutfielder = new BaseballPlayer(44, .505);
        anOutfielder.showPlayer();
        aCatcher.showPlayer();
    }
}
```



```
Command Prompt

C:\Java>java TestPlayer
Player #12 batting average is 0.218 There are 2 players
Player #31 batting average is 0.385 There are 2 players
Player #44 batting average is 0.505 There are 3 players
Player #12 batting average is 0.218 There are 3 players

C:\Java>
```



Using Constant Fields

- Create named constants using keyword **final**
 - Make its value unalterable after construction
- Can be set in class constructor
 - After construction cannot change **final** field's value



Using Constant Fields

```
public class Student
{
    private static final int SCHOOL_ID = 12345;
    private int stuNum;
    private double gpa;
    public Student(int stuNum, double gpa)
    {
        this.stuNum = stuNum;
        this.gpa = gpa;
    }
    public void showStudent()
    {
        System.out.println("Student #" + stuNum +
            " gpa is " + gpa);
    }
}
```

Figure 4-32 The Student class containing a symbolic constant



Using Constant Fields

- Fields declared to be **static** are not always **final**. Conversely, **final** fields are not always **static**. In summary,
- If you want to create a field that all instantiations of the class can access, but the field value can change, then it is static but not final.
 - For example, in the last section you saw a nonfinal static field in the `BaseballPlayer` class that held a changing count of all instantiated objects.



Using Constant Fields

- If you want each object created from a class to contain its own final value, you would declare the field to be final but not static.
 - For example, you might want each BaseballPlayer object to have its own, nonchanging date of joining the team.
- If you want all objects to share a single nonchanging value, then the field is static and final.



TWO TRUTHS AND A LIE

1. Methods declared as static receive a this reference that contains a reference to the object associated with them.
2. Methods declared as static are called class methods.
3. A final static field's value is shared by every object of a class.



VII: Using Automatically Imported, Prewritten Constants and Methods

- Many classes commonly used by wide variety of programmers
- **Package or library of classes**
 - Folder provides convenient grouping for classes
 - Many contain classes available only if explicitly named within program
 - Some classes available automatically



Using Automatically Imported, Prewritten Constants and Methods

- Fundamental or basic classes
 - Implicitly imported into every Java program
 - **java.lang** package
 - Only implicitly imported, named package
- Optional classes
 - Must be explicitly named



Using Automatically Imported, Prewritten Constants and Methods

- `java.lang.Math` class
 - Contains constants and methods used to perform common mathematical functions
 - No need to create instance
 - Imported automatically
 - Cannot instantiate objects of type `Math`
 - Constructor for `Math` class private



Method	Value That the Method Returns
<code>abs(x)</code>	Absolute value of x
<code>acos(x)</code>	Arc cosine of x
<code>asin(x)</code>	Arc sine of x
<code>atan(x)</code>	Arc tangent of x
<code>atan2(x, y)</code>	Theta component of the polar coordinate (r , θ) that corresponds to the Cartesian coordinate x , y
<code>ceil(x)</code>	Smallest integral value not less than x (ceiling)
<code>cos(x)</code>	Cosine of x
<code>exp(x)</code>	Exponent, where x is the base of the natural logarithms
<code>floor(x)</code>	Largest integral value not greater than x
<code>log(x)</code>	Natural logarithm of x
<code>max(x, y)</code>	Larger of x and y
<code>min(x, y)</code>	Smaller of x and y
<code>pow(x, y)</code>	x raised to the y power
<code>random()</code>	Random double number between 0.0 and 1.0
<code>rint(x)</code>	Closest integer to x (x is a double, and the return value is expressed as a double)
<code>round(x)</code>	Closest integer to x (where x is a float or double, and the return value is an int or long)
<code>sin(x)</code>	Sine of x
<code>sqrt(x)</code>	Square root of x
<code>tan(x)</code>	Tangent of x

Table 4-1 Common Math class methods



Using the `GregorianCalendar` Class

- Use prewritten classes
 - Use entire path with class name
 - Import class
 - Import package that contains class



Using the `GregorianCalendar` Class

▪ Wildcard symbol

- Alternative to importing class
 - Import entire package of classes
- Use asterisk
 - Can be replaced by any set of characters
 - Represents all classes in package
 - No disadvantage to importing extra classes
- Importing each class by name can be form of documentation



Using the `GregorianCalendar` Class

- **`GregorianCalendar`** class
 - Seven constructors
 - Default constructor creates calendar object containing current date and time in default locale (time zone)
 - Can also specify date, time, and time zone
 - `get ()` method
 - Access data fields



Using the GregorianCalendar Class

Arguments	Values Returned by <code>get()</code>
DAY_OF_YEAR	A value from 1 to 366
DAY_OF_MONTH	A value from 1 to 31
DAY_OF_WEEK	SUNDAY, MONDAY, ... SATURDAY, corresponding to values from 1 to 7
YEAR	The current year; for example, 2012
MONTH	JANUARY, FEBRUARY, ... DECEMBER, corresponding to values from 0 to 11
HOUR	A value from 1 to 12; the current hour in the A.M. or P.M.
AM_PM	A.M. or P.M., which correspond to values from 0 to 1
HOUR_OF_DAY	A value from 0 to 23 based on a 24-hour clock
MINUTE	The minute in the hour, a value from 0 to 59
SECOND	The second in the minute, a value from 0 to 59
MILLISECOND	The millisecond in the second, a value from 0 to 999

Table 4-2 Some possible arguments to and returns from the `GregorianCalendar.get()` method



Using the GregorianCalendar Class

