CHAPTER 9

THINKING IN OBJECTS

Objectives

- To create immutable objects from immutable classes to protect the contents of objects (§9.2).
- To determine the scope of variables in the context of a class (§9.3).
- To use the keyword `this` to refer to the object itself (§9.4).
- To apply class abstraction to develop software (§9.5).
- To explore the differences between the procedural paradigm and object-oriented paradigm (§9.6).
- To design programs using the object-oriented paradigm (§§9.7-9.9).
Chapter 9 Thinking in Objects

9.1 Introduction

The preceding two chapters introduced the concept of objects and classes. You learned how to program using objects from several classes in the Java API (e.g., Date, Random, JFrame, String, StringBuilder, File, Scanner, PrintWriter). This chapter focuses on class design. Several examples will illustrate the advantages of object-oriented programming. To begin, we will introduce several language features supporting these examples.

9.2 Immutable Objects and Classes

Normally, you create an object and allow its contents to be changed later. Occasionally, it is desirable to create an object whose contents cannot be changed once the object is created. We call such an object an immutable object and its class an immutable class. The String class, for example, is immutable. If you deleted the set method in the Circle class in Listing 7.7, the class would be immutable because radius is private and cannot be changed without a set method.

A class with all private data fields and no mutators is not necessarily immutable. An example is the class Student.

```java
public class Student {
    private int id;
    private String name;
    private java.util.Date dateCreated;

    public Student(int ssn, String newName) {
        id = ssn;
        name = newName;
        dateCreated = new java.util.Date();
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public java.util.Date getDateCreated() {
        return dateCreated;
    }
}
```

Since the full name java.util.Date for the Date class is used in line 4, there is no need to import the class.

As shown in the following code, the data field dateCreated is returned using the getDateCreated() method. This is a reference to a Date object. Through this reference, the content for dateCreated can be changed.

```java
public class Test {
    public static void main(String[] args) {
        Student student = new Student(111223333, "John");
        java.util.Date dateCreated = student.getDateCreated();
        dateCreated.setTime(200000); // Now dateCreated field is changed!
    }
}
```
9.3 The Scope of Variables

For a class to be immutable, it must meet the following requirements:

- Declare all data fields private;
- Provide no mutator methods;
- Provide no accessor method that returns a reference to a data field that is mutable.

9.3 The Scope of Variables

Chapter 5, “Methods,” discussed local variables and their scope rules. Local variables are declared and used inside a method locally. This section discusses the scope rules of all the variables in the context of a class.

Instance and static variables in a class are referred to as the class’s variables or data fields. A variable defined inside a method is referred to as a local variable. The scope of a class’s variables is the entire class, regardless of where the variables are declared. A class’s variables and methods can be declared in any order in the class, as shown in Figure 9.1(a). The exception is when a data field is initialized based on a reference to another data field. In such cases, the other data field must be declared first, as shown in Figure 9.1(b). For consistency, this book declares data fields at the beginning of the class.

```
public class Circle {
    private double radius = 1;
    public double findArea() {
        return radius * radius * Math.PI;
    }
}
```

(a) Variable radius and method findArea() can be declared in any order

```
public class Foo {
    private int j = i + 1;
    private int i;
}
```

(b) i has to be declared before j because j’s initial value is dependent on i.

**Figure 9.1** Members of a class can be declared in any order, with one exception.

You can declare a class’s variable only once, but you can declare the same variable name in a method many times in different nonnesting blocks.

If a local variable has the same name as a class’s variable, the local variable takes precedence and the class’s variable with the same name is hidden. For example, in the following program, x is defined as an instance variable and as a local variable in the method.

```
class Foo {
    int x = 0; // instance variable
    int y = 0;

    Foo() {
    }

    void p() {
        int x = 1; // local variable
        System.out.println("x = " + x);
        System.out.println("y = " + y);
    }
}
```

What is the printout for f.p(), where f is an instance of Foo? The printout for f.p() is 1 for x and 0 for y. Here is why:

- x is declared as a data field with the initial value of 0 in the class, but is also defined in the method p() with an initial value of 1. The latter x is referenced in the `System.out.println` statement.
- y is declared outside the method p(), but is accessible inside it.
Chapter 9  Thinking in Objects

Tip
To avoid confusion and mistakes, do not use the names of instance or static variables as local variable names, except for method parameters.

9.4 The this Reference

The this keyword is the name of a reference that refers to an object itself. One common use of the this keyword is reference a class’s hidden data fields. For example, a property name is often used as the parameter name in a set method for the property. In this case, you need to reference the hidden property name in the method in order to set a new value to it. A hidden static variable can be accessed simply by using the ClassName.StaticVariable reference. A hidden instance variable can be accessed by using the keyword this, as shown in Figure 9.2(a).

```
public class Foo {
    int i = 5;
    static double k = 0;
    void setI(int i) {
        this.i = i;
    }
    static void setK(double k) {
        Foo.k = k;
    }
}
```

(a)

Suppose that f1 and f2 are two objects of Foo. Invoking f1.setI(10) is to execute this.i = 10, where this refers f1. Invoking f2.setI(45) is to execute this.i = 45, where this refers f2.

```
Figure 9.2  The keyword this refers to the calling object that invokes the method.
```

The line this.i = i means “assign the value of parameter i to the data field i of the calling object.” The keyword this refers to the object that invokes the instance method setI, as shown in Figure 9.2(b). The line Foo.k = k means that the value in parameter k is assigned to the static data field k of the class, which is shared by all the objects of the class.

