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 chapter introduced objects and classes. You learned how to define classes, create objects, and use objects from several classes in the Java API (e.g., `Circle`, `Date`, `Random`, and `Point2D`). 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.

Our focus here is on class design. 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 and introducing new classes in the Java API.

10.2 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.

class abstraction

class's contract

class encapsulation

abstract data type

In Chapter 6, 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 10.1, 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).

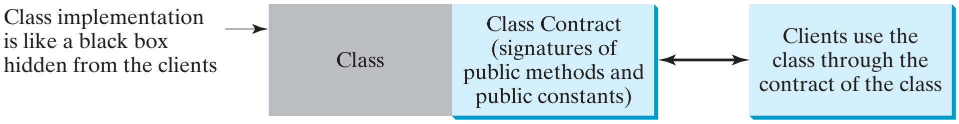

Class implementation is like a black box hidden from the clients
Class
Class Contract (signatures of public methods and public constants)
Clients use the class through the contract of the class

Figure 10.1 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 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.

VideoNote
The Loan class

Listing 2.9, 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 the loan's other 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 10.2 shows the UML class diagram for the `Loan` class.

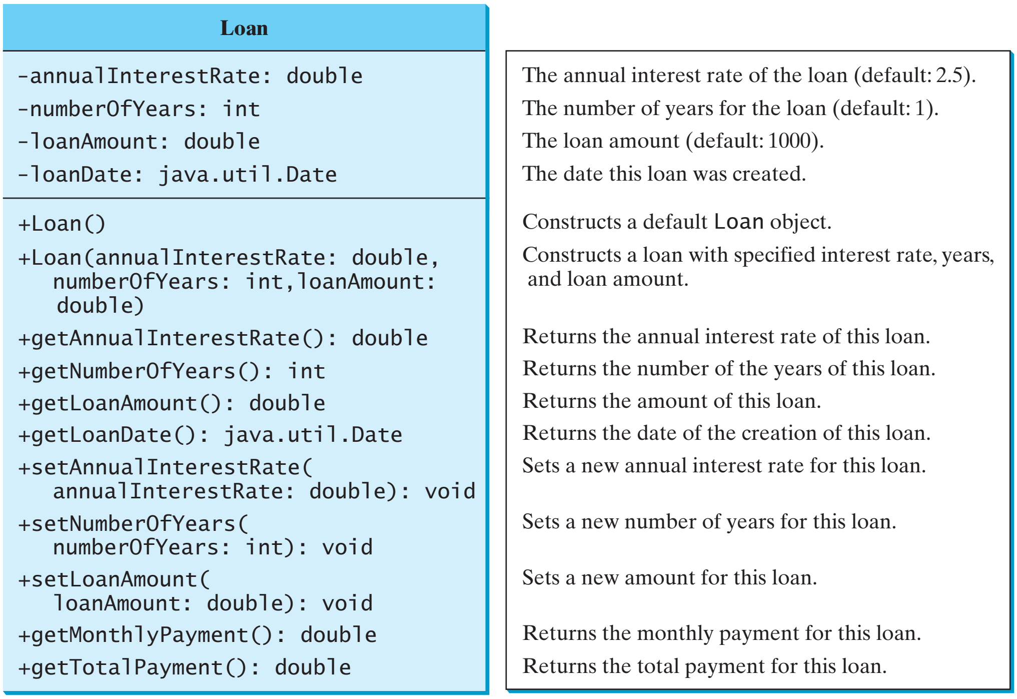

Loan
-annualInterestRate: double
-numberOfYears: int
-loanAmount: double
-loanDate: java.util.Date
+Loan()
+Loan(annualInterestRate: double, numberOfYears: int,loanAmount: double)
+getAnnualInterestRate(): double
+getNumberOfYears(): int
+getLoanAmount(): double
+getLoanDate(): java.util.Date
+setAnnualInterestRate(annualInterestRate: double): void
+setNumberOfYears(numberOfYears: int): void
+setLoanAmount(loanAmount: double): void
+getMonthlyPayment(): double
+getTotalPayment(): double
The annual interest rate of the loan (default: 2.5).
The number of years for the loan (default: 1).
The loan amount (default: 1000).
The date this loan was created.
Constructs a default Loan object.
Constructs a loan with specified interest rate, years, and loan amount.
Returns the annual interest rate of this loan.
Returns the number of the years of this loan.
Returns the amount of this loan.
Returns the date of the creation of this loan.
Sets a new annual interest rate for this loan.
Sets a new number of years for this loan.
Sets a new amount for this loan.
Returns the monthly payment for this loan.
Returns the total payment for this loan.

FIGURE 10.2 The `Loan` class models the properties and behaviors of loans.

The UML diagram in Figure 10.2 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

```java
import java.util.Scanner;

public class TestLoanClass {
  /** Main method */
  public static void main(String[] args) {
```

create Loan object

invoke instance method

invoke instance method

```
 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);
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(),
30         loan.getTotalPayment());
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

no-arg constructor

```
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() {
    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 -
      (1 / Math.pow(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;
  }
}
```

constructor

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 getter methods, three setter 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 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 10.2 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.

Check Point

10.1 If you redefine the **Loan** class in Listing 10.2 without setter methods, is the class immutable?

10.3 Thinking in Objects

Key Point

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 1–8 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 3.4, 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 10.3.

VideoNote
The BMI class

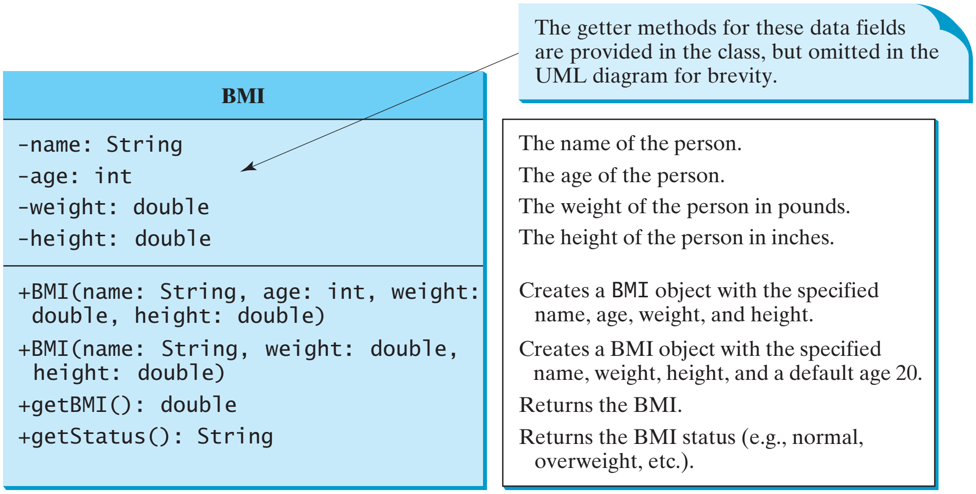

The getter methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
BMI
-name: String
-age: int
-weight: double
-height: double
+BMI(name: String, age: int, weight: double, height: double)
+BMI(name: String, weight: double, height: double)
+getBMI(): double
+getStatus(): String
The name of the person.
The age of the person.
The weight of the person in pounds.
The height of the person in inches.
Creates a BMI object with the specified name, age, weight, and height.
Creates a BMI object with the specified name, weight, height, and a default age 20.
Returns the BMI.
Returns the BMI status (e.g., normal, overweight, etc.).

FIGURE 10.3 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

```
public class BMI {
  private String name;
  private int age;
  private double weight; // in pounds
  private double height; // in inches
  public static final double KILOGRAMS_PER_POUND = 0.45359237;
  public static final double METERS_PER_INCH = 0.0254;

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

constructor

constructor

getBMI

getStatus

```
    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 < 18.5)
      return "Underweight";
    else if (bmi < 25)
      return "Normal";
    else if (bmi < 30)
      return "Overweight";
    else
      return "Obese";
  }

  public String getName() {
    return name;
  }

  public int getAge() {
    return age;
  }

  public double getWeight() {
    return weight;
  }

  public double getHeight() {
    return height;
  }
}
```

The mathematical formula for computing the BMI using weight and height is given in Section 3.8. 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 3.8.

