10.1 Introduction

The focus of this chapter is on class design and explores the differences between procedural programming and object-oriented programming.

The preceding two chapters introduced objects and classes. You learned how to define classes, create objects, and use objects from several classes in the Java API (e.g., `Date`, `Random`, `String`, `StringBuilder`, and `Scanner`). This book's approach is to teach problem solving and fundamental programming techniques before object-oriented programming. This chapter shows how procedural and object-oriented programming differ. You will see the benefits of object-oriented programming and learn to use it effectively.

We will use several examples to illustrate the advantages of the object-oriented approach. The examples involve designing new classes and using them in applications. We first introduce some language features supporting these examples.

10.2 Immutable Objects and Classes

You can define immutable classes to create immutable objects. The contents of immutable objects cannot be changed.

VideoNote

Immutable objects and `this` keyword

immutable object

immutable class

Normally, you create an object and allow its contents to be changed later. However, occasionally it is desirable to create an object whose contents cannot be changed once the object has been 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 `CircleWithPrivateDataFields` class in Listing 8.9, the class would be immutable, because radius is private and cannot be changed without a `set` method.

If a class is immutable, then all its data fields must be private and it cannot contain public `set` methods for any data fields. A class with all private data fields and no mutators is not necessarily immutable. For example, the following `Student` class has all private data fields and no `set` methods, but it is not an immutable class.

`Student` class

```
1  public class Student {
2    private int id;
3    private String name;
4    private java.util.Date dateCreated;
5
6    public Student(int ssn, String newName) {
7      id = ssn;
8      name = newName;
9      dateCreated = new java.util.Date();
10   }
11
12   public int getId() {
13     return id;
14   }
15
16   public String getName() {
17     return name;
18   }
19
20   public java.util.Date getDateCreated() {
21     return dateCreated;
22   }
23 }
```

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.

```
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!
  }
}
```

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

- All data fields must be private.
- There can't be any mutator methods for data fields.
- No accessor methods can return a reference to a data field that is mutable.

Interested readers may refer to Supplement III.AB for an extended discussion on immutable objects.

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

10.2 If all the data fields in a class are private and primitive types, and the class doesn't contain any `set` methods, is the class immutable?

10.3 Is the following class immutable?

```
public class A {
  private int[] values;

  public int[] getValues() {
    return values;
  }
}
```

Check Point

MyProgrammingLab™

10.3 The Scope of Variables

The scope of instance and static variables is the entire class, regardless of where the variables are declared.

Key Point

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 appear in any order in the class, as shown in Figure 10.1a. The exception is when a data field is initialized based on a reference to another data field. In such cases, the

class's variables

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

  private double radius = 1;
}
```

(a) The variable `radius` and method `findArea()` can be declared in any order.

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

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

FIGURE 10.1 Members of a class can be declared in any order, with one exception.

other data field must be declared first, as shown in Figure 10.1b. For consistency, this book declares data fields at the beginning of the class.

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.

hidden variables

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 both as an instance variable and as a local variable in the method.

```
public class F {
  private int x = 0; // Instance variable
  private int y = 0;

  public F() {
  }

  public 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 `F`? 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 it is also declared 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 `y` is accessible inside the method.

Tip

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

MyProgrammingLab™

10.4 What is the output of the following program?

```
public class Test {
  private static int i = 0;
  private static int j = 0;

  public static void main(String[] args) {
    int i = 2;
    int k = 3;

    {
      int j = 3;
      System.out.println("i + j is " + i + j);
    }

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

10.4 The this Reference

The keyword `this` *refers to the object itself. It can also be used inside a constructor to invoke another constructor of the same class.*

`this` keyword

The `this` *keyword* is the name of a reference that an object can use to refer to itself. You can use the `this` keyword to refer to the object's instance members. For example, the following code in (a) uses `this` to reference the object's `radius` and invokes its `getArea()` method explicitly. The `this` reference is normally omitted, as shown in (b). However, the `this` reference is needed to reference hidden data fields or invoke an overloaded constructor.

```
public class Circle {
  private double radius;

  ...

  public double getArea() {
    return this.radius * this.radius
        * Math.PI;
  }

  public String toString() {
    return "radius: " + this.radius
        + "area: " + this.getArea();
  }
}
```

(a)

Equivalent

```
public class Circle {
  private double radius;

  ...

  public double getArea() {
    return radius * radius * Math.PI;
  }

  public String toString() {
    return "radius: " + radius
        + "area: " + getArea();
  }
}
```

(b)

10.4.1 Using this to Reference Hidden Data Fields

hidden data fields

The `this` keyword can be used to reference a class's *hidden data fields*. For example, a data-field name is often used as the parameter name in a `set` method for the data field. In this case, the data field is hidden in the `set` method. You need to reference the hidden data-field 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 10.2a.

```
public class F {
  private int i = 5;
  private static double k = 0;

  public void setI(int i) {
    this.i = i;
  }

  public static void setK(double k) {
    F.k = k;
  }

  // Other methods omitted
}
```

(a)

```
Suppose that f1 and f2 are two objects of F.