```
import java.util.*;
import javax.swing.*;
public class AgeCalculator
{
    public static void main(String[] args)
    {
        GregorianCalendar now = new GregorianCalendar();
        int nowYear;
        int birthYear;
        int yearsOld;
        birthYear = Integer.parseInt
            (JOptionPane.showInputDialog(null,
                "In what year were you born?"));
        nowYear = now.get(GregorianCalendar.YEAR);
        yearsOld = nowYear - birthYear;
        JOptionPane.showMessageDialog(null,
            "This is the year you become " + yearsOld +
            " years old");
    }
}
```

Figure 4-33 The AgeCalculator application



TWO TRUTHS AND A LIE

1. The creators of Java have produced hundreds of classes for you to use in your programs.
2. Java packages are available only if you explicitly name them within your program.
3. The implicitly imported `java.lang` package contains fundamental Java classes.



VIII: Understanding Composition

■ Composition

- Describes relationship between classes when object of one class data field is within another class
- Called **has-a** relationship
 - Because one class "**has an**" instance of another
- Remember to supply values for contained object if it has no default constructor



Understanding Composition

```
public class NameAndAddress
{
    private String name;
    private String address;
    private int zipCode;
    public NameAndAddress(String nm, String add, int zip)
    {
        name = nm;
        address = add;
        zipCode = zip;
    }
    public void display()
    {
        System.out.println(name);
        System.out.println(address);
        System.out.println(zipCode);
    }
}
```

Figure 4-41 The NameAndAddress class



Understanding Composition

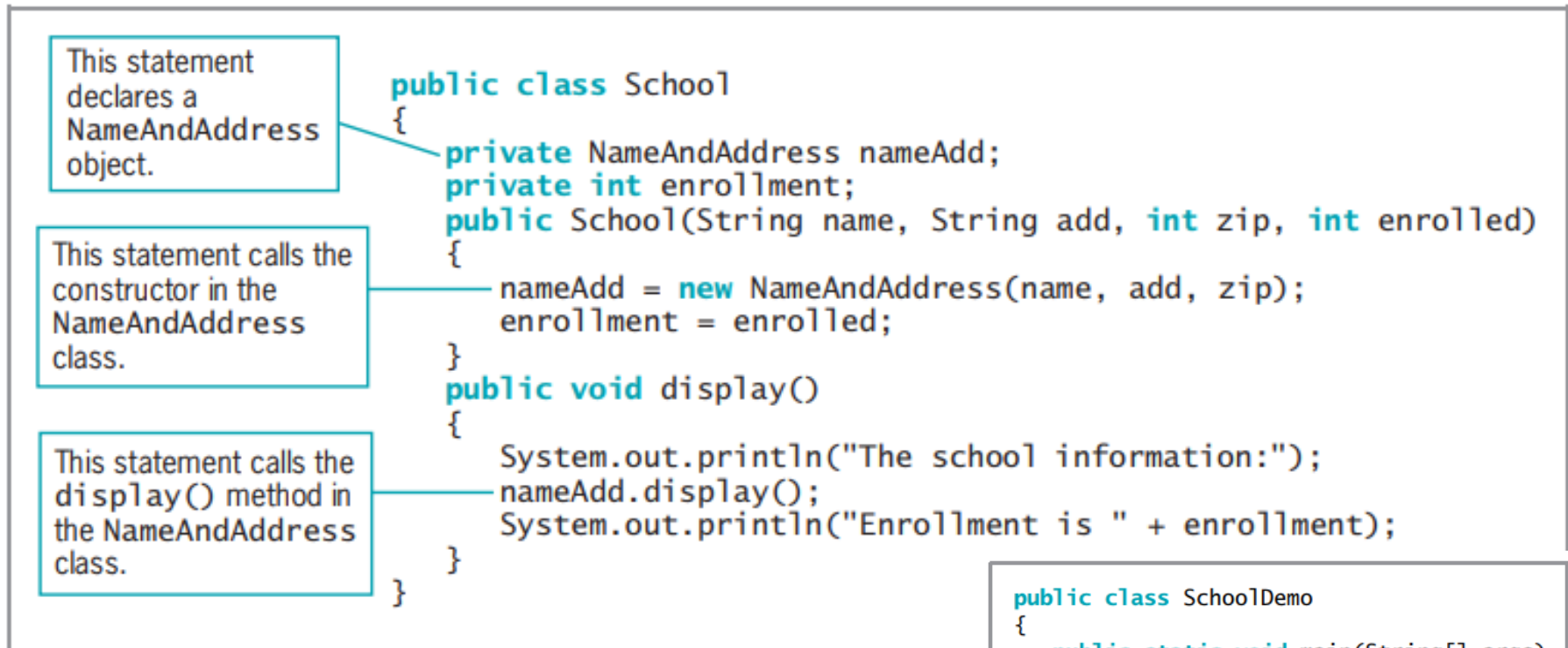


Figure 4-42 The `School` class

```

public class SchoolDemo
{
    public static void main(String[] args)
    {
        School mySchool = new School
            ("Audubon Elementary",
             "3500 Hoyne", 60618, 350);
        mySchool.display();
    }
}
  
```

Figure 4-43 The `SchoolDemo` program



A Brief Look at Nested and Inner Classes

- **Nested classes**

- Class within another class