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 passing 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.

Check Point

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

10.4 Class Relationships

Key Point

To design classes, you need to explore the relationships among classes. The common relationships among classes are association, aggregation, composition, *and* inheritance.

This section explores association, aggregation, and composition. The inheritance relationship will be introduced in the next chapter.

10.4.1 Association

association

Association is a general binary relationship that describes an activity between two classes. For example, a student taking a course is an association between the `Student` class and the `Course` class, and a faculty member teaching a course is an association between the `Faculty` class and the `Course` class. These associations can be represented in UML graphical notation, as shown in Figure 10.4.

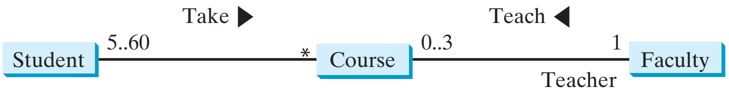


FIGURE 10.4 This UML diagram shows that a student may take any number of courses, a faculty member may teach at most three courses, a course may have from five to sixty students, and a course is taught by only one faculty member.

An association is illustrated by a solid line between two classes with an optional label that describes the relationship. In Figure 10.4, the labels are *Take* and *Teach*. Each relationship may have an optional small black triangle that indicates the direction of the relationship. In this figure, the direction indicates that a student takes a course (as opposed to a course taking a student).

Each class involved in the relationship may have a role name that describes the role it plays in the relationship. In Figure 10.4, *teacher* is the role name for `Faculty`.

multiplicity

Each class involved in an association may specify a *multiplicity*, which is placed at the side of the class to specify how many of the class's objects are involved in the relationship in UML. 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` indicates that the number of objects is between `m` and `n`, inclusively. In Figure 10.4, each student may take any number of courses, and each course must have at least five and at most sixty students. Each course is taught by only one faculty member, and a faculty member may teach from zero to three courses per semester.

In Java code, you can implement associations by using data fields and methods. For example, the relationships in Figure 10.4 may be implemented using the classes in Figure 10.5. The relation "a student takes a course" is implemented using the **addCourse** method in the **Student** class and the **addStuent** method in the **Course** class. The relation "a faculty teaches a course" is implemented using the **addCourse** method in the **Faculty** class and the **setFaculty** method in the **Course** class. The **Student** class may use a list to store the courses that the student is taking, the **Faculty** class may use a list to store the courses that the faculty is teaching, and the **Course** class may use a list to store students enrolled in the course and a data field to store the instructor who teaches the course.

```
public class Student {
  private Course[]
    courseList;

  public void addCourse(
    Course s) { ... }
}
```

```
public class Course {
  private Student[]
    classList;
  private Faculty faculty;

  public void addStudent(
    Student s) { ... }

  public void setFaculty(
    Faculty faculty) { ... }
}
```

```
public class Faculty {
  private Course[]
    courseList;

  public void addCourse(
    Course c) { ... }
}
```

FIGURE 10.5 The association relations are implemented using data fields and methods in classes.

many possible implementations

Note

There are many possible ways to implement relationships. For example, the student and faculty information in the **Course** class can be omitted, since they are already in the **Student** and **Faculty** class. Likewise, if you don't need to know the courses a student takes or a faculty member teaches, the data field **courseList** and the **addCourse** method in **Student** or **Faculty** can be omitted.

10.4.2 Aggregation and Composition

aggregation
aggregating object
aggregated object
aggregated class
aggregating class

Aggregation is a special form of association that represents an ownership relationship between two objects. Aggregation models *has-a* relationships. The owner object is called an *aggregating object*, and its class is called an *aggregating class*. The subject object is called an *aggregated object*, and its class is called an *aggregated class*.

composition

An object can be owned by several other aggregating objects. If an object is exclusively owned by an aggregating object, the relationship between the object and its aggregating object is referred to as a *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, since an address can be shared by several students. In UML, a filled diamond is attached to an aggregating class (in this case, **Student**) to denote the composition relationship with an aggregated class (**Name**), and an empty diamond is attached to an aggregating class (**Student**) to denote the aggregation relationship with an aggregated class (**Address**), as shown in Figure 10.6.

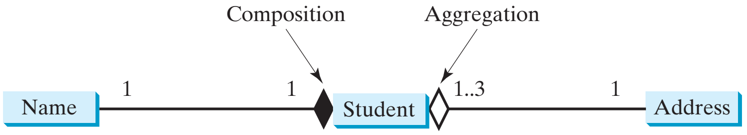
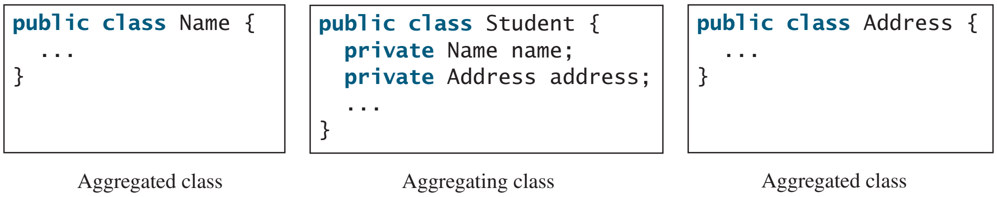


FIGURE 10.6 Each student has a name and an address.

In Figure 10.6, each student has only one multiplicity—address—and each address can 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 relationships in Figure 10.6 may be implemented using the classes in Figure 10.7. The relation "a student has a name" and "a student has an address" are implemented in the data field **name** and **address** in the **Student** class.



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

Aggregated class

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

Aggregating class

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

Aggregated class

FIGURE 10.7 The composition relations are implemented using data fields in classes.

Aggregation may exist between objects of the same class. For example, a person may have a supervisor. This is illustrated in Figure 10.8.

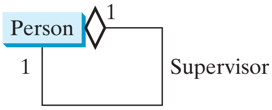


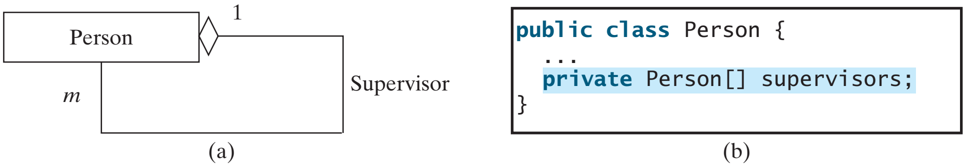
FIGURE 10.8 A person may have a supervisor.

In the relationship "a person has a supervisor," 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 10.9a, you may use an array to store supervisors, as shown in Figure 10.9b.



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

(b)

FIGURE 10.9 A person can have several supervisors.

Note

Since aggregation and composition relationships are represented using classes in the same way, we will not differentiate them and call both compositions for simplicity.

aggregation or composition

Check Point

10.3 What are common relationships among classes?

10.4 What is association? What is aggregation? What is composition?

10.5 What is UML notation of aggregation and composition?

10.6 Why both aggregation and composition are together referred to as composition?

10.5 Case Study: Designing the **Course** Class

Key Point

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 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 10.10.