Another common use of the this keyword to enable a constructor to invoke another constructor of the same class. For example, you can redefine the Circle class as follows:

```
public class Circle {
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    public Circle() {
        this(1.0);
    }
    public double getArea() {
        return this.radius * this.radius * Math.PI;
    }
}
```

Every instance variable belongs to an instance represented by this, which is normally omitted.

The line this(1.0) in the second constructor invokes the first constructor with a double value argument.
Class Abstraction and Encapsulation

In Chapter 5, “Methods,” you learned about method abstraction and used it in program development. Java provides many levels of abstraction. **Class abstraction** is the separation of class implementation from the use of a class. The creator of a class describes it and lets the user know how it can be used. The collection of methods and fields that are accessible from outside the class, together with the description of how these members are expected to behave, serves as the **class’s contract**. As shown in Figure 9.3, the user of the class does not need to know how the class is implemented. The details of implementation are encapsulated and hidden from the user. This is known as **class encapsulation**. For example, you can create a **Circle** object and find the area of the circle without knowing how the area is computed.

Class abstraction and encapsulation are two sides of the same coin. There are many real-life examples that illustrate the concept of class abstraction. Consider building a computer system, for instance. Your personal computer is made up of many components, such as a CPU, CD-ROM, floppy disk, motherboard, fan, and so on. Each component can be viewed as an object that has properties and methods. To get the components to work together, all you need to know is how each component is used and how it interacts with the others. You don’t need to know how they work internally. The internal implementation is encapsulated and hidden from you. You can build a computer without knowing how a component is implemented.

The computer-system analogy precisely mirrors the object-oriented approach. Each component can be viewed as an object of the class for the component. For example, you might have a class that models all kinds of fans for use in a computer, with properties like fan size and speed, and methods like start, stop, and so on. A specific fan is an instance of this class with specific property values.

As another example, consider getting a loan. A specific loan can be viewed as an object of a **Loan** class. Interest rate, loan amount, and loan period are its data properties, and computing monthly payment and total payment are its methods. When you buy a car, a loan object is created by instantiating the class with your loan interest rate, loan amount, and loan period. You can then use the methods to find the monthly payment and total payment of your loan. As a user of the **Loan** class, you don’t need to know how these methods are implemented.

Listing 2.7, ComputeLoan.java, presented a program for computing loan payments. The program cannot be reused in other programs. One way to fix this problem is to define static
methods for computing monthly payment and total payment. However, this solution has limitations. Suppose you wish to associate a date with the loan. The ideal way is to create an object that ties the properties for loan information and date together. Figure 9.4 shows the UML class diagram for the **Loan** class.

### FIGURE 9.4

The **Loan** class models the properties and behaviors of loans.

The UML diagram in Figure 9.4 serves as the contract for the **Loan** class. Throughout the book, you will play the role of both class user and class writer. The user can use the class without knowing how the class is implemented. Assume that the **Loan** class is available. Let us begin by writing a test program that uses the **Loan** class in Listing 9.1.

### LISTING 9.1 TestLoanClass.java

```java
import java.util.Scanner;

public class TestLoanClass {
    /** Main method */
    public static void main(String[] args) {
        // Create a Scanner
        Scanner input = new Scanner(System.in);

        // Enter yearly interest rate
        System.out.print("Enter yearly interest rate, for example, 8.25: ");
        double annualInterestRate = input.nextDouble();

        // Enter number of years
        System.out.print("Enter number of years as an integer: ");
        int numberOfYears = input.nextInt();

        // Enter loan amount
        System.out.print("Enter loan amount, for example, 120000.95: ");
```
9.5 Class Abstraction and Encapsulation

```java
double loanAmount = input.nextDouble();

// Create Loan object
Loan loan = new Loan(annualInterestRate, numberOfYears, loanAmount);

// Format to keep two digits after the decimal point
double monthlyPayment = (int)(loan.getMonthlyPayment() * 100) / 100.0;
double totalPayment = (int)(loan.getTotalPayment() * 100) / 100.0;

// Display results
System.out.println("The loan was created on " +
  loan.getLoanDate().toString() + \nThe monthly payment is " +
  monthlyPayment + \nThe total payment is " + totalPayment);
```

```java
LISTING 9.2 Loan.java

public class Loan {
  private double annualInterestRate;
  private int numberOfYears;
  private double loanAmount;
  private java.util.Date loanDate;

  /** Default constructor */
  public Loan() {
    this(2.5, 1, 1000);
  }

  /** Construct a loan with specified annual interest rate,
   * number of years and loan amount */
  public Loan(double annualInterestRate, int numberOfYears,
              double loanAmount) {
    this.annualInterestRate = annualInterestRate;
    this.numberOfYears = numberOfYears;
    this.loanAmount = loanAmount;
    loanDate = new java.util.Date();
  }

  /** Return annualInterestRate */
  public double getAnnualInterestRate() {
```

Enter yearly interest rate, for example, 8.25: 2.5
Enter number of years as an integer: 5
Enter loan amount, for example, 120000.95: 1000
The loan was created on Sat Jun 10 21:12:50 EDT 2006
The monthly payment is 17.74
The total payment is 1064.84

The main method reads interest rate, payment period (in years), and loan amount; creates a Loan object; and then obtains the monthly payment (lines 33–34) and total payment (lines 35-36) using the instance methods in the Loan class.