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

Invoking F.setK(33) is to execute
   F.k = 33. setK is a static method
```

(b)

FIGURE 10.2 The keyword `this` refers to the calling object that invokes the method.

The `this` keyword gives us a way to refer to the object that invokes an instance method. To invoke `f1.setI(10)`, `this.i = i` is executed, which assigns the value of parameter `i` to the data field `i` of this calling object `f1`. The keyword `this` refers to the object that invokes

the instance method `setI`, as shown in Figure 10.2b. The line `F.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.

10.4.2 Using `this` to Invoke a Constructor

The `this` keyword can be used to invoke another constructor of the same class. For example, you can rewrite the `Circle` class as follows:

```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  ...
}
```

The `this` keyword is used to reference the hidden data field radius of the object being constructed.

The `this` keyword is used to invoke another constructor.

The line `this(1.0)` in the second constructor invokes the first constructor with a `double` value argument.

Note

Java requires that the `this(arg-list)` statement appear first in the constructor before any other executable statements.

Tip

If a class has multiple constructors, it is better to implement them using `this(arg-list)` as much as possible. In general, a constructor with no or fewer arguments can invoke a constructor with more arguments using `this(arg-list)`. This syntax often simplifies coding and makes the class easier to read and to maintain.

MyProgrammingLab™

10.5 Describe the role of the `this` keyword.

10.6 What is wrong in the following code?

```
1  public class C {
2    private int p;
3
4    public C() {
5      System.out.println("C's no-arg constructor invoked");
6      this(0);
7    }
8
9    public C(int p) {
10     p = p;
11   }
12
13   public void setP(int p) {
14     p = p;
15   }
16 }
```

10.7 What is wrong in the following code?

```
public class Test {
  private int id;

  public void m1() {
    this.id = 45;
  }

  public void m2() {
    Test.id = 45;
  }
}
```

10.5 Class Abstraction and Encapsulation

Class abstraction *is the separation of class implementation from the use of a class. The details of implementation are encapsulated and hidden from the user. This is known as* class encapsulation.

In Chapter 5, you learned about method abstraction and used it in stepwise refinement. Java provides many levels of abstraction, and *class abstraction* separates class implementation from how the class is used. The creator of a class describes the functions of the class and lets the user know how the class 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 10.3, 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 called *class encapsulation*. For example, you can create a `Circle` object and find the area of the circle without knowing how the area is computed. For this reason, a class is also known as an *abstract data type* (ADT).

class abstraction

class's contract

class encapsulation

abstract data type

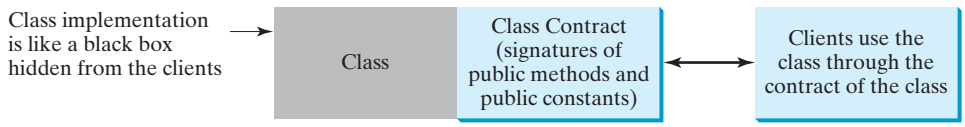


FIGURE 10.3 Class abstraction separates class implementation from the use of the class.

Class abstraction and encapsulation are two sides of the same coin. Many real-life examples illustrate the concept of class abstraction. Consider, for instance, building a computer system. Your personal computer has many components—a CPU, memory, 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, you need know only how each component is used and how it interacts with the others. You don't need to know how the components 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 such as fan size and speed and methods such as start and stop. 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. The interest rate, loan amount, and loan period are its data properties, and

VideoNote
The Loan class

computing the 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 2.8, ComputeLoan.java, presented a program for computing loan payments. That program cannot be reused in other programs because the code for computing the payments is in the `main` method. One way to fix this problem is to define static methods for computing the monthly payment and total payment. However, this solution has limitations. Suppose you wish to associate a date with the loan. There is no good way to tie a date with a loan without using objects. The traditional procedural programming paradigm is action-driven, and data are separated from actions. The object-oriented programming paradigm focuses on objects, and actions are defined along with the data in objects. To tie a date with a loan, you can define a loan class with a date along with other of the loan's properties as data fields. A loan object now contains data and actions for manipulating and processing data, and the loan data and actions are integrated in one object. Figure 10.4 shows the UML class diagram for the **Loan** class.

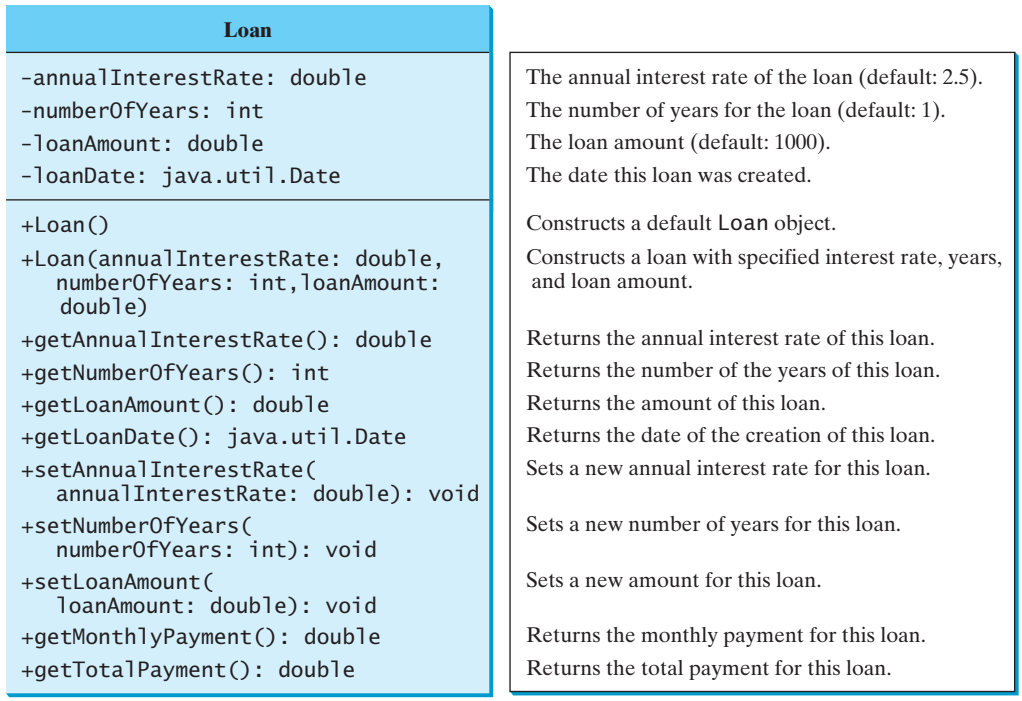


FIGURE 10.4 The **Loan** class models the properties and behaviors of loans.

The UML diagram in Figure 10.4 serves as the contract for the **Loan** class. Throughout this book, you will play the roles of both class user and class developer. Remember that a class user can use the class without knowing how the class is implemented.

Assume that the **Loan** class is available. The program in Listing 10.1 uses that class.

LISTING 10.1 TestLoanClass.java

