Java Programming, 9e

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

Understanding Blocks and Scope (1 of 7)

- **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 (2 of 7)

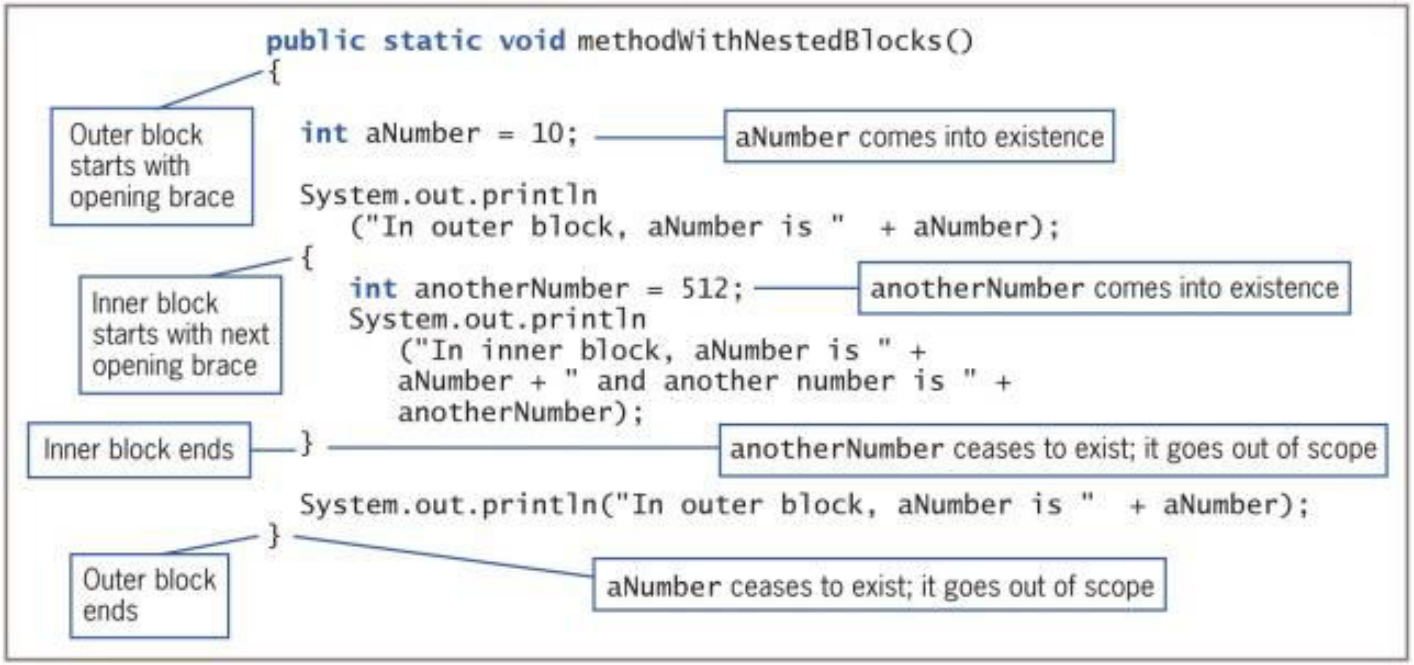

Figure 4-1 A method with nested blocks

Understanding Blocks and Scope (3 of 7)

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

Understanding Blocks and Scope (4 of 7)

```
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 (5 of 7)

- **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
  - **Shadowing**: locally declared variables always mask or hide other variables with the same name elsewhere in the class

Understanding Blocks and Scope (6 of 7)

```
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 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 (7 of 7)

```
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
```

Figure 4-7 Output of the `OverridingVariable` application

Overloading a Method (1 of 2)

- **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 is convenient for programmers to use one reasonable name
    - For tasks that are functionally identical
    - Except for argument types

Overloading a Method (2 of 2)

```
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 (1 of 2)

- 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 (2 of 2)

```
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 (1 of 2)

- Java automatically provides a constructor method when class-created default constructors do not require parameters
- Writing 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 (2 of 2)

- 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 (1 of 2)