The Loan class can be implemented as in Listing 9.2.
Chapter 9  Thinking in Objects

310

```java
  return annualInterestRate;
}
/** Set a new annualInterestRate */
public void setAnnualInterestRate(double annualInterestRate) {
  this.annualInterestRate = annualInterestRate;
}
/** Return numberOfYears */
public int getNumberOfYears() {
  return numberOfYears;
}
/** Set a new numberOfYears */
public void setNumberOfYears(int numberOfYears) {
  this.numberOfYears = numberOfYears;
}
/** Return loanAmount */
public double getLoanAmount() {
  return loanAmount;
}
/** Set a new loanAmount */
public void setLoanAmount(double loanAmount) {
  this.loanAmount = loanAmount;
}
/** Find monthly payment */
public double getMonthlyPayment() {
  double monthlyInterestRate = annualInterestRate / 1200;
  double monthlyPayment = loanAmount * monthlyInterestRate / (1 -
                  (Math.pow(1 / (1 + monthlyInterestRate), numberOfYears * 12)));
  return monthlyPayment;
}
/** Find total payment */
public double getTotalPayment() {
  double totalPayment = getMonthlyPayment() * numberOfYears * 12;
  return totalPayment;
}
/** Return loan date */
public java.util.Date getLoanDate() {
  return loanDate;
}
```

From a class developer’s perspective, a class is designed for use by many different customers. In order to be useful in a wide range of applications, a class should provide a variety of ways for customization through constructors, properties, and methods.

The `Loan` class contains two constructors, four get methods, three set methods, and the methods for finding monthly payment and total payment. You can construct a `Loan` object by using the no-arg constructor or the one with three parameters: annual interest rate, number of years, and loan amount. When a loan object is created, its date is stored in the `loanDate` field. The `getLoanDate` method returns the date. The three get methods, `getAnnualInterest`, `getNumberOfYears`, and `getLoanAmount`, return annual interest rate, payment years, and loan amount, respectively. All the data properties and methods in this class are tied to a specific instance of the `Loan` class. Therefore, they are instance variables or methods.
Important Pedagogical TIP

The UML diagram for the Loan class is shown in Figure 9.4. Students should begin by writing a test program that uses the Loan class even though they don’t know how the Loan class is implemented. This has three benefits:

- It demonstrates that developing a class and using a class are two separate tasks.
- It enables you to skip the complex implementation of certain classes without interrupting the sequence of the book.
- It is easier to learn how to implement a class if you are familiar with the class through using it.

For all the examples from now on, you may first create an object from the class and try to use its methods and then turn your attention to its implementation.

9.6 Object-Oriented Thinking

Chapters 1–6 introduced fundamental programming techniques for problem solving using loops, methods, and arrays. The study of these techniques lays a solid foundation for object-oriented programming. Classes provide more flexibility and modularity for building reusable software. This section improves the solution for a problem introduced in Chapter 3 using the object-oriented approach. From the improvements, you will gain insight on the differences between the procedural programming and object-oriented programming and see the benefits of developing reusable code using objects and classes.

Listing 3.6, ComputeBMI.java, presented a program for computing body mass index. The program cannot be reused in other programs. To make the code reusable, define a static method to compute body mass index as follows:

```java
public static double getBMI(double weight, double height)
```

This method is useful for computing body mass index for a specified weight and height. However, it has limitations. Suppose you need to associate the weight and height with a person’s name and birth date. You may declare separate variables to store these values. But these values are not tightly coupled. The ideal way to couple them is to create an object that contains them. Since these values are tied to individual objects, they should be stored in instance data fields. You can declare a class named BMI, as shown in Figure 9.5.

```java
class BMI {
    String name;
    int age;
    double weight;
    double height;

    BMI(String name, int age, double weight, double height) {
        this.name = name;
        this.age = age;
        this.weight = weight;
        this.height = height;
    }

    BMI(String name, double weight, double height) {
        this.name = name;
        this.weight = weight;
        this.height = height;
        this.age = 20;
    }

    double getBMI() {
        return weight / (height * height);
    }

    String getStatus() {
        return (getBMI() < 18.5) ? "underweight" : (getBMI() < 25) ? "normal" : (getBMI() < 30) ? "overweight" : "obese";
    }
}
```

For all the examples from now on, you may first create an object from the class and try to use its methods and then turn your attention to its implementation.

**Figure 9.5** The BMI class encapsulates BMI information.
Assume that the **BMI** class is available. Listing 9.3 gives a test program that uses this class.

