JAVA™ PROGRAMMING

Chapter 4: More Object Concepts

Objectives

- Understand blocks and scope
- Overload a method
- Avoid ambiguity
- Create and call constructors with parameters
- Use the `this` reference

Objectives (cont'd.)

- Use static fields
- Use automatically imported, prewritten constants and methods
- Use composition and nest classes

Understanding Blocks and Scope

- **Blocks**
  - Use opening and closing curly braces
  - Can exist entirely within another block or entirely outside of and separate from another block
  - Cannot overlap
  - Types:
    - **Outside block** (or **outer block**)
    - **Inside block** (or **inner block**)
    - **Nested** (contained entirely within the outside block)

Understanding Blocks and Scope (cont'd.)

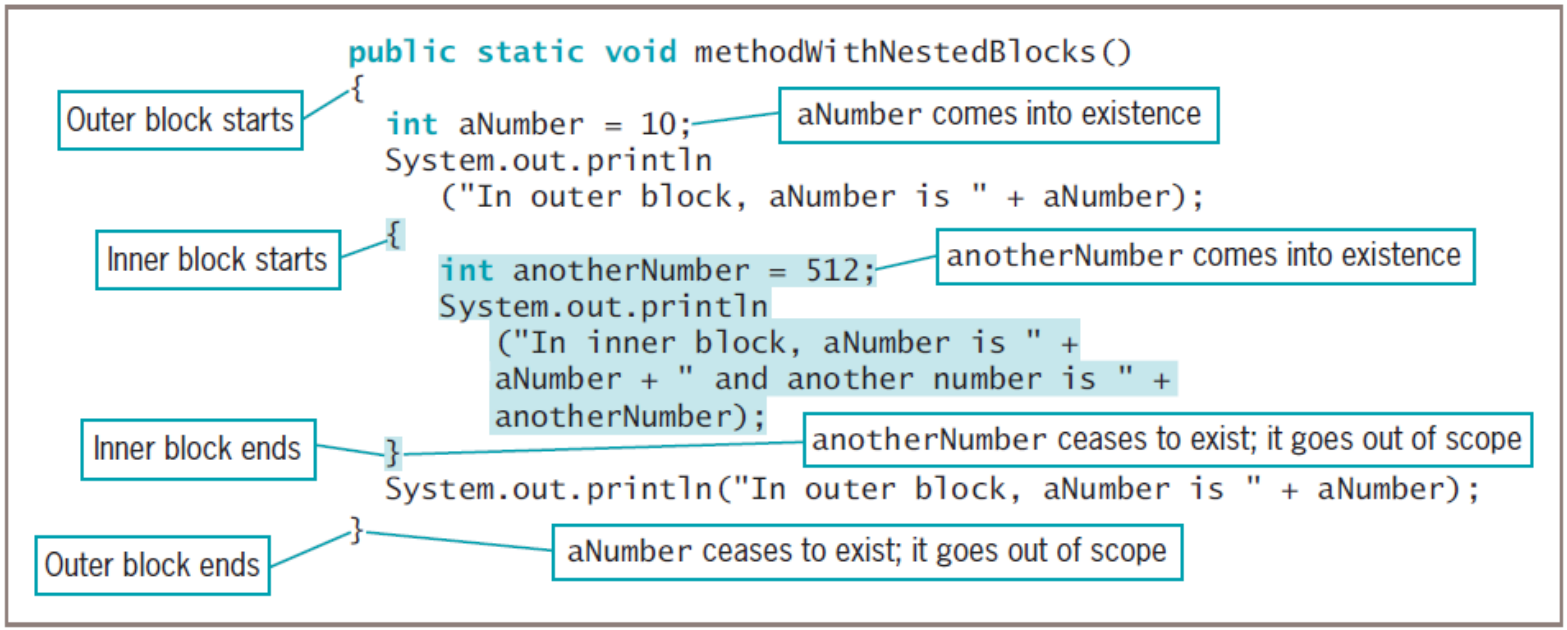


Figure 4-1 A method with nested blocks

Understanding Blocks and Scope (cont'd.)

- Scope
  - The portion of a 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 (cont'd.)