- 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 (2 of 2)

```
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 (1 of 5)

- Instantiating an object from a class
  - Memory is reserved for each instance field in the class
  - It is 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 (2 of 5)

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

Learning About the `this` Reference (3 of 5)

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

Learning About the `this` Reference (4 of 5)

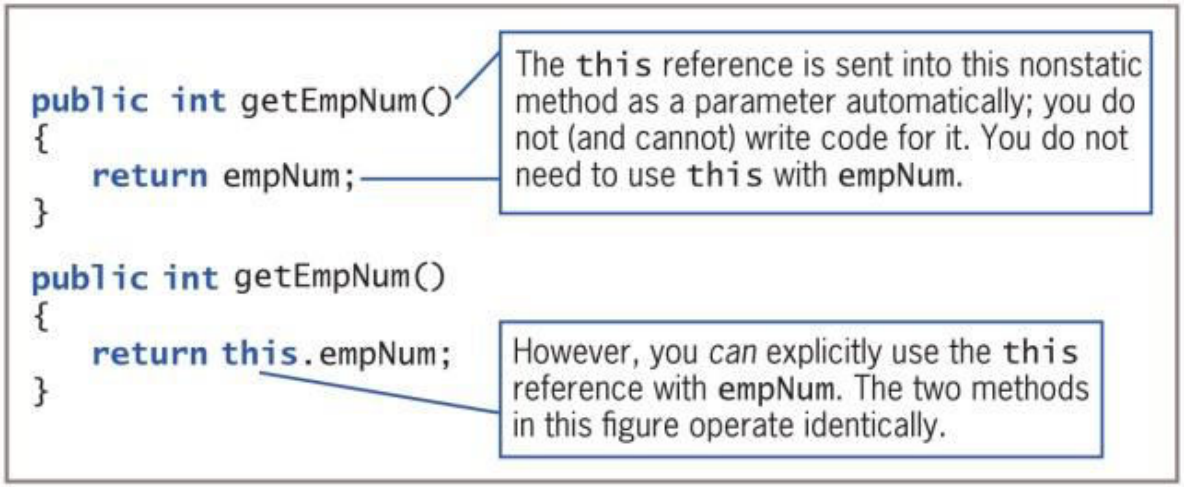

Figure 4-24 Two versions of the `getEmpNum()` method, with and without an explicit `this` reference

Learning About the `this` Reference (5 of 5)

- **`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 (1 of 3)

- 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 (2 of 3)

```
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

Using the this Reference to Make Overloaded Constructors More Efficient (3 of 3)

```
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

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 (1 of 2)

- 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 (2 of 2)

```
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);
   }
}
```

static final
symbolic constant

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

Using Automatically Imported, Prewritten Constants and Methods (1 of 5)

- 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 (2 of 5)

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

Using Automatically Imported, Prewritten Constants and Methods (3 of 5)

- 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

Using Automatically Imported, Prewritten Constants and Methods (4 of 5)

Table 4-1 Common Math class methods (continues)	
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

Using Automatically Imported, Prewritten Constants and Methods (5 of 5)

Table 4-1 Common Math class methods (continued)	
Method	**Value that the Method Returns**
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

Importing Classes That Are Not Imported Automatically (1 of 2)

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

Importing Classes That Are Not Imported Automatically (2 of 2)

- **Wildcard symbol**
  - An alternative to importing the class
    - Import the entire package of classes
  - Asterisk wildcard
    - 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 `LocalDate` Class (1 of 4)

- `LocalDate` class
  - No time zone information included with this class (local date only)
  - Static methods `now()` and `of()`
  - Class constructors are non-public
  - Various methods provided to perform date arithmetic
- **Enumeration**
  - Data type that consists of a list of values, such as:
    - JANUARY, FEBRUARY, MARCH etc.

Using the LocalDate Class (2 of 4)

```
import java.time.*;
public class LocalDateDemo
{
   public static void main(String[] args)
   {
      Local Date today = Loca1Date.now();
      Loca1Date graduationDate = Loca1Date.of(20l8, 5, 29);
      System.out.println("Today is " + today);
      System.out.println("Graduation is " + graduationDate);
   }
}
```

Figure 4-37 The LocalDateDemo application

Using the LocalDate Class (3 of 4)

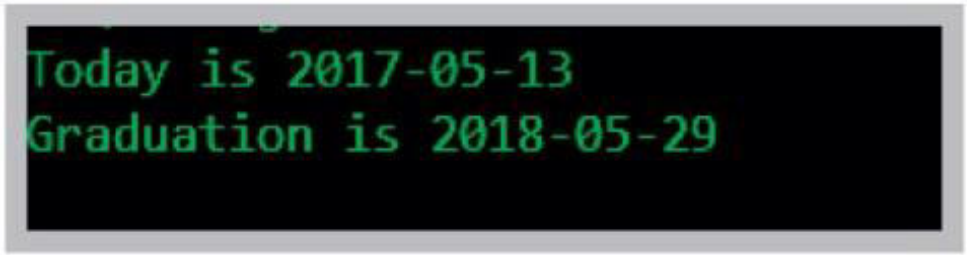

Figure 4-38 Execution of the `LocalDateDemo` application

Using the `LocalDate` Class (4 of 4)