```
1  import java.util.Scanner;
2
3  public class TestLoanClass {
4    /** Main method */
5    public static void main(String[] args) {
```

```
6       // Create a Scanner
7       Scanner input = new Scanner(System.in);
8
9       // Enter annual interest rate
10      System.out.print(
11        "Enter annual interest rate, for example, 8.25: ");
12      double annualInterestRate = input.nextDouble();
13
14      // Enter number of years
15      System.out.print("Enter number of years as an integer: ");
16      int numberOfYears = input.nextInt();
17
18      // Enter loan amount
19      System.out.print("Enter loan amount, for example, 120000.95: ");
20      double loanAmount = input.nextDouble();
21
22      // Create a Loan object
23      Loan loan =
24        new Loan(annualInterestRate, numberOfYears, loanAmount);          create Loan object
25
26      // Display loan date, monthly payment, and total payment
27      System.out.printf("The loan was created on %s\n" +
28        "The monthly payment is %.2f\nThe total payment is %.2f\n",
29        loan.getLoanDate().toString(), loan.getMonthlyPayment(),          invoke instance method
30        loan.getTotalPayment());                                          invoke instance method
31    }
32  }
```

```
Enter annual interest rate, for example, 8.25: 2.5 Enter
Enter number of years as an integer: 5 Enter
Enter loan amount, for example, 120000.95: 1000 Enter
The loan was created on Sat Jun 16 21:12:50 EDT 2012
The monthly payment is 17.75
The total payment is 1064.84
```

The `main` method reads the interest rate, the payment period (in years), and the loan amount; creates a `Loan` object; and then obtains the monthly payment (line 29) and the total payment (line 30) using the instance methods in the `Loan` class.

The `Loan` class can be implemented as in Listing 10.2.

LISTING 10.2 Loan.java

```
1   public class Loan {
2     private double annualInterestRate;
3     private int numberOfYears;
4     private double loanAmount;
5     private java.util.Date loanDate;
6
7     /** Default constructor */
8     public Loan() {                                                       no-arg constructor
9       this(2.5, 1, 1000);
10    }
11
12    /** Construct a loan with specified annual interest rate,
13        number of years, and loan amount
14      */
```

constructor

```
15    public Loan(double annualInterestRate, int numberOfYears,
16        double loanAmount) {
17      this.annualInterestRate = annualInterestRate;
18      this.numberOfYears = numberOfYears;
19      this.loanAmount = loanAmount;
20      loanDate = new java.util.Date();
21    }
22
23    /** Return annualInterestRate */
24    public double getAnnualInterestRate() {
25      return annualInterestRate;
26    }
27
28    /** Set a new annualInterestRate */
29    public void setAnnualInterestRate(double annualInterestRate) {
30      this.annualInterestRate = annualInterestRate;
31    }
32
33    /** Return numberOfYears */
34    public int getNumberOfYears() {
35      return numberOfYears;
36    }
37
38    /** Set a new numberOfYears */
39    public void setNumberOfYears(int numberOfYears) {
40      this.numberOfYears = numberOfYears;
41    }
42
43    /** Return loanAmount */
44    public double getLoanAmount() {
45      return loanAmount;
46    }
47
48    /** Set a new loanAmount */
49    public void setLoanAmount(double loanAmount) {
50      this.loanAmount = loanAmount;
51    }
52
53    /** Find monthly payment */
54    public double getMonthlyPayment() {
55      double monthlyInterestRate = annualInterestRate / 1200;
56      double monthlyPayment = loanAmount * monthlyInterestRate / (1 -
57        (1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12)));
58      return monthlyPayment;
59    }
60
61    /** Find total payment */
62    public double getTotalPayment() {
63      double totalPayment = getMonthlyPayment() * numberOfYears * 12;
64      return totalPayment;
65    }
66
67    /** Return loan date */
68    public java.util.Date getLoanDate() {
69      return loanDate;
70    }
71  }
```

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 the monthly payment and the total payment. You can construct a `Loan` object by using the no-arg constructor or the constructor 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 the 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 and methods.

Important Pedagogical Tip

Use the UML diagram for the `Loan` class shown in Figure 10.4 to write a test program that uses the `Loan` class even though you 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 this book.
- It is easier to learn how to implement a class if you are familiar with it by using the class.

For all the class examples from now on, create an object from the class and try using its methods before turning your attention to its implementation.

10.8 If you redefine the `Loan` class in Listing 10.2 without `set` methods, is the class immutable?

MyProgrammingLab™

10.6 Object-Oriented Thinking

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.

Chapters 1–7 introduced fundamental programming techniques for problem solving using loops, methods, and arrays. Knowing 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 these improvements, you will gain insight into the differences between procedural and object-oriented programming and see the benefits of developing reusable code using objects and classes.

Listing 3.5, ComputeAndInterpretBMI.java, presented a program for computing body mass index. The code cannot be reused in other programs, because the code is in the `main` method. To make it reusable, define a static method to compute body mass index as follows:

```
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 could declare separate variables to store these values, but these values would not be tightly coupled. The ideal way to couple them is to create an object that contains them all. Since these values are tied to individual objects, they should be stored in instance data fields. You can define a class named `BMI` as shown in Figure 10.5.

VideoNote
The BMI class

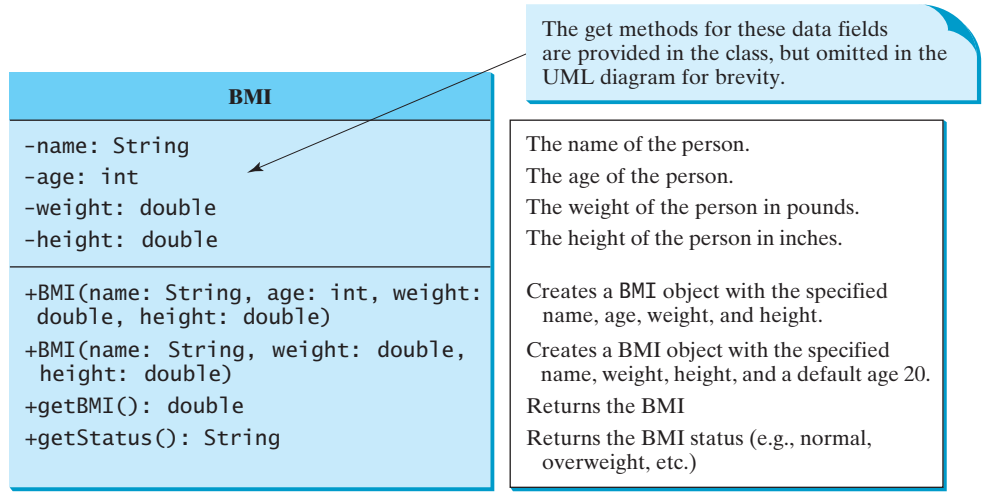


FIGURE 10.5 The `BMI` class encapsulates BMI information.

Assume that the `BMI` class is available. Listing 10.3 gives a test program that uses this class.

LISTING 10.3 UseBMIClass.java

```
1  public class UseBMIClass {
2    public static void main(String[] args) {
3      BMI bmi1 = new BMI("Kim Yang", 18, 145, 70);
4      System.out.println("The BMI for " + bmi1.getName() + " is "
5        + bmi1.getBMI() + " " + bmi1.getStatus());
6
7      BMI bmi2 = new BMI("Susan King", 215, 70);
8      System.out.println("The BMI for " + bmi2.getName() + " is "
9        + bmi2.getBMI() + " " + bmi2.getStatus());
10   }
11 }
```

create an object
invoke instance method

create an object
invoke instance method

```
The BMI for Kim Yang is 20.81 Normal
The BMI for Susan King is 30.85 Obese
```

Line 3 creates the object `bmi1` for `Kim Yang` and line 7 creates the object `bmi2` for `Susan 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 10.4.

LISTING 10.4 BMI.java

```
1  public class BMI {
2    private String name;
3    private int age;
4    private double weight; // in pounds
5    private double height; // in inches
6    public static final double KILOGRAMS_PER_POUND = 0.45359237;
```

```
 7    public static final double METERS_PER_INCH = 0.0254;
 8
 9    public BMI(String name, int age, double weight, double height) {      constructor
10      this.name = name;
11      this.age = age;
12      this.weight = weight;
13      this.height = height;
14    }
15
16    public BMI(String name, double weight, double height) {               constructor
17      this(name, 20, weight, height);
18    }
19
20    public double getBMI() {                                              getBMI
21      double bmi = weight * KILOGRAMS_PER_POUND /
22        ((height * METERS_PER_INCH) * (height * METERS_PER_INCH));
23      return Math.round(bmi * 100) / 100.0;
24    }
25
26    public String getStatus() {                                           getStatus
27      double bmi = getBMI();
28      if (bmi < 18.5)
29        return "Underweight";
30      else if (bmi < 25)
31        return "Normal";
32      else if (bmi < 30)
33        return "Overweight";
34      else
35        return "Obese";
36    }
37
38    public String getName() {
39      return name;
40    }
41
42    public int getAge() {
43      return age;
44    }
45
46    public double getWeight() {
47      return weight;
48    }
49
50    public double getHeight() {
51      return height;
52    }
53  }
```

The mathematical formula for computing the BMI using weight and height is given in Section 3.9. 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 Section 3.9.

procedural vs. object-oriented paradigms

This example demonstrates the advantages of 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 in 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.