**Listing 9.3 UseBMIClass.java**

```java
public class UseBMIClass {
    public static void main(String[] args) {
        System.out.println("The BMI for "+" is "
                        + bmi1.getName() + " " + bmi1.getBMI() + " " + bmi1.getStatus());
        System.out.println("The BMI for "+" is "
                        + bmi2.getName() + " " + bmi2.getBMI() + " " + bmi2.getStatus());
    }
}
```

Line 3 creates an object `bmi1` for John Doe and line 7 creates an object `bmi2` for Peter King. You can use the instance methods `getName()`, `getBMI()`, and `getStatus()` to return the BMI information in a **BMI** object.

The **BMI** class can be implemented as in Listing 9.4.

**Listing 9.4 BMI.java**

```java
public class BMI {
    private String name;
    private int age;
    private double weight; // in pounds
    private double height; // in inches
    public final double KILOGRAMS_PER_POUND = 0.45359237;
    public final double METERS_PER_INCH = 0.0254;
    public BMI(String name, int age, double weight, double height) { 
        this.name = name;
        this.age = age;
        this.weight = weight;
        this.height = height;
    }
    public BMI(String name, double weight, double height) {
        this(name, 20, weight, height);
    }
    public double getBMI() {
        double bmi = weight * KILOGRAMS_PER_POUND / ((height * METERS_PER_INCH) * (height * METERS_PER_INCH));
        return Math.round(bmi * 100) / 100.0;
    }
    public String getStatus() {
        double bmi = getBMI();
        if (bmi < 16)
```
The mathematic formula for computing the BMI using weight and height is given in §3.3.6, Problem: Computing Body Mass Index. The instance method `getBMI()` returns the BMI. Since the weight and height are instance data fields in the object, the `getBMI()` method can use these properties to compute the BMI for the object.

The instance method `getStatus()` returns a string that interprets the BMI. The interpretation is also given in §3.3.6.

Note that the `BMI` class is immutable.

This example demonstrates the advantages of using the object-oriented paradigm over the procedural paradigm. The procedural paradigm focuses on designing methods. The object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

In procedural programming, data and operations on the data are separate, and this methodology requires sending data to methods. Object-oriented programming places data and the operations that pertain to them within a single entity called an `object`; this approach solves many of the problems inherent in procedural programming. The object-oriented programming approach organizes programs in a way that mirrors the real world, in which all objects are associated with both attributes and activities. Using objects improves software reusability and makes programs easier to develop and easier to maintain. Programming in Java involves thinking in terms of objects; a Java program can be viewed as a collection of cooperating objects.

### 9.7 Designing the Course Class

The philosophy of this book is teaching by example and learning by doing. To this extent, the book provides a variety of examples to demonstrate object-oriented programming. The following three sections offer additional examples on designing classes.
Suppose you need to process course information. Each course has a name and students who take the course. You should be able to add/drop a student to/from the course. You can use a class to model the courses, as shown in Figure 9.6.

### Course

<table>
<thead>
<tr>
<th>CourseName: String</th>
<th>The name of the course.</th>
</tr>
</thead>
<tbody>
<tr>
<td>students: String[]</td>
<td>An array to store the students for the course.</td>
</tr>
<tr>
<td>numberOfStudents: int</td>
<td>The number of students (default: 0).</td>
</tr>
</tbody>
</table>

+ Course(courseName: String) | Creates a course with the specified name. |
| + getCourseName(): String | Returns the course name. |
| + addStudent(student: String): void | Adds a new student to the course. |
| + dropStudent(String student): void | Drops a student from the course. |
| + getStudents(): String[] | Returns the students for the course. |
| + getNumberOfStudents(): int | Returns the number of students for the course. |

**Figure 9.6** The `Course` class models the courses.

A `Course` object can be created using the constructor `Course(String name)` by passing a course name. You can add students to the course using the `addStudent(String student)` method, drop a student from the course using the `dropStudent(String student)` method, and return all the students for the course using the `getStudents()` method. Suppose the class is available; Listing 9.5 gives a test class that creates two courses and adds students to them.

### Listing 9.5 TestCourse.java

```java
public class TestCourse {
    public static void main(String[] args) {
        Course course1 = new Course("Data Structures");
        Course course2 = new Course("Database Systems");

        course1.addStudent("Peter Jones");
        course1.addStudent("Brian Smith");
        course1.addStudent("Anne Kennedy");

        course2.addStudent("Peter Jones");
        course2.addStudent("Steve Smith");

        System.out.println("Number of students in course1: " + course1.getNumberOfStudents());
        String[] students = course1.getStudents();
        for (int i = 0; i < course1.getNumberOfStudents(); i++)
            System.out.print(students[i] + ", ");
        System.out.println();

        System.out.println("Number of students in course2: " + course2.getNumberOfStudents());
    }
}
```
The Course class is implemented in Listing 9.6. It uses an array to store the students for the course. For simplicity, assume that the maximum course enrollment is 100. The array is created using `new String[100]` in line 3. The `addStudent` method (line 10) adds a student to the array. Whenever a new student is added to the course, `numberOfStudents` is increased (line 11). The `getStudents` method returns the array. The `dropStudent` method (line 27) is left as an exercise.