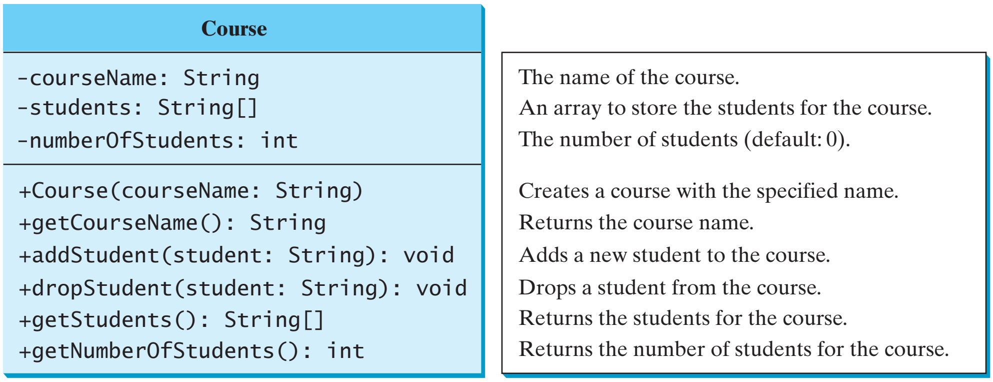

Course
-courseName: String
-students: String[]
-numberOfStudents: int
+Course(courseName: String)
+getCourseName(): String
+addStudent(student: String): void
+dropStudent(student: String): void
+getStudents(): String[]
+getNumberOfStudents(): int
The name of the course.
An array to store the students for the course.
The number of students (default: 0).
Creates a course with the specified name.
Returns the course name.
Adds a new student to the course.
Drops a student from the course.
Returns the students for the course.
Returns the number of students for the course.

Figure 10.10 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 **Course** class is available; Listing 10.5 gives a test class that creates two courses and adds students to them.

Listing 10.5 TestCourse.java

create a course

add a student

number of students

return students

```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("Kim 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.print("Number of students in course2: "
      + course2.getNumberOfStudents());
  }
}
```

```
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];
 4    private int numberOfStudents;
 5
 6    public Course(String courseName) {
 7      this.courseName = courseName;
 8    }
 9
10    public void addStudent(String student) {
11      students[numberOfStudents] = student;
12      numberOfStudents++;
13    }
14
15    public String[] getStudents() {
16      return students;
17    }
18
19    public int getNumberOfStudents() {
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  }
```

create students

add a course

return students

number of students

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 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, but you could 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.

10.6 Case Study: Designing a Class for Stacks

This section designs a class for modeling stacks.

stack

Recall that a *stack* is a data structure that holds data in a last-in, first-out fashion, as shown in Figure 10.11.

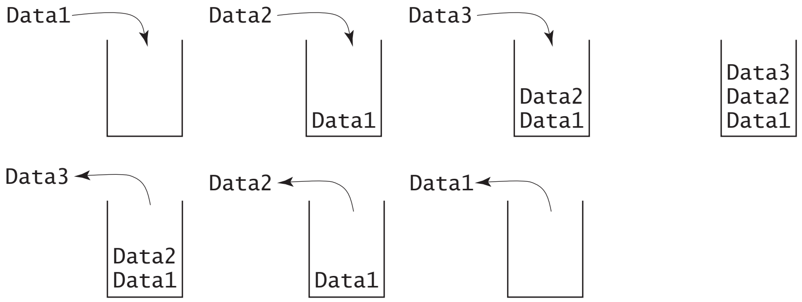
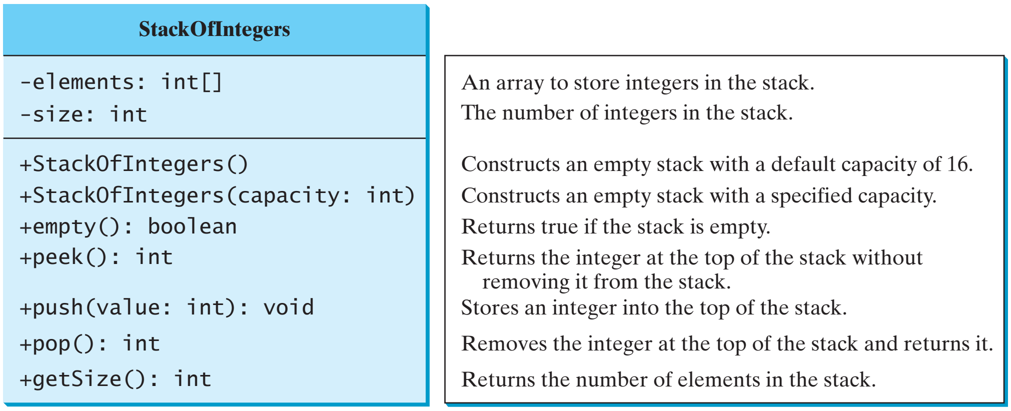


FIGURE 10.11 A stack holds data in a last-in, first-out fashion.

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.

VideoNote
The `StackOfIntegers` class

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 10.12.



StackOfIntegers

```
-elements: int[]
-size: int

+StackOfIntegers()
+StackOfIntegers(capacity: int)
+empty(): boolean
+peek(): int

+push(value: int): void
+pop(): int
+getSize(): int
```

An array to store integers in the stack.
The number of integers in the stack.

Constructs an empty stack with a default capacity of 16.
Constructs an empty stack with a specified capacity.
Returns true if the stack is empty.
Returns the integer at the top of the stack without removing it from the stack.
Stores an integer into the top of the stack.
Removes the integer at the top of the stack and returns it.
Returns the number of elements in the stack.

FIGURE 10.12 The `StackOfIntegers` class encapsulates the stack storage and provides the operations for manipulating the stack.

Suppose that the class is available. The test program in Listing 10.7 uses the class to create a stack (line 3), store ten integers `0`, `1`, `2`, . . . , and `9` (line 6), and displays them in reverse order (line 9).

LISTING 10.7 TestStackOfIntegers.java

create a stack

```
1  public class TestStackOfIntegers {
2    public static void main(String[] args) {
3      StackOfIntegers stack = new StackOfIntegers();
```

```

    for (int i = 0; i < 10; i++)
      stack.push(i);

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

push to stack

pop from stack

```
9 8 7 6 5 4 3 2 1 0
```

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 16. 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 10.13. For an empty stack, size is 0.

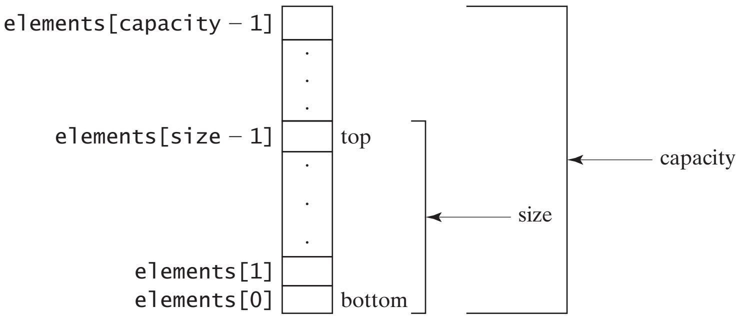


FIGURE 10.13 The StackOfIntegers class encapsulates the stack storage and provides the operations for manipulating the stack.

The StackOfIntegers class is implemented in Listing 10.8. 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 24). If the stack is full (i.e., size >= capacity), create a new array of twice the current capacity (line 19), copy the contents of the current array to the new array (line 20), and assign the reference of the new array to the current array in the stack (line 21). Now you can add the new value to the array (line 24).

LISTING 10.8 StackOfIntegers.java

```
 1 public class StackOfIntegers {
 2   private int[] elements;
 3   private int size;
 4   public static final int DEFAULT_CAPACITY = 16;
 5
 6   /** Construct a stack with the default capacity 16 */
 7   public StackOfIntegers() {
 8     this (DEFAULT_CAPACITY);
 9   }
10
11   /** Construct a stack with the specified maximum capacity */
12   public StackOfIntegers(int capacity) {
13     elements = new int[capacity];
14   }
15
```

max capacity 16

double the capacity

add to stack