- **Redeclare the variable**
  - You cannot declare the same variable name more than once within a block
  - An illegal action

Understanding Blocks and Scope (cont'd.)

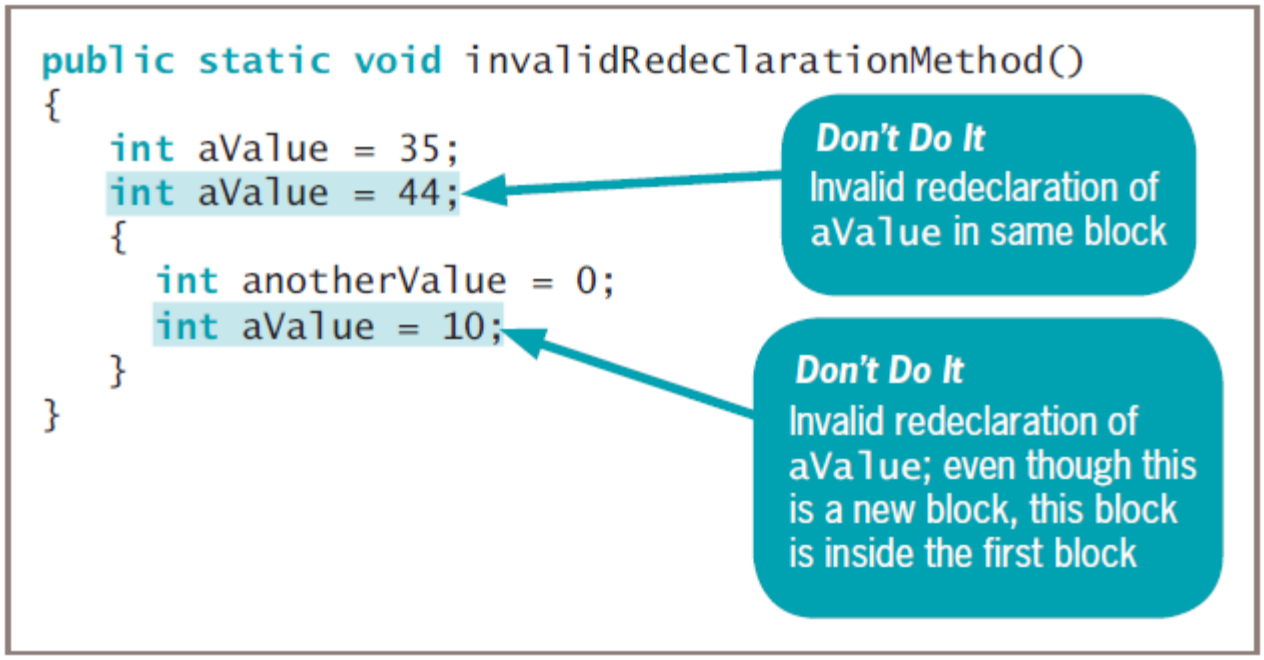


Figure 4-5 The `invalidRedeclarationMethod()`

Understanding Blocks and Scope (cont'd.)

- **Override**
  - Occurs when you use the variable's name within the method in which it is declared
    - The variable takes precedence over any other variable with the same name in another method
  - Locally declared variables always mask or hide other variables with the same name elsewhere in the class