**LISTING 9.6 Course.java**

```java
public class Course {
    private String courseName;
    private String[] students = new String[100];
    private int numberOfStudents;

    public Course(String courseName) {
        this.courseName = courseName;
    }

    public void addStudent(String student) {
        students[numberOfStudents] = student;
        numberOfStudents++;
    }

    public String[] getStudents() {
        return students;
    }

    public int getNumberOfStudents() {
        return numberOfStudents;
    }

    public String getCourseName() {
        return courseName;
    }

    public void dropStudent(String student) {
        // Left as an exercise in Exercise 9.9
    }
}
```

The array size is fixed to be 100 (line 3) in Listing 9.6. You can improve it to automatically increase the array size in Exercise 9.9.

When you create a Course object, an array object is created. A Course object contains a reference to the array. For simplicity, you can say that the Course object contains the array.

The user can create a Course and manipulate it through the public methods `addStudent`, `dropStudent`, `getNumberOfStudents`, and `getStudents`. However, the user doesn’t need to know how these methods are implemented. The Course class encapsulates the internal implementation. This example uses an array to store students. You may use a different data structure to store students. The program that uses Course does not need to change as long as the contract of the public methods remains unchanged.
9.8 Designing a Class for Stacks

Recall that a stack is a data structure that holds objects in a last-in, first-out fashion, as shown in Figure 9.7.

![Figure 9.7] A stack holds objects in a last-in, first-out fashion.

Stacks have many applications. For example, the compiler uses a stack to process method invocations. When a method is invoked, its parameters and local variables are pushed into a stack. When a method calls another method, the new method’s parameters and local variables are pushed into the stack. When a method finishes its work and returns to its caller, its associated space is released from the stack.

You can define a class to model stacks. For simplicity, assume the stack holds the int values. So, name the stack class `StackOfIntegers`. The UML diagram for the class is shown in Figure 9.8.

![Figure 9.8] The `StackOfIntegers` class encapsulates the stack storage and provides the operations for manipulating the stack.

Suppose that the class is available. Let us write a test program in Listing 9.7 that uses the class to create a stack (line 3), stores ten integers 0, 1, 2, . . . , and 9 (line 6), and displays them in reverse order (line 9).

**Listing 9.7 TestStackOfIntegers.java**

```java
public class TestStackOfIntegers {
    public static void main(String[] args) {
        StackOfIntegers stack = new StackOfIntegers();
        // create a stack
```
9.8 Designing a Class for Stacks

```java
public class StackOfIntegers {
    private int[] elements;
    private int size;
    public static final int DEFAULT_CAPACITY = 16;

    /** Construct a stack with the default capacity 16 */
    public StackOfIntegers() {
        this(DEFAULT_CAPACITY);
    }

    /** Construct a stack with the specified maximum capacity */
    public StackOfIntegers(int capacity) {
```

How do you implement the `StackOfIntegers` class? The elements in the stack are stored in an array named `elements`. When you create a stack, the array is also created. The no-arg constructor creates an array with the default capacity of 19. The variable `size` counts the number of elements in the stack, and `size - 1` is the index of the element at the top of the stack, as shown in Figure 9.9. For an empty stack, `size` is 0.

```
    while (!stack.empty())
        System.out.print(stack.pop() + " ");
    }
}
```

![Image of StackOfIntegers class]

**Figure 9.9** The `StackOfIntegers` class encapsulates the stack storage and provides the operations for manipulating the stack.

The `StackOfIntegers` class is implemented in Listing 9.8. The methods `empty()`, `peek()`, `pop()`, and `getSize()` are easy to implement. To implement `push(int value)`, assign `value` to `elements[size]` if `size < capacity` (line 23). If the stack is full (i.e., `size >= capacity`), create a new array of twice the current capacity (line 18), copy the contents of the current array to the new array (line 19), and assign the reference of the new array to the current array in the stack (line 20). Now you can add the new value to the array (line 23).

---

9.8 Designing a Class for Stacks 317
318 Chapter 9 Thinking in Objects

double the capacity

9.9 Designing the GuessDate Class

Listing 3.8, GuessBirthDate.java, and Listing 6.14, GuessBirthDateUsingArray.java, presented two programs for guessing birth dates. Both programs use the same data developed with the procedural paradigm. The majority of code in these two programs is to define the five sets of data. You cannot reuse the code in these two programs. To make the code reusable, design a class to encapsulate the data, as defined in Figure 9.10.

<table>
<thead>
<tr>
<th>GuessDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>-dates: int[][]</td>
</tr>
<tr>
<td>+getValue(setNo: int, row: int, column: int): int</td>
</tr>
</tbody>
</table>

The static array to hold dates.

Returns a date at the specified row and column in a given set.

Figure 9.10 The GuessDate class defines data for guessing birth dates.

Note that getValue is defined as a static method because it is not dependent on a specific object of the GuessDate class. The GuessDate class encapsulates dates as a private member. The user of this class does not need to know how dates is implemented and does not even need to know the existence of the dates field in the class. All that the user needs to
9.9 Designing the **GuessData Class**

Know is how to use this method to access dates. Suppose this class is available. As shown in Figure 3.4, there are five sets of dates. Invoking `getValue(setNo, row, column)` returns the date at the specified row and column in the given set. For example, `getValue(1, 0, 0)` returns 16.

Assume that the `GuessDate` class is available. Listing 9.9 gives a test program that uses this class.