```
  /** Push a new integer to the top of the stack */
  public void push(int value) {
    if (size >= elements.length) {
      int[] temp = new int[elements.length * 2];
      System.arraycopy(elements, 0, temp, 0, elements.length);
      elements = temp;
    }

    elements[size++] = value;
  }

  /** Return and remove the top element from the stack */
  public int pop() {
    return elements[--size];
  }

  /** Return the top element from the stack */
  public int peek() {
    return elements[size - 1];
  }

  /** Test whether the stack is empty */
  public boolean empty() {
    return size == 0;
  }

  /** Return the number of elements in the stack */
  public int getSize() {
    return size;
  }
}
```

10.7 Processing Primitive Data Type Values as Objects

Key Point

A primitive type value is not an object, but it can be wrapped in an object using a wrapper class in the Java API.

why wrapper class?

Owing to performance considerations, primitive data type values are not objects in Java. Because of the overhead of processing objects, the language's performance would be adversely affected if primitive data type values were treated as objects. However, 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, wrapping **double** into the **Double** class, and wrapping **char** into the **Character** class,). By using a wrapper class, you can process primitive data type values as objects. Java provides **Boolean**, **Character**, **Double**, **Float**, **Byte**, **Short**, **Integer**, and **Long** wrapper classes in the **java.lang** package for primitive data types. The **Boolean** class wraps a Boolean value **true** or **false**. This section uses **Integer** and **Double** as examples to introduce the numeric wrapper classes.

naming convention

Note

Most wrapper class names for a primitive type are the same as the primitive data type name with the first letter capitalized. The exceptions are **Integer** and **Character**.

Numeric wrapper classes are very similar to each other. Each contains the methods **doubleValue()**, **floatValue()**, **intValue()**, **longValue()**, **shortValue()**, and **byteValue()**. These methods "convert" objects into primitive type values. The key features of **Integer** and **Double** are shown in Figure 10.14.

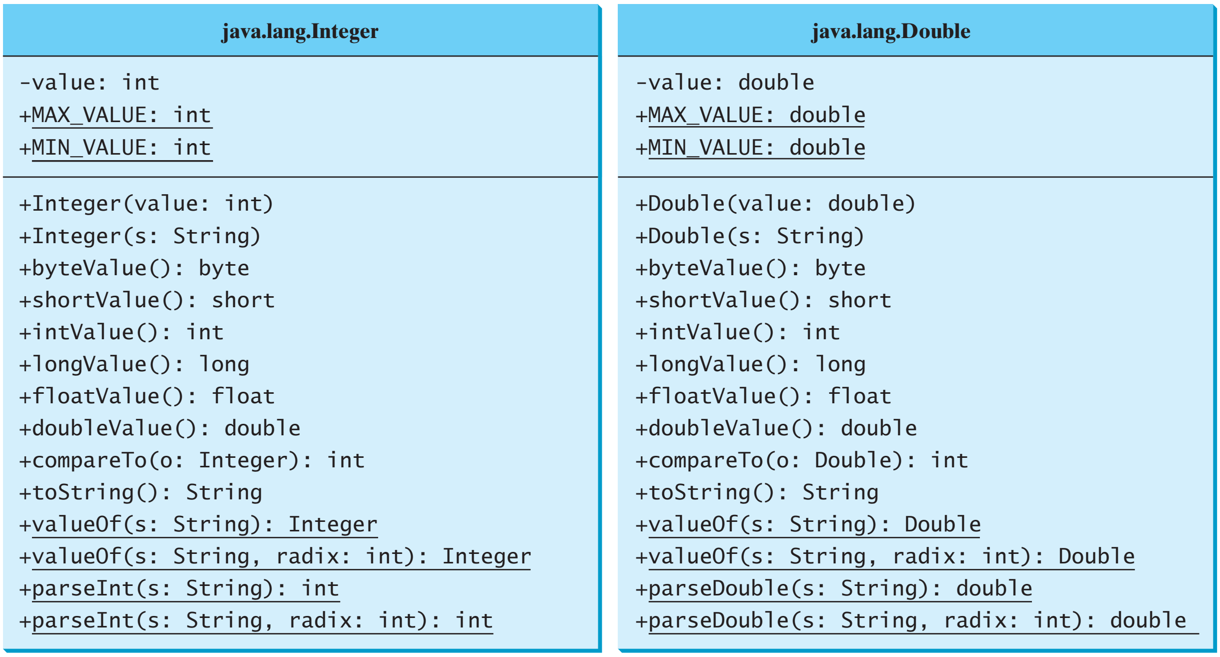


FIGURE 10.14 The wrapper classes provide constructors, constants, and conversion methods for manipulating various data types.

constructors

You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value—for example, `new Double(5.0)`, `new Double("5.0")`, `new Integer(5)`, and `new Integer("5")`.

no no-arg constructor
immutable

The wrapper classes do not have no-arg constructors. The instances of all wrapper classes are immutable; this means that, once the objects are created, their internal values cannot be changed.

constants

Each numeric wrapper class has the constants `MAX_VALUE` and `MIN_VALUE`. `MAX_VALUE` represents the maximum value of the corresponding primitive data type. For `Byte`, `Short`, `Integer`, and `Long`, `MIN_VALUE` represents the minimum `byte`, `short`, `int`, and `long` values. For `Float` and `Double`, `MIN_VALUE` represents the minimum *positive* `float` and `double` values. The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E–45), and the maximum double floating-point number (1.79769313486231570e + 308d).

```
System.out.println("The maximum integer is " + Integer.MAX_VALUE);
System.out.println("The minimum positive float is " +
  Float.MIN_VALUE);
System.out.println(
  "The maximum double-precision floating-point number is " +
  Double.MAX_VALUE);
```

conversion methods

Each numeric wrapper class contains the methods `doubleValue()`, `floatValue()`, `intValue()`, `longValue()`, and `shortValue()` for returning a `double`, `float`, `int`, `long`, or `short` value for the wrapper object. For example,

`new Double(12.4).intValue()` returns `12`;
`new Integer(12).doubleValue()` returns `12.0`;

`compareTo` method

Recall that the `String` class contains the `compareTo` method for comparing two strings. The numeric wrapper classes contain the `compareTo` method for comparing two numbers and returns 1, 0, or -1, if this number is greater than, equal to, or less than the other number. For example,

```
new Double(12.4).compareTo(new Double(12.3)) returns 1;
new Double(12.3).compareTo(new Double(12.3)) returns 0;
new Double(12.3).compareTo(new Double(12.51)) returns -1;
```

static valueOf methods

The numeric wrapper classes have a useful static method, valueOf (String s). This method creates a new object initialized to the value represented by the specified string. For example,

```
Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");
```

static parsing methods

You have used the parseInt method in the Integer class to parse a numeric string into an int value and the parseDouble method in the Double class to parse a numeric string into a double value. Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value based on 10 (decimal) or any specified radix (e.g., 2 for binary, 8 for octal, and 16 for hexadecimal).