MyProgrammingLab™

10.9 Is the `BMI` class defined in Listing 10.4 immutable?

10.7 Object Composition

An object can contain another object. The relationship between the two is called composition.

In Listing 10.2, you defined the `Loan` class to contain a `Date` data field. The relationship between `Loan` and `Date` is composition. In Listing 10.4, you defined the `BMI` class to contain a `String` data field. The relationship between `BMI` and `String` is composition.

aggregation
has-a relationship

Composition is actually a special case of the aggregation relationship. Aggregation models *has-a* relationships and represents an ownership relationship between two objects. The owner object is called an *aggregating object* and its class an *aggregating class*. The subject object is called an *aggregated object* and its class an *aggregated class*.

composition

An object may be owned by several other aggregating objects. If an object is exclusively owned by an aggregating object, the relationship between them is referred to as *composition*. For example, "a student has a name" is a composition relationship between the `Student` class and the `Name` class, whereas "a student has an address" is an aggregation relationship between the `Student` class and the `Address` class, because an address may be shared by several students. In UML notation, a filled diamond is attached to an aggregating class (e.g., `Student`) to denote the composition relationship with an aggregated class (e.g., `Name`), and an empty diamond is attached to an aggregating class (e.g., `Student`) to denote the aggregation relationship with an aggregated class (e.g., `Address`), as shown in Figure 10.6.

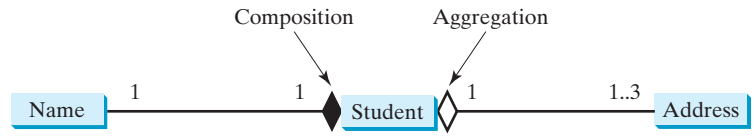


FIGURE 10.6 A student has a name and an address.

multiplicity

Each class involved in a relationship may specify a *multiplicity*. A multiplicity could be a number or an interval that specifies how many of the class's objects are involved in the relationship. The character `*` means an unlimited number of objects, and the interval `m..n` means that the number of objects should be between `m` and `n`, inclusive. In Figure 10.6, each student has only one address, and each address may be shared by up to `3` students. Each student has one name, and a name is unique for each student.

An aggregation relationship is usually represented as a data field in the aggregating class. For example, the relationship in Figure 10.6 can be represented as follows:

```
public class Name {
  ...
}
```

Aggregated class

```
public class Student {
  private Name name;
  private Address address;

  ...
}
```

Aggregating class

```
public class Address {
  ...
}
```

Aggregated class

Aggregation may exist between objects of the same class. For example, a person may have a supervisor. This is illustrated in Figure 10.7.

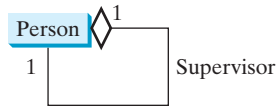


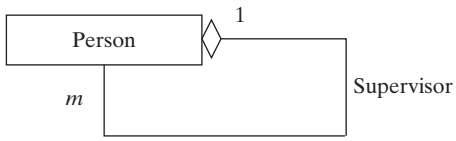
FIGURE 10.7 A person may have a supervisor.

In the relationship "a person has a supervisor," as shown in Figure 10.7, a supervisor can be represented as a data field in the `Person` class, as follows:

```
public class Person {
  // The type for the data is the class itself
  private Person supervisor;

  ...
}
```

If a person can have several supervisors, as shown in Figure 10.8a, you may use an array to store supervisors, as shown in Figure 10.8b.



(a)

```
public class Person {
  ...
  private Person[] supervisors;
}
```

(b)

FIGURE 10.8 A person can have several supervisors.

Note

Since aggregation and composition relationships are represented using classes in the same way, many texts don't differentiate them and call both compositions. We will do the same in this book for simplicity.

aggregation or composition

10.10 What is an aggregation relationship between two objects?

10.11 What is a composition relationship between two objects?

MyProgrammingLab™

10.8 Case Study: Designing the `Course` Class

This section designs a class for modeling courses.

This book's philosophy is *teaching by example and learning by doing*. The book provides a wide variety of examples to demonstrate object-oriented programming. This section and the next two offer additional examples on designing classes.

Suppose you need to process course information. Each course has a name and has students enrolled. 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 10.9.

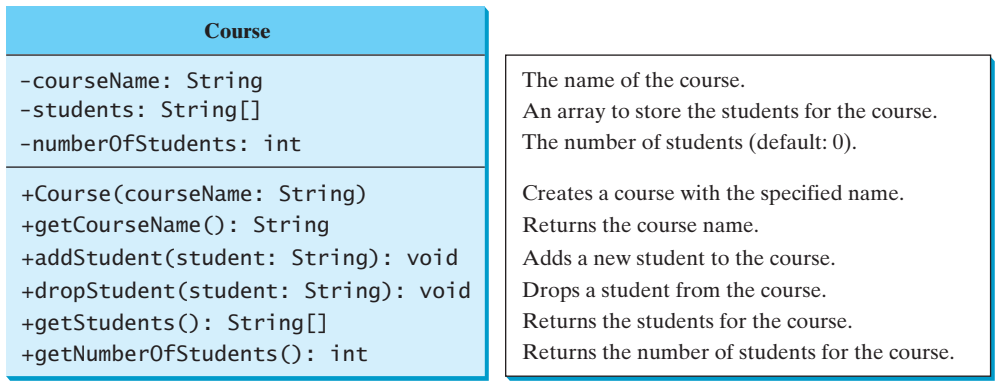


FIGURE 10.9 The `Course` class models the courses.