**LISTING 9.9 UseGuessDateClass.java**

```java
import java.util.Scanner;

public class UseGuessDateClass {
    public static void main(String[] args) {
        int date = 0; // Date to be determined
        int answer;

        // Create a Scanner
        Scanner input = new Scanner(System.in);

        for (int i = 0; i < 5; i++) {
            System.out.println("Is your birth date in Set" + (i + 1) + "?\n");
            for (int j = 0; j < 4; j++) {
                for (int k = 0; k < 4; k++)
                    System.out.print(GuessDate.getValue(i, j, k) + "");
                System.out.println();
            }
            System.out.println("Enter 0 for No and 1 for Yes: ");
            answer = input.nextInt();
            if (answer == 1)
                date += GuessDate.getValue(i, 0, 0);
        }
        System.out.println("Your birth date is " + date);
    }
}
```

---

**Interactive Example**

Is your birth date in Set1?
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31
Enter 0 for No and 1 for Yes: 1

Is your birth date in Set2?
8 9 10 11
12 13 14 15
24 25 26 27
28 29 30 31
Enter 0 for No and 1 for Yes: 1

Is your birth date in Set3?
1 3 5 7
9 11 13 15
17 19 21 23
25 27 29 31
Enter 0 for No and 1 for Yes: 0
Since `getValue` is a static method, you don’t need to create an object in order to invoke it. `GuessDate.getValue(i, j, k)` (line 17) returns the date at row `i` and column `k` in Set `i`.

The `GuessDate` class can be implemented in Listing 9.10.

**LISTING 9.10** GuessDate.java

```java
public class GuessDate {
    private final static int[][][] dates = {
        {{16, 17, 18, 19},
         {20, 21, 22, 23},
         {24, 25, 26, 27},
         {28, 29, 30, 31}},
        {{8, 9, 10, 11},
         {12, 13, 14, 15},
         {24, 25, 26, 27},
         {28, 29, 30, 31}},
        {{1, 3, 5, 7},
         {9, 11, 13, 15},
         {17, 19, 21, 23},
         {25, 27, 29, 31}},
        {{2, 3, 6, 7},
         {10, 11, 14, 15},
         {18, 19, 22, 23},
         {26, 27, 30, 31}},
        {{4, 5, 6, 7},
         {12, 13, 14, 15},
         {20, 21, 22, 23},
         {28, 29, 30, 31}}};

    /** Prevent the user from creating objects from GuessDate */
    private GuessDate() {
    }

    /** Return a date at the specified row and column in a given set */
    public static int getValue(int setNo, int k, int j) {
        return dates[setNo][k][j];
    }
}
```

This class uses a three-dimensional array to store dates (lines 2–22). You may use a different data structure (i.e., five two-dimensional arrays for representing five sets of numbers). The
implementation of the `getValue` method will change, but the program that uses `GuessDate` does not need to change as long as the contract of the public method `getValue` remains unchanged. This shows the benefit of data encapsulation.

The class defines a private no-arg constructor (line 25) to prevent the user from creating objects for this class. Since all methods are static in this class, there is no need to create objects from this class.

**Key Terms**

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>immutable class</td>
<td>304</td>
</tr>
<tr>
<td>immutable object</td>
<td>304</td>
</tr>
<tr>
<td>stack</td>
<td>316</td>
</tr>
<tr>
<td>this keyword</td>
<td>306</td>
</tr>
</tbody>
</table>

**Chapter Summary**

■ An immutable object cannot be modified once it is created. To prevent users from modifying an object, you may declare immutable classes.

■ The scope of instance and static variables is the entire class, regardless of where the variables are declared. Instance and static variables can be declared anywhere in the class. For consistency, they are declared at the beginning of the class.

■ The keyword `this` can be used to refer to the calling object. It can also be used inside a constructor to invoke another constructor of the same class.

■ The procedural paradigm focuses on designing methods. The object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

**Review Questions**

**Section 9.2 Immutable Objects and Classes**

9.1 If a class contains only private data fields and no set methods, is the class immutable?

9.2 If all the data fields in a class are private and primitive type, and the class contains no `set` methods, is the class immutable?

9.3 Is the following class immutable?

```java
class A {
    private int[] values;
    public int[] getValues() {
        return values;
    }
}
```

9.4 If you redefine the `Loan` class in the preceding chapter without `set` methods, is the class immutable?

**Section 9.3 The Scope of Variables**

9.5 What is the output of the following program?
Section 9.4 The this Reference

9.6 Describe the role of the this keyword. What is wrong in the following code?

```java
public class C {
    int p;

    public C() {
        System.out.println("C's no-arg constructor invoked");
        this(0);
    }

    public C(int p) {
        p = p;
    }

    public void setP(int p) {
        p = p;
    }
}
```

Programming Exercises

9.1* (The Time class) Design a class named Time. The class contains:

- Data fields hour, minute, and second that represent a time.
- A no-arg constructor that creates a Time object for the current time. (The data fields value will represent the current time.)
- A constructor that constructs a Time object with a specified elapsed time since midnight, Jan 1, 1970, in milliseconds. (The data fields value will represent this time.)
- Three get methods for the data fields hour, minute, and second, respectively.