```
// These two methods are in the Byte class
public static byte parseByte(String s)
public static byte parseByte(String s, int radix)

// These two methods are in the Short class
public static short parseShort(String s)
public static short parseShort(String s, int radix)

// These two methods are in the Integer class
public static int parseInt(String s)
public static int parseInt(String s, int radix)

// These two methods are in the Long class
public static long parseLong(String s)
public static long parseLong(String s, int radix)

// These two methods are in the Float class
public static float parseFloat(String s)
public static float parseFloat(String s, int radix)

// These two methods are in the Double class
public static double parseDouble(String s)
public static double parseDouble(String s, int radix)
```

For example,

```
Integer.parseInt("11", 2) returns 3;
Integer.parseInt("12", 8) returns 10;
Integer.parseInt("13", 10) returns 13;
Integer.parseInt("1A", 16) returns 26;
```

Integer.parseInt("12", 2) would raise a runtime exception because 12 is not a binary number.

converting decimal to hex

Note that you can convert a decimal number into a hex number using the format method. For example,

```
String.format("%x", 26) returns 1A;
```

Check Point

10.7 Describe primitive-type wrapper classes.

10.8 Can each of the following statements be compiled?

a. `Integer i = new Integer("23");`

b. `Integer i = new Integer(23);`

c. `Integer i = Integer.valueOf("23");`

d. `Integer i = Integer.parseInt("23", 8);`

e. `Double d = new Double();`

f. `Double d = Double.valueOf("23.45");`

g. `int i = (Integer.valueOf("23")).intValue();`

h. `double d = (Double.valueOf("23.4")).doubleValue();`

i. `int i = (Double.valueOf("23.4")).intValue();`

j. `String s = (Double.valueOf("23.4")).toString();`

10.9 How do you convert an integer into a string? How do you convert a numeric string into an integer? How do you convert a double number into a string? How do you convert a numeric string into a double value?

10.10 Show the output of the following code.

```
public class Test {
  public static void main(String[] args) {
    Integer x = new Integer(3);
    System.out.println(x.intValue());
    System.out.println(x.compareTo(new Integer(4)));
  }
}
```

10.11 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    System.out.println(Integer.parseInt("10"));
    System.out.println(Integer.parseInt("10", 10));
    System.out.println(Integer.parseInt("10", 16));
    System.out.println(Integer.parseInt("11"));
    System.out.println(Integer.parseInt("11", 10));
    System.out.println(Integer.parseInt("11", 16));
  }
}
```

10.8 Automatic Conversion between Primitive Types and Wrapper Class Types

Key Point

A primitive type value can be automatically converted to an object using a wrapper class, and vice versa, depending on the context.

Converting a primitive value to a wrapper object is called *boxing*. The reverse conversion is called *unboxing*. Java allows primitive types and wrapper classes to be converted automatically. The compiler will automatically box a primitive value that appears in a context requiring an object, and will unbox an object that appears in a context requiring a primitive value. This is called *autoboxing* and *autounboxing*.

boxing
unboxing

autoboxing
autounboxing

For instance, the following statement in (a) can be simplified as in (b) due to autoboxing.

```
Integer intObject = new Integer (2);
```

(a)

Equivalent

autoboxing

```
Integer intObject = 2;
```

(b)

Consider the following example:

```
1  Integer[] intArray = {1, 2, 3};
2  System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

In line 1, the primitive values **1**, **2**, and **3** are automatically boxed into objects **new Integer(1)**, **new Integer(2)**, and **new Integer(3)**. In line 2, the objects **intArray[0]**, **intArray[1]**, and **intArray[2]** are automatically unboxed into **int** values that are added together.

Check Point

10.12 What are autoboxing and autounboxing? Are the following statements correct?

a. `Integer x = 3 + new Integer(5);`

b. `Integer x = 3;`

c. `Double x = 3;`

d. `Double x = 3.0;`

e. `int x = new Integer(3);`

f. `int x = new Integer(3) + new Integer(4);`

10.13 Show the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    Double x = 3.5;
    System.out.println(x.intValue());
    System.out.println(x.compareTo(4.5));
  }
}
```

10.9 The **BigInteger** and **BigDecimal** Classes

Key Point

The **BigInteger** *and* **BigDecimal** *classes can be used to represent integers or decimal numbers of any size and precision.*

immutable

VideoNote

Process large numbers

If you need to compute with very large integers or high-precision floating-point values, you can use the **BigInteger** and **BigDecimal** classes in the **java.math** package. Both are *immutable*. The largest integer of the **long** type is **Long.MAX_VALUE** (i.e., **9223372036854775807**). An instance of **BigInteger** can represent an integer of any size. You can use **new BigInteger(String)** and **new BigDecimal(String)** to create an instance of **BigInteger** and **BigDecimal**, use the **add**, **subtract**, **multiply**, **divide**, and **remainder** methods to perform arithmetic operations, and use the **compareTo** method to compare two big numbers. For example, the following code creates two **BigInteger** objects and multiplies them.

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

The output is 18446744073709551614.

There is no limit to the precision of a BigDecimal object. The divide method may throw an ArithmeticException if the result cannot be terminated. However, you can use the overloaded divide(BigDecimal d, int scale, int roundingMode) method to specify a scale and a rounding mode to avoid this exception, where scale is the maximum number of digits after the decimal point. For example, the following code creates two BigDecimal objects and performs division with scale 20 and rounding mode BigDecimal.ROUND_UP.

```
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
```

The output is 0.33333333333333333334.

Note that the factorial of an integer can be very large. Listing 10.9 gives a method that can return the factorial of any integer.

LISTING 10.9 LargeFactorial.java

```
 1  import java.math.*;
 2
 3  public class LargeFactorial {
 4    public static void main(String[] args) {
 5      System.out.println("50! is \n" + factorial(50));
 6    }
 7
 8    public static BigInteger factorial(long n) {
 9      BigInteger result = BigInteger.ONE;
10      for (int i = 1; i <= n; i++)
11        result = result.multiply(new BigInteger(i + ""));
12
13      return result;
14    }
15  }
```

constant

multiply

```
50! is
30414093201713378043612608166064768844377641568960512000000000000
```

BigInteger.ONE (line 9) is a constant defined in the BigInteger class. BigInteger.ONE is the same as new BigInteger("1").

A new result is obtained by invoking the multiply method (line 11).

Check Point

10.14 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    java.math.BigInteger x = new java.math.BigInteger("3");
    java.math.BigInteger y = new java.math.BigInteger("7");
    java.math.BigInteger z = x.add(y);
    System.out.println("x is " + x);
    System.out.println("y is " + y);
    System.out.println("z is " + z);
  }
}
```

10.10 The String Class

Key Point

A String object is immutable: Its content cannot be changed once the string is created.

VideoNote

The String class

Strings were introduced in Section 4.4. You know strings are objects. You can invoke the charAt(index) method to obtain a character at the specified index from a string, the length() method to return the size of a string, the substring method to return a substring in a string, and the indexOf and lastIndexOf methods to return the first or last index of a matching character or a substring. We will take a closer look at strings in this section.

The String class has 13 constructors and more than 40 methods for manipulating strings. Not only is it very useful in programming, but it is also a good example for learning classes and objects.

10.10.1 Constructing a String

You can create a string object from a string literal or from an array of characters. To create a string from a string literal, use the syntax:

```
String newString = new String(stringLiteral);
```

The argument stringLiteral is a sequence of characters enclosed inside double quotes. The following statement creates a String object message for the string literal "Welcome to Java":

```
String message = new String("Welcome to Java");
```

string literal object

Java treats a string literal as a String object. Thus, the following statement is valid:

```
String message = "Welcome to Java";
```

You can also create a string from an array of characters. For example, the following statements create the string "Good Day":

```
char[] charArray = {'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y'};
String message = new String(charArray);
```

String variable, String object, string value

Note

A String variable holds a reference to a String object that stores a string value. Strictly speaking, the terms *String variable*, *String object*, and *string value* are different, but most of the time the distinctions between them can be ignored. For simplicity, the term *string* will often be used to refer to String variable, String object, and string value.

10.10.2 Immutable Strings and Interned Strings

immutable

A String object is immutable; its contents cannot be changed. Does the following code change the contents of the string?