- Stored together in one file

- **Nested class types**

- **static** member
 - A static member class has access to all static methods of the top-level class.
- Nonstatic member
 - Also known as inner classes: This type of class requires an instance; it has access to all data and methods of the top-level class.



A Brief Look at Nested and Inner Classes

- Local classes
 - These are local to a block of code.
- Anonymous classes
 - These are local classes that have no identifier



TWO TRUTHS AND A LIE

1. Exposition describes the relationship between classes when an object of one class is a data field within another class.
2. When you use an object as a data member of another object, you must remember to supply values for the contained object if it has no default constructor.
3. A nested class resides within another class.



You Do It

- Demonstrating scope
- Overloading methods
- Creating overloaded constructors
- Using an explicitly imported prewritten class
- Creating an interactive application with a timer



Don't Do It

- Don't try to use a variable that is out of scope
- Don't assume that a constant is still a constant when passed to a method's parameter
- Don't overload methods by giving them different return types
- Don't think that *default constructor* means only the automatically supplied version
- Don't forget to write a default constructor for a class that has other constructors



Summary

- Variable's scope
 - Portion of program in which you can reference variable
- Block
 - Code between a pair of curly braces
- Overloading
 - Writing multiple methods with same name but different argument lists
- Store separate copies of data fields for each object
 - But just one copy of each method



Summary

- `static` class variables
 - Shared by every instantiation of a class
- Prewritten classes
 - Stored in packages
- `import` statement
 - Notifies Java program that class names refer to those within imported class
- A class can contain other objects as data members
- You can create nested classes that are stored in the same file



End of Module.



REFERENCE:

Farrell, J. (2016). *Java Programming*. 8th Edition.
Course Technology, Cengage Learning.