Understanding Blocks and Scope (cont'd.)

```
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 is " + aNumber);
     aNumber = 862;
     System.out.println("In secondMethod(), after an assignment " +
        "aNumber is " + 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 than the one in main(). It is declared locally in this method.

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

Figure 4-6 The OverridingVariable class

Understanding Blocks and Scope (cont'd.)

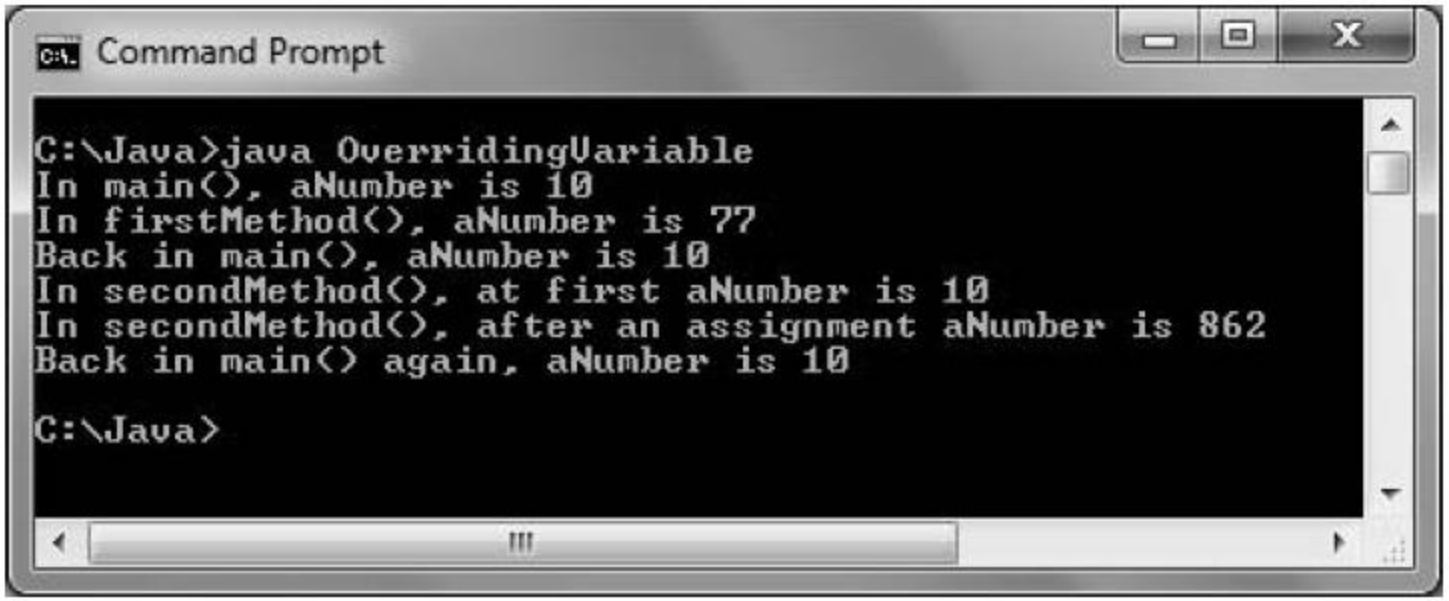


Figure 4-7 Output of the `OverridingVariable` application

Overloading a Method

- **Overloading**
  - Using one term to indicate diverse meanings
  - Writing multiple methods with the same name but with different arguments
  - The compiler understands the meaning based on the arguments used with the method call
  - It's convenient for programmers to use one reasonable name
    - For tasks that are functionally identical
    - Except for argument types

Overloading a Method (cont'd.)

```
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

Automatic Type Promotion in Method Calls

- If an application contains just one version of a method:
  - Call the method using a parameter of the correct data type or one that can be promoted to the correct data type
    - Order of promotion:
      - `Double`, `float`, `long`, `int`

Automatic Type Promotion in Method Calls (cont'd.)

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

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

Learning About Ambiguity

- **Ambiguous** situation
  - When the compiler cannot determine which method to use
- Overload methods
  - Correctly provide different parameter lists for methods with the same name
- Illegal methods
  - Methods with identical names that have identical argument lists but different return types

Creating and Calling Constructors with Parameters

- Java automatically provides a 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

Creating and Calling Constructors with Parameters (cont'd.)

- When you write a constructor for a class, you no longer receive the automatically provided default constructor
- If a class's only constructor requires an argument, you must provide an argument for every object of the class

Overloading Constructors

- Use constructor parameters to initialize field values, or any other purpose
- If constructor parameter lists differ, there is no ambiguity about which constructor method to call

Overloading Constructors (cont'd.)

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

Figure 4-22 The `Employee` class that contains two constructors

Learning About the `this` Reference

- Instantiate an object from a class
  - Memory is reserved for each instance field in the class
  - It's not necessary to store a separate copy of each variable and method for each instantiation of a class
- In Java:
  - One copy of each method in a class is stored
  - All instantiated objects can use one copy

Learning About the `this` Reference (cont'd.)

- **Reference**
  - An object's memory address
  - Implicit
    - Automatically understood without actually being written

Learning About the `this` Reference (cont'd.)