```
String s = "Java";
s = "HTML";
```

The answer is no. The first statement creates a String object with the content "Java" and assigns its reference to s. The second statement creates a new String object with the content "HTML" and assigns its reference to s. The first String object still exists after the assignment, but it can no longer be accessed, because variable s now points to the new object, as shown in Figure 10.15.

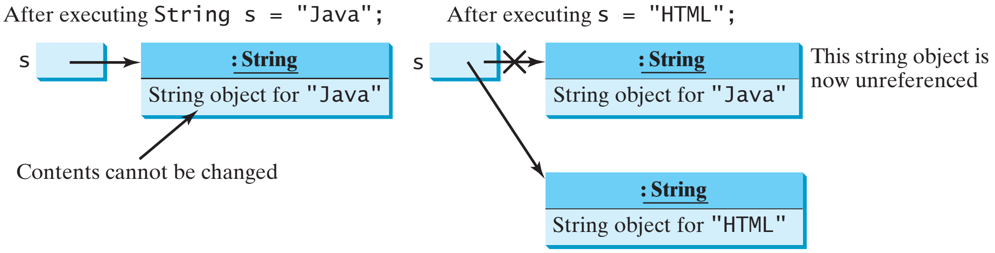

After executing String s = "Java";
After executing s = "HTML";
s
: String
String object for "Java"
Contents cannot be changed
s
: String
String object for "Java"
This string object is now unreferenced
: String
String object for "HTML"

FIGURE 10.15 Strings are immutable; once created, their contents cannot be changed.

Because strings are immutable and are ubiquitous in programming, the JVM uses a unique instance for string literals with the same character sequence in order to improve efficiency and save memory. Such an instance is called an *interned string*. For example, the following statements:

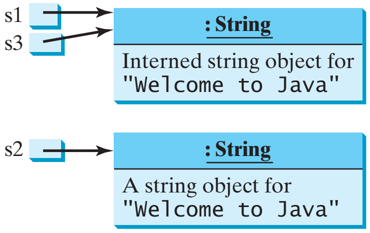
interned string

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```


s1
s3
: String
Interned string object for "Welcome to Java"
s2
: String
A string object for "Welcome to Java"

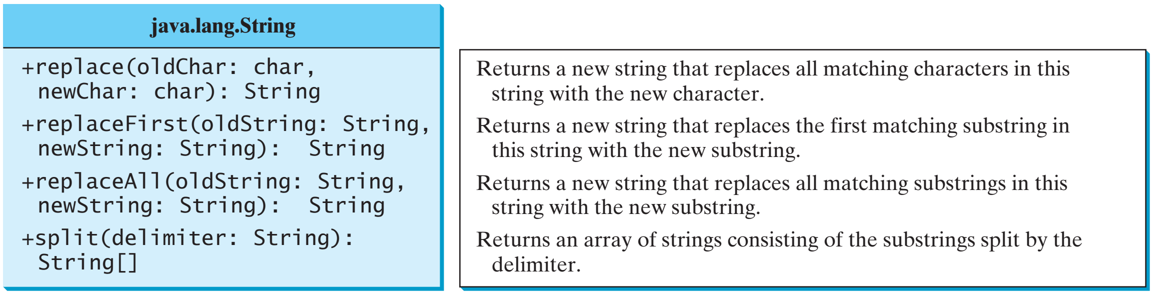
display

```
s1 == s2 is false
s1 == s3 is true
```

In the preceding statements, `s1` and `s3` refer to the same interned string—`"Welcome to Java"`—so `s1 == s3` is `true`. However, `s1 == s2` is `false`, because `s1` and `s2` are two different string objects, even though they have the same contents.

10.10.3 Replacing and Splitting Strings

The `String` class provides the methods for replacing and splitting strings, as shown in Figure 10.16.



java.lang.String

```
+replace(oldChar: char,
  newChar: char): String
+replaceFirst(oldString: String,
  newString: String):  String
+replaceAll(oldString: String,
  newString: String):  String
+split(delimiter: String):
  String[]
```

Returns a new string that replaces all matching characters in this string with the new character.

Returns a new string that replaces the first matching substring in this string with the new substring.

Returns a new string that replaces all matching substrings in this string with the new substring.

Returns an array of strings consisting of the substrings split by the delimiter.

FIGURE 10.16 The `String` class contains the methods for replacing and splitting strings.

Once a string is created, its contents cannot be changed. The methods **replace**, **replaceFirst**, and **replaceAll** return a new string derived from the original string (without changing the original string!). Several versions of the **replace** methods are provided to replace a character or a substring in the string with a new character or a new substring.

For example,

replace
replaceFirst
replace
replace

"Welcome".replace('e', 'A') returns a new string, **WAlcomA**.
"Welcome".replaceFirst("e", "AB") returns a new string, **WABlcome**.
"Welcome".replace("e", "AB") returns a new string, **WABlcomAB**.
"Welcome".replace("el", "AB") returns a new string, **WABcome**.

split

The **split** method can be used to extract tokens from a string with the specified delimiters. For example, the following code

```
String[] tokens = "Java#HTML#Perl".split("#");
for (int i = 0; i < tokens.length; i++)
  System.out.print(tokens[i] + " ");
```

displays

Java HTML Perl

10.10.4 Matching, Replacing and Splitting by Patterns

why regular expression?

Often you will need to write code that validates user input, such as to check whether the input is a number, a string with all lowercase letters, or a Social Security number. How do you write this type of code? A simple and effective way to accomplish this task is to use the regular expression.

regular expression
regex

A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings. You can match, replace, or split a string by specifying a pattern. This is an extremely useful and powerful feature.

matches(regex)

Let us begin with the **matches** method in the **String** class. At first glance, the **matches** method is very similar to the **equals** method. For example, the following two statements both evaluate to **true**.

```
"Java".matches("Java");
"Java".equals("Java");
```

However, the **matches** method is more powerful. It can match not only a fixed string, but also a set of strings that follow a pattern. For example, the following statements all evaluate to **true**:

```
"Java is fun".matches("Java.*")
"Java is cool".matches("Java.*")
"Java is powerful".matches("Java.*")
```

Java.* in the preceding statements is a regular expression. It describes a string pattern that begins with Java followed by *any* zero or more characters. Here, the substring matches any zero or more characters.

The following statement evaluates to **true**.

```
"440-02-4534".matches("\\d{3}-\\d{2}-\\d{4}")
```

Here **\\d** represents a single digit, and **\\d{3}** represents three digits.

The replaceAll, replaceFirst, and split methods can be used with a regular expression. For example, the following statement returns a new string that replaces $, +, or # in a+b$#c with the string NNN.

replaceAll(regex)

```
String s = "a+b$#c".replaceAll("[$+#]", "NNN");
System.out.println(s);
```

Here the regular expression [$+#] specifies a pattern that matches $, +, or #. So, the output is aNNNbNNNNNNc.

The following statement splits the string into an array of strings delimited by punctuation marks.

split(regex)

```
String[] tokens = "Java,C?C#,C++".split("[.,:;?]");

for (int i = 0; i < tokens.length; i++)
  System.out.println(tokens[i]);
```

further studies

In this example, the regular expression [.,:;?] specifies a pattern that matches ., ,, :, ;, or ?. Each of these characters is a delimiter for splitting the string. Thus, the string is split into Java, C, C#, and C++, which are stored in array tokens.

Regular expression patterns are complex for beginning students to understand. For this reason, simple patterns are introduced in this section. Please refer to Appendix H, Regular Expressions, to learn more about these patterns.

10.10.5 Conversion between Strings and Arrays

Strings are not arrays, but a string can be converted into an array, and vice versa. To convert a string into an array of characters, use the toCharArray method. For example, the following statement converts the string Java to an array.

toCharArray

```
char[] chars = "Java".toCharArray();
```

Thus, chars[0] is J, chars[1] is a, chars[2] is v, and chars[3] is a.

You can also use the getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) method to copy a substring of the string from index srcBegin to index srcEnd-1 into a character array dst starting from index dstBegin. For example, the following code copies a substring "3720" in "CS3720" from index 2 to index 6-1 into the character array dst starting from index 4.

getChars

```
char[] dst = {'J', 'A', 'V', 'A', '1', '3', '0', '1'};
"CS3720".getChars(2, 6, dst, 4);
```

Thus, dst becomes {'J', 'A', 'V', 'A', '3', '7', '2', '0'}.

To convert an array of characters into a string, use the String(char[]) constructor or the valueOf(char[]) method. For example, the following statement constructs a string from an array using the String constructor.