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 in the course using the `getStudents()` method. Suppose the class is available; Listing 10.5 gives a test class that creates two courses and adds students to them.

LISTING 10.5 TestCourse.java

```
1  public class TestCourse {
2    public static void main(String[] args) {
3      Course course1 = new Course("Data Structures");
4      Course course2 = new Course("Database Systems");
5
6      course1.addStudent("Peter Jones");
7      course1.addStudent("Kim Smith");
8      course1.addStudent("Anne Kennedy");
9
10     course2.addStudent("Peter Jones");
11     course2.addStudent("Steve Smith");
12
13     System.out.println("Number of students in course1: "
14       + course1.getNumberOfStudents());
15     String[] students = course1.getStudents();
16     for (int i = 0; i < course1.getNumberOfStudents(); i++)
17       System.out.print(students[i] + ", ");
18
19     System.out.println();
20     System.out.print("Number of students in course2: "
```

create a course

add a student

number of students

return students

```
21        + course2.getNumberOfStudents());
22    }
23  }
```

```
Number of students in course1: 3
Peter Jones, Kim Smith, Anne Kennedy,
Number of students in course2: 2
```

The **Course** class is implemented in Listing 10.6. It uses an array to store the students in 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 12). The **getStudents** method returns the array. The **dropStudent** method (line 27) is left as an exercise.

LISTING 10.6 Course.java

```
1  public class Course {
2    private String courseName;
3    private String[] students = new String[100];                    create students
4    private int numberOfStudents;
5
6    public Course(String courseName) {                              add a course
7      this.courseName = courseName;
8    }
9
10   public void addStudent(String student) {
11     students[numberOfStudents] = student;
12     numberOfStudents++;
13   }
14
15   public String[] getStudents() {                                 return students
16     return students;
17   }
18
19   public int getNumberOfStudents() {                              number of students
20     return numberOfStudents;
21   }
22
23   public String getCourseName() {
24     return courseName;
25   }
26
27   public void dropStudent(String student) {
28     // Left as an exercise in Programming Exercise 10.9
29   }
30 }
```

The array size is fixed to be **100** (line 3), so you cannot have more than 100 students in the course. You can improve the class by automatically increasing the array size in Programming Exercise 10.9.

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** object and manipulate it through the public methods **addStudent**, **dropStudent**, **getNumberOfStudents**, and **getStudents**. However, the

Key Terms

abstract data type (ADT) 375
aggregation 382
boxing 396
class abstraction 375
class encapsulation 375
class's contract 375
class's variable 371
composition 382
has-a relationship 382
immutable class 370
immutable object 370
multiplicity 382
stack 386
`this` keyword 373
unboxing 396

Chapter Summary

1. Once it is created, an *immutable object* cannot be modified. To prevent users from modifying an object, you can define *immutable classes*.

2. 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 in this book.

3. 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.

4. 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.

5. Many Java methods require the use of objects as arguments. Java offers a convenient way to incorporate, or wrap, a primitive data type into an object (e.g., wrapping `int` into the `Integer` class, and wrapping `double` into the `Double` class).

6. Java can automatically convert a primitive type value to its corresponding wrapper object in the context and vice versa.

7. The `BigInteger` class is useful for computing and processing integers of any size. The `BigDecimal` class can be used to compute and process floating-point numbers with any arbitrary precision.

Test Questions

Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

Programming Exercises

Sections 10.2–10.6

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

- The data fields `hour`, `minute`, and `second` that represent a time.
- A no-arg constructor that creates a `Time` object for the current time. (The values of the data fields will represent the current time.)

- A constructor that constructs a `Time` object with a specified elapsed time since midnight, January 1, 1970, in milliseconds. (The values of the data fields will represent this time.)
- A constructor that constructs a `Time` object with the specified hour, minute, and second.
- Three `get` methods for the data fields `hour`, `minute`, and `second`, respectively.
- A method named `setTime(long elapseTime)` that sets a new time for the object using the elapsed time. For example, if the elapsed time is `555550000` milliseconds, the hour is `10`, the minute is `19`, and the second is `10`.

Draw the UML diagram for the class and then implement the class. Write a test program that creates two `Time` objects (using `new Time()` and `new Time(555550000)`) and displays their hour, minute, and second in the format hour:minute:second.

(*Hint*: The first two constructors will extract the hour, minute, and second from the elapsed time. For the no-arg constructor, the current time can be obtained using `System.currentTimeMillis()`, as shown in Listing 2.6, ShowCurrentTime.java.)

10.2 (*The `BMI` class*) Add the following new constructor in the `BMI` class:

```
/** Construct a BMI with the specified name, age, weight,
 * feet, and inches
 */
public BMI(String name, int age, double weight, double feet,
  double inches)
```

10.3 (*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 returns the `int` value.
- The methods `isEven()`, `isOdd()`, and `isPrime()` that return `true` if the value in this object is even, odd, or prime, respectively.
- The static methods `isEven(int)`, `isOdd(int)`, and `isPrime(int)` that return `true` if the specified value is even, odd, or prime, respectively.
- The static methods `isEven(MyInteger)`, `isOdd(MyInteger)`, and `isPrime(MyInteger)` that return `true` if the specified value is even, odd, or prime, respectively.
- The methods `equals(int)` and `equals(MyInteger)` that return `true` if the value in this object is equal to the specified value.
- A static method `parseInt(char[])` that converts an array of numeric characters to an `int` value.
- A static method `parseInt(String)` that converts a string into an `int` value.