```
import java.time.*;
import java.util.Scanner;
public class DeliveryDate
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      LocalDate orderDate;
      int mo;
      int day;
      int year;
      final int WEEKS_FOR_DELIVERY = 2;
      System.out.print("Enter order month ");
      mo = input.nextInt();
      System.out.print("Enter order day ");
      day = input.nextInt();
      System.out.print("Enter order year ");
      year = input.nextInt();
      orderDate = LocalDate.of(year, mo, day);
      System.out.println("Order date is " + orderDate);
      System.out.println("Delivery date is " +
         orderDate.plusWeeks(WEEKS_FOR_DELIVERY)) ;
   }
}
```

Figure 4-39 The `DeliveryDate` application

Understanding Composition and Nested Classes (1 of 2)

- **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 (2 of 2)

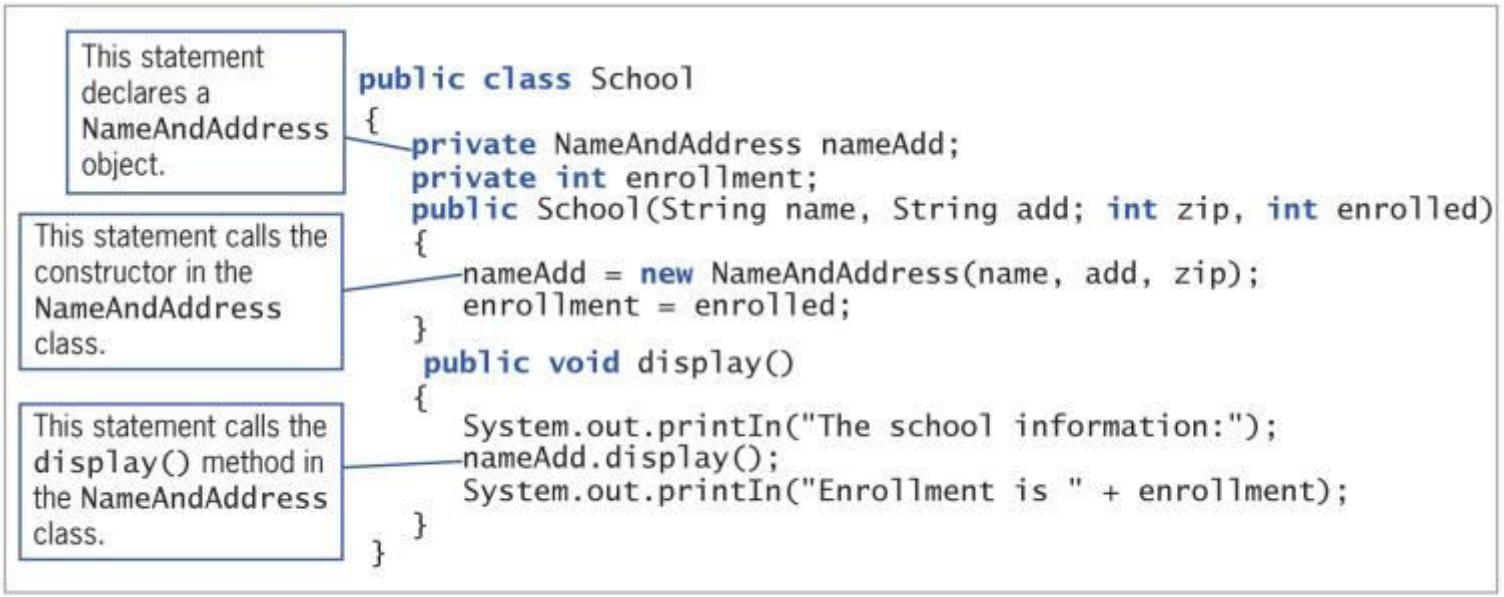

Figure 4-43 The School class

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**

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
- Don’t assume that a wildcard in an import statement works like a DOS or UNIX wildcard. The wildcard works only with specific packages and does not import embedded packages.

Summary (1 of 2)

- 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 (2 of 2)

- `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