```
String str = new String(new char[]{'J', 'a', 'v', 'a'});
```

The next statement constructs a string from an array using the valueOf method.

valueOf

```
String str = String.valueOf(new char[]{'J', 'a', 'v', 'a'});
```

10.10.6 Converting Characters and Numeric Values to Strings

Recall that you can use Double.parseDouble(str) or Integer.parseInt(str) to convert a string to a double value or an int value and you can convert a character or a number into a string by using the string concatenating operator. Another way of converting a overloaded `valueOf`

number into a string is to use the overloaded static `valueOf` method. This method can also be used to convert a character or an array of characters into a string, as shown in Figure 10.17.

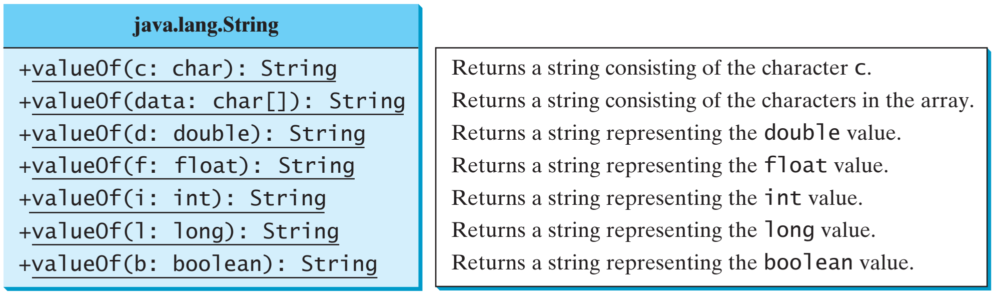

java.lang.String
+valueOf(c: char): String
+valueOf(data: char[]): String
+valueOf(d: double): String
+valueOf(f: float): String
+valueOf(i: int): String
+valueOf(l: long): String
+valueOf(b: boolean): String
Returns a string consisting of the character c.
Returns a string consisting of the characters in the array.
Returns a string representing the double value.
Returns a string representing the float value.
Returns a string representing the int value.
Returns a string representing the long value.
Returns a string representing the boolean value.

Figure 10.17 The `String` class contains the static methods for creating strings from primitive type values.

For example, to convert a `double` value `5.44` to a string, use `String.valueOf(5.44)`. The return value is a string consisting of the characters `'5'`, `'.'`, `'4'`, and `'4'`.

10.10.7 Formatting Strings

The `String` class contains the static `format` method to return a formatted string. The syntax to invoke this method is:

```
String.format(format, item1, item2, ..., itemk)
```

This method is similar to the `printf` method except that the `format` method returns a formatted string, whereas the `printf` method displays a formatted string. For example,

```
String s = String.format("%7.2f%6d%-4s", 45.556, 14, "AB");
System.out.println(s);
```

displays

```
□□45.56□□□□14AB□□
```

Note that

```
System.out.printf(format, item1, item2, ..., itemk);
```

is equivalent to

```
System.out.print(
  String.format(format, item1, item2, ..., itemk));
```

where the square box (□) denotes a blank space.

Check Point

10.15 Suppose that `s1`, `s2`, `s3`, and `s4` are four strings, given as follows:

```
String s1 = "Welcome to Java";
String s2 = s1;
String s3 = new String("Welcome to Java");
String s4 = "Welcome to Java";
```

What are the results of the following expressions?

a. `s1 == s2`

b. `s1 == s3`

c. s1 == s4

d. s1.equals(s3)

e. s1.equals(s4)

f. "Welcome to Java".replace("Java", "HTML")

g. s1.replace('o', 'T')

h. s1.replaceAll("o", "T")

i. s1.replaceFirst("o", "T")

j. s1.toCharArray()

10.16 To create the string Welcome to Java, you may use a statement like this:

```
String s = "Welcome to Java";
```

or:

```
String s = new String("Welcome to Java");
```

Which one is better? Why?

10.17 What is the output of the following code?

```
String s1 = "Welcome to Java";
String s2 = s1.replace("o", "abc");
System.out.println(s1);
System.out.println(s2);
```

10.18 Let s1 be "Welcome" and s2 be "welcome". Write the code for the following statements:

a. Replace all occurrences of the character e with E in s1 and assign the new string to s2.

b. Split Welcome to Java and HTML into an array tokens delimited by a space and assign the first two tokens into s1 and s2.

10.19 Does any method in the String class change the contents of the string?

10.20 Suppose string s is created using new String(); what is s.length()?

10.21 How do you convert a char, an array of characters, or a number to a string?

10.22 Why does the following code cause a NullPointerException?

```
public class Test {
  private String text;

  public Test(String s) {
    String text  = s;
  }

  public static void main(String[] args) {
    Test test = new Test("ABC");
    System.out.println(test.text.toLowerCase());
  }
}
```

10.23 What is wrong in the following program?

```
public class Test {
  String text;

  public void Test(String s) {
    text = s;
  }

  public static void main(String[] args) {
    Test test = new Test("ABC");
    System.out.println(test);
  }
}
```

10.24 Show the output of the following code.

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Hi, ABC, good".matches("ABC "));
    System.out.println("Hi, ABC, good".matches(".*ABC.*"));
    System.out.println("A,B;C".replaceAll(",;", "#"));
    System.out.println("A,B;C".replaceAll("[,;]", "#"));

    String[] tokens = "A,B;C".split("[,;]");
    for (int i = 0; i < tokens.length; i++)
      System.out.print(tokens[i] +  " ");
  }
}
```

10.25 Show the output of the following code.

```
public class Test {
  public static void main(String[] args) {
    String s = "Hi, Good Morning";
    System.out.println(m(s));
  }

  public static int m(String s) {
    int count = 0;
    for (int i = 0; i < s.length(); i++)
      if (Character.isUpperCase(s.charAt(i)))
        count++;

    return count;
  }
}
```

10.11 The StringBuilder and StringBuffer Classes

Key Point

The StringBuilder and StringBuffer classes are similar to the String class except that the String class is immutable.

In general, the StringBuilder and StringBuffer classes can be used wherever a string is used. StringBuilder and StringBuffer are more flexible than String. You can add, insert, or append new contents into StringBuilder and StringBuffer objects, whereas the value of a String object is fixed once the string is created.

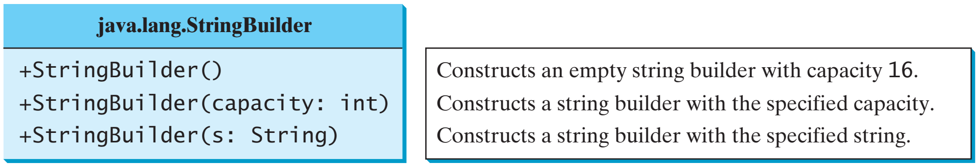
StringBuilder

The StringBuilder class is similar to StringBuffer except that the methods for modifying the buffer in StringBuffer are *synchronized*, which means that only one task is allowed to execute the methods. Use StringBuffer if the class might be accessed by multiple tasks concurrently, because synchronization is needed in this case to prevent corruptions to

StringBuffer. Concurrent programming will be introduced in Chapter 30. Using StringBuilder is more efficient if it is accessed by just a single task, because no synchronization is needed in this case. The constructors and methods in StringBuffer and StringBuilder are almost the same. This section covers StringBuilder. You can replace StringBuilder in all occurrences in this section by StringBuffer. The program can compile and run without any other changes.