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

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

VideoNote
The `MyPoint` class

- The 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 the 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 and then implement the class. Write a test program that creates the two points (`0`, `0`) and (`10`, `30.5`) and displays the distance between them.

Sections 10.7–10.11

***10.5** (*Displaying the prime factors*) Write a program that prompts the user to enter 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.

***10.6** (*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.

****10.7** (*Game: ATM machine*) Use the `Account` class created in Programming Exercise 8.7 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 run. 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. Thus, once the system starts, it will not stop.

```
Enter an id: 4 Enter

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 1 Enter
The balance is 100.0

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 2 Enter
Enter an amount to withdraw: 3 Enter

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 1 Enter
The balance is 97.0
```

```
Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 3 Enter
Enter an amount to deposit: 10 Enter

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 1 Enter
The balance is 107.0

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 4 Enter

Enter an id:
```

*****10.8** (*Financial: the* `Tax` *class*) Programming Exercise 7.12 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 or qualifying widow(er), `2`—married filing separately, and `3`—head of household. Use the public static constants `SINGLE_FILER` (`0`), `MARRIED_JOINTLY_OR_QUALIFYING_WIDOW(ER)` (`1`), `MARRIED_SEPARATELY` (`2`), `HEAD_OF_HOUSEHOLD` (`3`) to represent the statuses.
- `int[][] brackets`: Stores the tax brackets for each filing status.
- `double[] rates`: Stores the 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 and then implement the class. Write a test program that uses the `Tax` class to print the 2001 and 2009 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 2009 were given in Table 3.2. The tax rates for 2001 are shown in Table 10.1.

****10.9** (*The* `Course` *class*) Revise the `Course` class as follows:

- The array size is fixed in Listing 10.6. 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.

TABLE 10.1 2001 United States Federal Personal Tax Rates

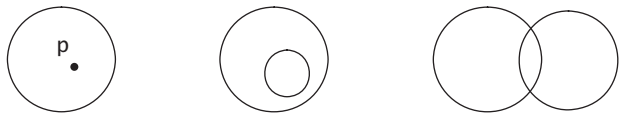
Tax rate	*Single filers*	*Married filing jointly or qualifying widow(er)*	*Married filing separately*	*Head of household*
15%	Up to $27,050	Up to $45,200	Up to $22,600	Up to $36,250
27.5%	$27,051–$65,550	$45,201–$109,250	$22,601–$54,625	$36,251–$93,650
30.5%	$65,551–$136,750	$109,251–$166,500	$54,626–$83,250	$93,651–$151,650
35.5%	$136,751–$297,350	$166,501–$297,350	$83,251–$148,675	$151,651–$297,350
39.1%	$297,351 or more	$297,351 or more	$ 148,676 or more	$297,351 or more

***10.10** (*Game: The* `GuessDate` *class*) Modify the `GuessDate` class in Listing 10.10. Instead of representing dates in a three-dimensional array, use five two-dimensional arrays to represent the five sets of numbers. Thus, you need to declare:

```
private static int[][] set1 = {{1,  3,  5,  7}, ... };
private static int[][] set2 = {{2,  3,  6,  7}, ... };
private static int[][] set3 = {{4,  5,  6,  7}, ... };
private static int[][] set4 = {{8,  9, 10, 11}, ... };
private static int[][] set5 = {{16, 17, 18, 19}, ... };
```

***10.11** (*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 10.15a).
- A method `contains(Circle2D circle)` that returns `true` if the specified circle is inside this circle (see Figure 10.15b).
- A method `overlaps(Circle2D circle)` that returns `true` if the specified circle overlaps with this circle (see Figure 10.15c).



(a) (b) (c)

FIGURE 10.15 (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 and then 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))`.

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

- Three points named `p1`, `p2`, and `p3` of the type `MyPoint` with `get` and `set` methods. `MyPoint` is defined in Exercise 10.4.
- A no-arg constructor that creates a default triangle with the 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 10.16a).
- A method `contains(Triangle2D t)` that returns `true` if the specified triangle is inside this triangle (see Figure 10.16b).
- A method `overlaps(Triangle2D t)` that returns `true` if the specified triangle overlaps with this triangle (see Figure 10.16c).

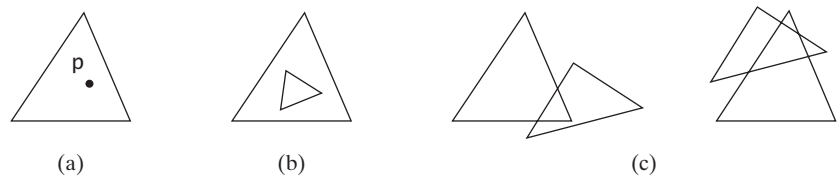


FIGURE 10.16 (a) A point is inside the triangle. (b) A triangle is inside another triangle. (c) A triangle overlaps another triangle.

Draw the UML diagram for the class and then implement the class. Write a test program that creates a `Triangle2D` objects `t1` using the constructor `new Triangle2D(new MyPoint(2.5, 2), new MyPoint(4.2, 3), new MyPoint(5, 3.5))`, displays its area and perimeter, and displays the result of `t1.contains(3, 3)`, `r1.contains(new Triangle2D(new MyPoint(2.9, 2), new MyPoint(4, 1), MyPoint(1, 3.4)))`, and `t1.overlaps(new Triangle2D(new MyPoint(2, 5.5), new MyPoint(4, -3), MyPoint(2, 6.5)))`.