Draw the UML diagram for the class. Implement the class. Write a test program that creates two Time objects (using new Time() and new Time(555550000)) and display their hour, minute, and second.

(Hint: The current time can be obtained using System.currentTimeMillis(), as shown in Listing 2.9, ShowCurrentTime.java. The other constructor sets the hour, minute, and second for the specified elapsed time. For example, if the elapsed time is 555550000 milliseconds, the hour is 10, the minute is 19, and the second is 10.)
9.2 *(The BMI class) Add the following new constructor in the BMI class:

```java
/** Construct a BMI with the specified name, age, weight, *
 * feet and inches */

public BMI(String name, int age, double weight, double feet,
           double inches)
```

9.3 *(The MyInteger class) Design a class named MyInteger. The class contains:

- An int data field named value that stores the int value represented by this object.
- A constructor that creates a MyInteger object for the specified int value.
- A get method that return the int value.
- Methods isEven(), isOdd(), and isPrime() that return true if the value is even, odd, or prime, respectively.
- Static methods isEven(int), isOdd(int), and isPrime(int) that return true if the specified value is even, odd, or prime, respectively.
- Static methods isEven(MyInteger), isOdd(MyInteger), and isPrime(MyInteger) that return true if the specified value is even, odd, or prime, respectively.
- Methods equals(int) and equals(MyInteger) that return true if the value in the object is equal to the specified value.
- A static method parseInt(char[]) that converts an array of numeric characters to an int value.

Draw the UML diagram for the class. Implement the class. Write a client program that tests all methods in the class.

9.4 *(The MyPoint class) Design a class named MyPoint to represent a point with x- and y-coordinates. The class contains:

- Two data fields x and y that represent the coordinates with get methods.
- A no-arg constructor that creates a point (0, 0).
- A constructor that constructs a point with specified coordinates.
- Two get methods for data fields x and y, respectively.
- A method named distance that returns the distance from this point to another point of the MyPoint type.
- A method named distance that returns the distance from this point to another point with specified x and y-coordinates.

Draw the UML diagram for the class. Implement the class. Write a test program that creates two points (0, 0) and (10, 30.5) and displays the distance between them.

9.5* *(Displaying the prime factors) Write a program that receives a positive integer and displays all its smallest factors in decreasing order. For example, if the integer is 120, the smallest factors are displayed as 5, 3, 2, 2, 2. Use the StackOfIntegers class to store the factors (e.g., 2, 2, 2, 3, 5) and retrieve and display them in reverse order.

9.6* *(Displaying the prime numbers) Write a program that displays all the prime numbers less than 120 in decreasing order. Use the StackOfIntegers class to store the prime numbers (e.g., 2, 3, 5, ...) and retrieve and display them in reverse order.

9.7** *(Game: ATM machine) Use the Account class created in Exercise 7.3 to simulate an ATM machine. Create ten accounts in an array with id 0, 1, ..., 9, and initial
balance $100. The system prompts the user to enter an id. If the id is entered incorrectly, ask the user to enter a correct id. Once an id is accepted, the main menu is displayed as shown in the sample output. You can enter a choice 1 for viewing the current balance, 2 for withdrawing money, 3 for depositing money, and 4 for exiting the main menu. Once you exit, the system will prompt for an id again. So, once the system starts, it will not stop.
9.8*** *(Financial: the Tax class)* Exercise 6.30 writes a program for computing taxes using arrays. Design a class named Tax to contain the following instance data fields:

- **int filingStatus**: One of the four tax filing statuses: 0—single filer, 1—married filing jointly, 2—married filing separately, and 3—head of household. Use the public static constants SINGLE_FILER (0), MARRIED_JOINTLY (1), MARRIED_SEPARATELY (2), HEAD_OF_HOUSEHOLD (3) to represent the status.
- **int[][] brackets**: Stores the tax brackets for each filing status.
- **double[] rates**: Stores tax rates for each bracket.
- **double taxableIncome**: Stores the taxable income.

Provide the `get` and `set` methods for each data field and the `getTax()` method that returns the tax. Also provide a no-arg constructor and the constructor `Tax(filingStatus, brackets, rates, taxableIncome)`.

Draw the UML diagram for the class. Implement the class. Write a test program that uses the Tax class to print the 2001 and 2002 tax tables for taxable income from $50,000 to $60,000 with intervals of $1,000 for all four statuses. The tax rates for the year 2002 were given in Table 3.7. The tax rates for 2001 are shown in Table 9.1.

**Table 9.1** 2001 United States Federal Personal Tax Rates

<table>
<thead>
<tr>
<th>Tax rate</th>
<th>Single filers</th>
<th>Married filing jointly or qualifying widow(er)</th>
<th>Married filing separately</th>
<th>Head of household</th>
</tr>
</thead>
<tbody>
<tr>
<td>15%</td>
<td>Up to $27,050</td>
<td>Up to $45,200</td>
<td>Up to $22,600</td>
<td>Up to $36,250</td>
</tr>
<tr>
<td>27.5%</td>
<td>$27,051–$65,550</td>
<td>$45,201–$109,250</td>
<td>$22,601–$54,625</td>
<td>$36,251–$93,650</td>
</tr>
<tr>
<td>30.5%</td>
<td>$65,551–$136,750</td>
<td>$109,251–$166,500</td>
<td>$54,626–$83,250</td>
<td>$93,651–$151,560</td>
</tr>
<tr>
<td>39.1%</td>
<td>$297,351 or more</td>
<td>$297,351 or more</td>
<td>$148,676 or more</td>
<td>$297,351 or more</td>
</tr>
</tbody>
</table>

9.9** *(The Course class)* Revise the Course class as follows:

- The array size is fixed in Listing 9.8. Improve it to automatically increase the array size by creating a new larger array and copying the contents of the current array to it.
- Implement the `dropStudent` method.
- Add a new method named `clear()` that removes all students from the course.