- **`this` reference**
  - The reference to an object
  - Passed to any object's nonstatic class method
  - A reserved word in Java
- You don't need to use the `this` reference in methods you write in most situations

Learning About the `this` Reference (cont'd.)

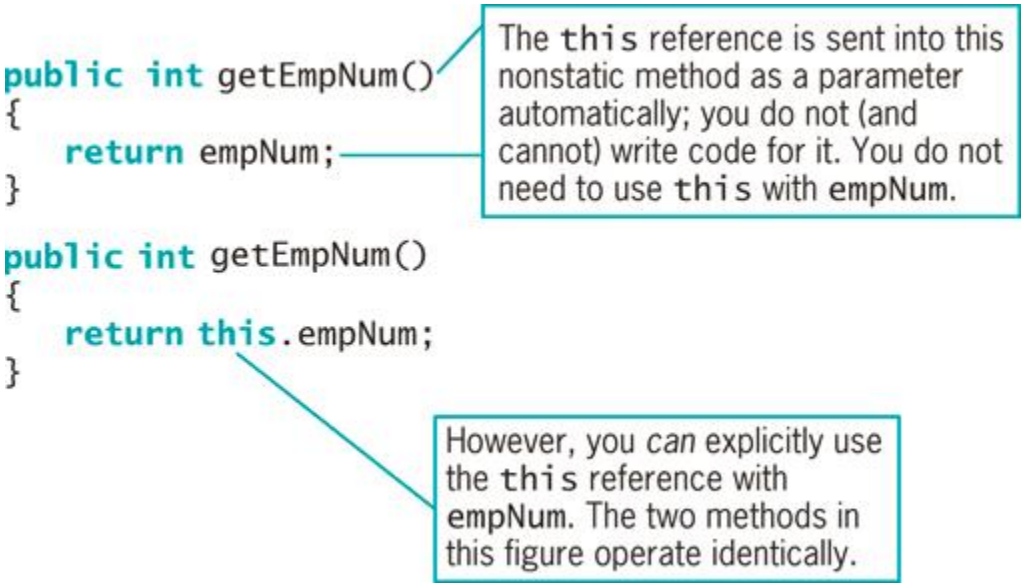


Figure 4-24 Two versions of the `getEmpNum()` method, with and without an explicit `this` reference

Learning About the `this` Reference (cont'd.)

- **`this` reference** (cont'd.)
  - Implicitly received by instance methods
  - Use to make classes work correctly
  - When used with a field name in a class method, the reference is to the class field instead of to the local variable declared within the method

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

```
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;
    }
}
```

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);
    }
}
```

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

Using `static` Fields

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

Using Constant Fields

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

Using Constant Fields (cont'd.)

```
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 4-35 The `Student` class containing a symbolic constant

Using Automatically Imported, Prewritten Constants and Methods

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

Using Automatically Imported, Prewritten Constants and Methods (cont'd.)

- **`java.lang`** package
  - Implicitly imported into every Java program
  - The only automatically imported, named package
  - The classes it contains are f**undamental classes** (or basic classes)
- **Optional classes**
  - Must be explicitly named

Using Automatically Imported, Prewritten Constants and Methods (cont'd.)

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

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

Table 4-1 Common `Math` class methods

Importing Classes That Are Not Imported Automatically

- Use prewritten classes
  - Use the entire path with the class name
  - Import the class
  - Import the package that contains the class
- **Wildcard symbol**
  - An alternative to importing the class
    - Import the entire package of classes
  - Use an asterisk
    - Can be replaced by any set of characters
    - Represents all classes in a package
    - There is no disadvantage to importing extra classes
  - Importing each class by name can be a form of documentation

Using the `GregorianCalendar` Class

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

Using the `GregorianCalendar` Class (cont'd.)

Arguments	Values Returned by `get()`
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 (cont'd.)