(*Hint*: For the formula to compute the area of a triangle, see Programming Exercise 2.15. Use the `java.awt.geo.Line2D` class in the Java API to implement the `contains` and `overlaps` methods. The `Line2D` class contains the methods for checking whether two line segments intersect and whether a line contains a point, and so on. 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 10.17. 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.)

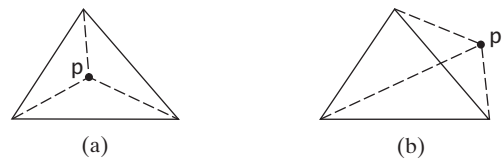
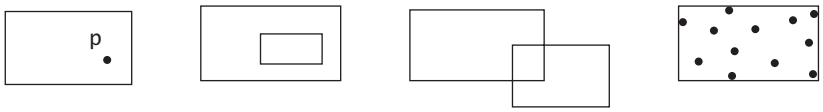


FIGURE 10.17 (a) A point is inside the triangle. (b) A point is outside the triangle.

***10.13** (*Geometry: the* `MyRectangle2D` *class*) Define the `MyRectangle2D` 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`, `width`, and `height`.
- 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 10.18a).
- A method `contains(MyRectangle2D r)` that returns `true` if the specified rectangle is inside this rectangle (see Figure 10.18b).
- A method `overlaps(MyRectangle2D r)` that returns `true` if the specified rectangle overlaps with this rectangle (see Figure 10.18c).



(a) (b) (c) (d)

FIGURE 10.18 A point is inside the rectangle. (b) A rectangle is inside another rectangle. (c) A rectangle overlaps another rectangle. (d) Points are enclosed inside a rectangle.

Draw the UML diagram for the class and then implement the class. Write a test program that creates a `MyRectangle2D` object `r1` (`new MyRectangle2D(2, 2, 5.5, 4.9)`), displays its area and perimeter, and displays the result of `r1.contains(3, 3)`, `r1.contains(new MyRectangle2D(4, 5, 10.5, 3.2))`, and `r1.overlaps(new MyRectangle2D(3, 5, 2.3, 5.4))`.

***10.14** (*The* `MyDate` *class*) Design a class named `MyDate`. The class contains:

- The data fields `year`, `month`, and `day` that represent a date. `month` is 0-based, i.e., `0` is for January.
- A no-arg constructor that creates a `MyDate` object for the current date.
- A constructor that constructs a `MyDate` object with a specified elapsed time since midnight, January 1, 1970, in milliseconds.
- A constructor that constructs a `MyDate` object with the specified year, month, and day.
- Three `get` methods for the data fields `year`, `month`, and `day`, respectively.
- A method named `setDate(long elapsedTime)` that sets a new date for the object using the elapsed time.

Draw the UML diagram for the class and then implement the class. Write a test program that creates two `MyDate` objects (using `new MyDate()` and `new MyDate(34355555133101L)`) and displays their year, month, and day.

(*Hint*: The first two constructors will extract the year, month, and day from the elapsed time. For example, if the elapsed time is `561555550000` milliseconds, the year is

1987, the month is 9, and the day is 18. You may use the GregorianCalendar class discussed in Programming Exercise 8.5 to simplify coding.)

*10.15 (*Geometry: finding the bounding rectangle*) A bounding rectangle is the minimum rectangle that encloses a set of points in a two-dimensional plane, as shown in Figure 10.18d. Write a method that returns a bounding rectangle for a set of points in a two-dimensional plane, as follows:

```
public static MyRectangle2D getRectangle(double[][] points)
```

The Rectangle2D class is defined in Exercise 10.13. Write a test program that prompts the user to enter five points and displays the bounding rectangle's center, width, and height. Here is a sample run:

```
Enter five points: 1.0 2.5 3 4 5 6 7 8 9 10 Enter
The bounding rectangle's center (5.0, 6.25), width 8.0, height 7.5
```

Sections 10.12–10.14

*10.16 (*Divisible by 2 or 3*) Find the first ten numbers with 50 decimal digits that are divisible by 2 or 3.

*10.17 (*Square numbers*) Find the first ten square numbers that are greater than Long.MAX_VALUE. A square number is a number in the form of n^2.

*10.18 (*Large prime numbers*) Write a program that finds five prime numbers larger than Long.MAX_VALUE.

*10.19 (*Mersenne prime*) A prime number is called a *Mersenne prime* if it can be written in the form $2^p - 1$ for some positive integer p. Write a program that finds all Mersenne primes with $p \leq 100$ and displays the output as shown below. (*Hint*: You have to use BigInteger to store the number, because it is too big to be stored in long. Your program may take several hours to run.)

```
p         2^p - 1

2           3
3           7
5          31
...
```

*10.20 (*Approximate e*) Programming Exercise 4.26 approximates e using the following series:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots + \frac{1}{i!}$$

In order to get better precision, use BigDecimal with 25 digits of precision in the computation. Write a program that displays the e value for i = 100, 200, . . . , and 1000.

10.21 (*Divisible by 5 or 6*) Find the first ten numbers (greater than Long.MAX_VALUE) that are divisible by 5 or 6.