Write a test program that creates a course, adds three students, removes one, and displays the students in the course.

9.10* *(Game: The GuessDate class)* Modify the GuessDate class in Listing 9.10. Instead of representing dates in a three-dimensional array, use five two-dimensional arrays to represent the five sets of numbers. So you need to declare:

```java
private static int[][] set1 = {{16, 17, 18, 19}, ... };
private static int[][] set2 = {{8, 9, 10, 11}, ... };
private static int[][] set3 = {{1, 3, 5, 7}, ... };
private static int[][] set4 = {{2, 3, 6, 7}, ... };
private static int[][] set5 = {{4, 5, 6, 7}, ... };
```
9.11* (Geometry: The Circle2D class) Define the Circle2D class that contains:

- Two double data fields named x and y that specify the center of the circle with get methods.
- A data field radius with a get method.
- A no-arg constructor that creates a default circle with (0, 0) for (x, y) and 1 for radius.
- A constructor that creates a circle with the specified x, y, and radius.
- A method getArea() that returns the area of the circle.
- A method getPerimeter() that returns the perimeter of the circle.
- A method contains(double x, double y) that returns true if the specified point (x, y) is inside this circle. See Figure 9.11(a).
- A method contains(Circle2D circle) that returns true if the specified circle is inside this circle. See Figure 9.11(b).
- A method overlaps(Circle2D circle) that returns true if the specified circle overlaps with this circle. See Figure 9.11(c).

![Diagram of circles](image)

**Figure 9.11** (a) A point is inside the circle. (b) A circle is inside another circle. (c) A circle overlaps another circle.

Draw the UML diagram for the class. Implement the class. Write a test program that creates a Circle2D object c1 (new Circle2D(2, 2, 5.5)), displays its area and perimeter, and displays the result of c1.contains(3, 3), c1.contains(new Circle2D(4, 5, 10.5)), and c1.overlaps(new Circle2D(3, 5, 2.3)).

9.12* (Geometry: The Rectangle2D class) Define the Rectangle2D class that contains:

- Two double data fields named x and y that specify the center of the rectangle with get and set methods. (Assume that the rectangle sides are parallel to x- or y- axes.)
- The data fields width and height with get and set methods.
- A no-arg constructor that creates a default rectangle with (0, 0) for (x, y) and 1 for both width and height.
- A constructor that creates a rectangle with the specified x, y, and radius.
- A method getArea() that returns the area of the rectangle.
- A method getPerimeter() that returns the perimeter of the rectangle.
- A method contains(double x, double y) that returns true if the specified point (x, y) is inside this rectangle. See Figure 9.12(a).
- A method contains(Rectangle2D r) that returns true if the specified rectangle is inside this rectangle. See Figure 9.12(b).
- A method overlaps(Rectangle2D r) that returns true if the specified rectangle overlaps with this rectangle. See Figure 9.12(c).
Programming Exercises 327

(a) A point is inside the rectangle. (b) A rectangle is inside another rectangle. (c) A rectangle overlaps another rectangle.

**Figure 9.12**

9.13*** (Geometry: The Triangle2D class) Define the Triangle2D class that contains:

- Three points named \( p_1 \), \( p_2 \), and \( p_3 \) with the type MyPoint with get and set methods. MyPoint is defined in Exercise 9.4.
- A no-arg constructor that creates a default triangle with points \( (0, 0) \), \( (1, 1) \), and \( (2, 5) \).
- A constructor that creates a triangle with the specified points.
- A method `getArea()` that returns the area of the triangle.
- A method `getPerimeter()` that returns the perimeter of the triangle.
- A method `contains(MyPoint p)` that returns true if the specified point \( p \) is inside this triangle. See Figure 9.13(a).
- A method `contains(Triangle2D t)` that returns true if the specified triangle is inside this triangle. See Figure 9.13(b).
- A method `overlaps(Triangle2D t)` that returns true if the specified triangle overlaps with this triangle. See Figure 9.13(c).

**Figure 9.13**

(a) A point is inside the triangle. (b) A triangle is inside another triangle. (c) A triangle overlaps another triangle.
whether two line segments intersect and whether a line contains a point, etc. Please see the Java API for more information on `Line2D`. To detect whether a point is inside a triangle, draw three dashed lines, as shown in Figure 9.14. If the point is inside a triangle, each dashed line should intersect a side only once. If a dashed line intersects a side twice, then the point must be outside the triangle.)

![Figure 9.14](image)

**Figure 9.14** (a) A point is inside the triangle. (b) A point is outside the triangle.