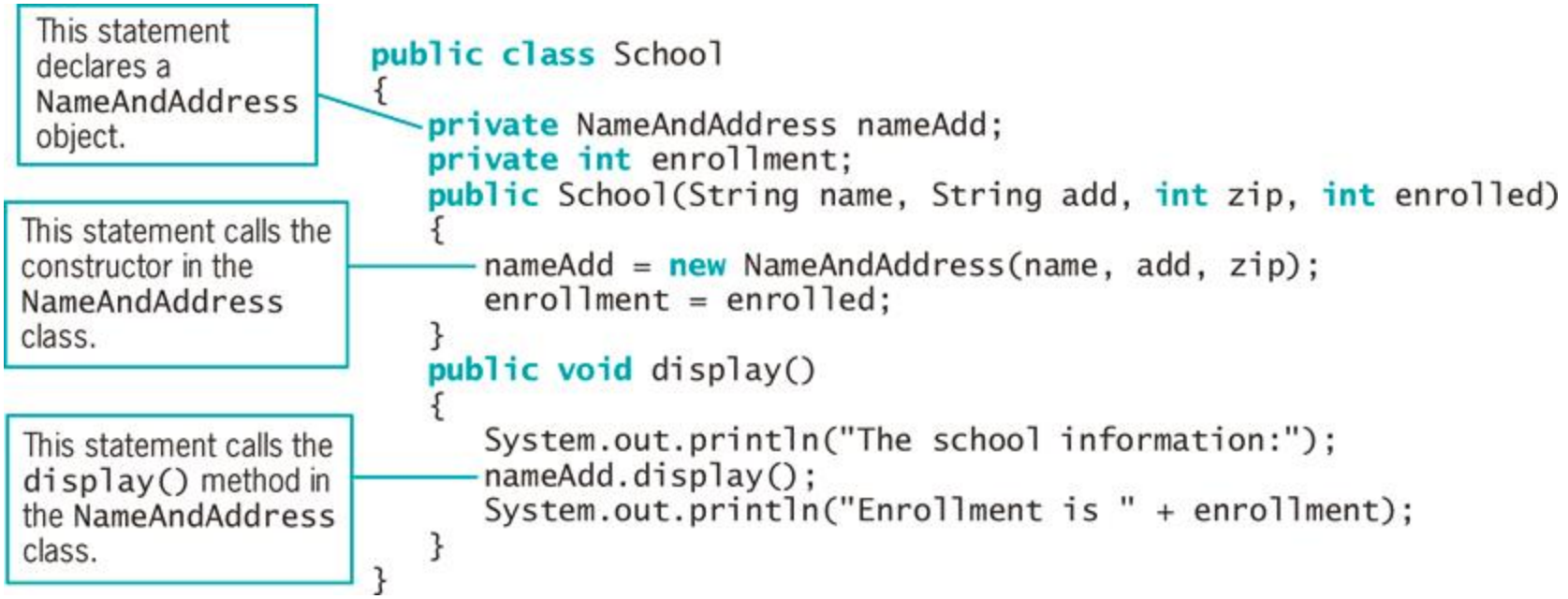
```
import java.util.*;
import javax.swing.*;
public class AgeCalculator
{
   public static void main(String[] args)
   {
      GregorianCalendar now = new GregorianCalendar();
      int nowYear;
      int birthYear;
      int yearsOld;
      birthYear = Integer.parseInt
         (JOptionPane.showInputDialog(null,
         "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 4-37 The `AgeCalculator` application

Understanding Composition and Nested Classes

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

Understanding Composition and Nested Classes (cont'd.)




```
public class School
{
    private NameAndAddress nameAdd;
    private int enrollment;
    public School(String name, String add, int zip, int enrolled)
    {
        nameAdd = new NameAndAddress(name, add, zip);
        enrollment = enrolled;
    }
    public void display()
    {
        System.out.println("The school information:");
        nameAdd.display();
        System.out.println("Enrollment is " + enrollment);
    }
}
```

Figure 4-42 The `School` class

Nested Classes

- **Nested classes**
  - A class within another class
  - Stored together in one file
- Nested class types
  - **`static` member classes**
  - **Nonstatic member classes**
  - **Local classes**
  - **Anonymous classes**

You Do It

- Demonstrating Scope
- Overloading Methods
- Creating Overloaded Constructors
- Using the `this` Reference to Make Constructors More Efficient
- Using Static and Nonstatic `final` fields
- Using the Java Web Site

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

Summary (cont'd.)

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