StringBuilder constructors

The StringBuilder class has three constructors and more than 30 methods for managing the builder and modifying strings in the builder. You can create an empty string builder or a string builder from a string using the constructors, as shown in Figure 10.18.

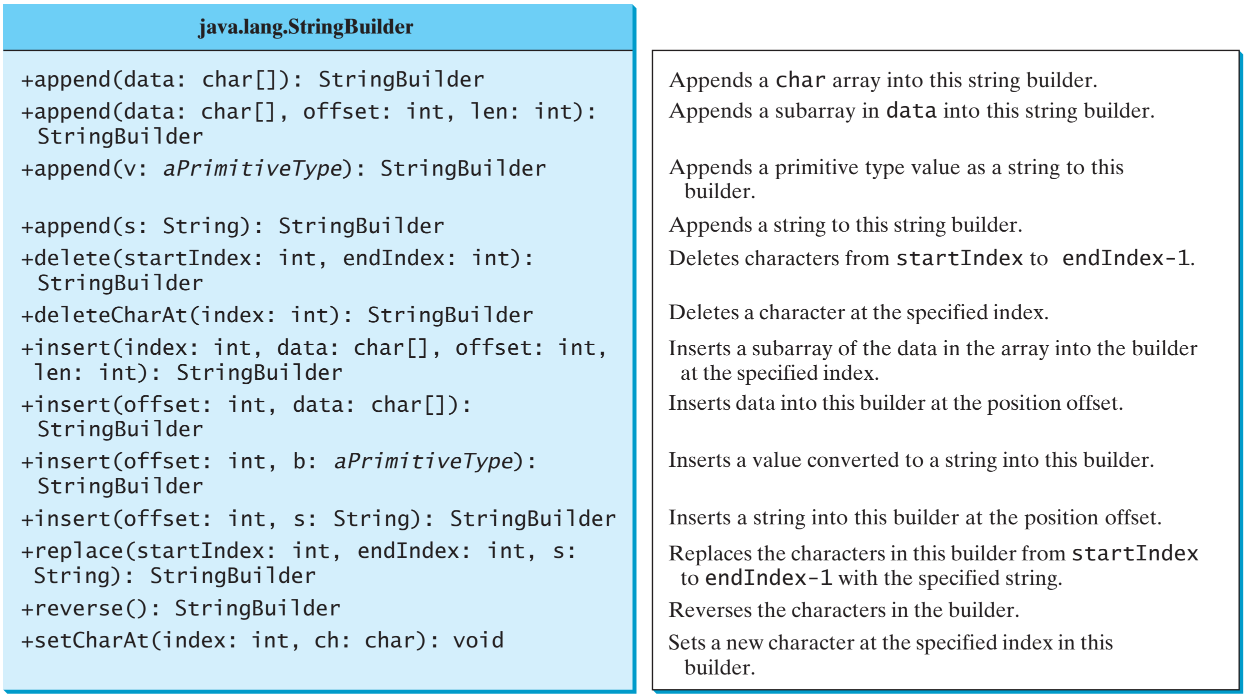


java.lang.StringBuilder	
+StringBuilder()	Constructs an empty string builder with capacity 16.
+StringBuilder(capacity: int)	Constructs a string builder with the specified capacity.
+StringBuilder(s: String)	Constructs a string builder with the specified string.

FIGURE 10.18 The StringBuilder class contains the constructors for creating instances of StringBuilder.

10.11.1 Modifying Strings in the StringBuilder

You can append new contents at the end of a string builder, insert new contents at a specified position in a string builder, and delete or replace characters in a string builder, using the methods listed in Figure 10.19.



java.lang.StringBuilder	
+append(data: char[]): StringBuilder	Appends a char array into this string builder.
+append(data: char[], offset: int, len: int): StringBuilder	Appends a subarray in data into this string builder.
+append(v: *aPrimitiveType*): StringBuilder	Appends a primitive type value as a string to this builder.
+append(s: String): StringBuilder	Appends a string to this string builder.
+delete(startIndex: int, endIndex: int): StringBuilder	Deletes characters from startIndex to endIndex-1.
+deleteCharAt(index: int): StringBuilder	Deletes a character at the specified index.
+insert(index: int, data: char[], offset: int, len: int): StringBuilder	Inserts a subarray of the data in the array into the builder at the specified index.
+insert(offset: int, data: char[]): StringBuilder	Inserts data into this builder at the position offset.
+insert(offset: int, b: *aPrimitiveType*): StringBuilder	Inserts a value converted to a string into this builder.
+insert(offset: int, s: String): StringBuilder	Inserts a string into this builder at the position offset.
+replace(startIndex: int, endIndex: int, s: String): StringBuilder	Replaces the characters in this builder from startIndex to endIndex-1 with the specified string.
+reverse(): StringBuilder	Reverses the characters in the builder.
+setCharAt(index: int, ch: char): void	Sets a new character at the specified index in this builder.

FIGURE 10.19 The StringBuilder class contains the methods for modifying string builders.

The `StringBuilder` class provides several overloaded methods to append `boolean`, `char`, `char[]`, `double`, `float`, `int`, `long`, and `String` into a string builder. For example, the following code appends strings and characters into `stringBuilder` to form a new string, `Welcome to Java`.

append

```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append("Welcome");
stringBuilder.append(' ');
stringBuilder.append("to");
stringBuilder.append(' ');
stringBuilder.append("Java");
```

The `StringBuilder` class also contains overloaded methods to insert `boolean`, `char`, `char array`, `double`, `float`, `int`, `long`, and `String` into a string builder. Consider the following code:

insert

```
stringBuilder.insert(11, "HTML and ");
```

Suppose `stringBuilder` contains `Welcome to Java` before the `insert` method is applied. This code inserts `"HTML and "` at position 11 in `stringBuilder` (just before the `J`). The new `stringBuilder` is `Welcome to HTML and Java`.

You can also delete characters from a string in the builder using the two `delete` methods, reverse the string using the `reverse` method, replace characters using the `replace` method, or set a new character in a string using the `setCharAt` method.

For example, suppose `stringBuilder` contains `Welcome to Java` before each of the following methods is applied:

delete
deleteCharAt
reverse
replace
setCharAt

`stringBuilder.delete(8, 11)` changes the builder to `Welcome Java`.
`stringBuilder.deleteCharAt(8)` changes the builder to `Welcome o Java`.
`stringBuilder.reverse()` changes the builder to `avaJ ot emocleW`.
`stringBuilder.replace(11, 15, "HTML")` changes the builder to `Welcome to HTML`.
`stringBuilder.setCharAt(0, 'w')` sets the builder to `welcome to Java`.

All these modification methods except `setCharAt` do two things:

- Change the contents of the string builder

ignore return value

- Return the reference of the string builder

For example, the following statement

```
StringBuilder stringBuilder1 = stringBuilder.reverse();
```

reverses the string in the builder and assigns the builder's reference to `stringBuilder1`. Thus, `stringBuilder` and `stringBuilder1` both point to the same `StringBuilder` object. Recall that a value-returning method can be invoked as a statement, if you are not interested in the return value of the method. In this case, the return value is simply ignored. For example, in the following statement

```
stringBuilder.reverse();
```

the return value is ignored.

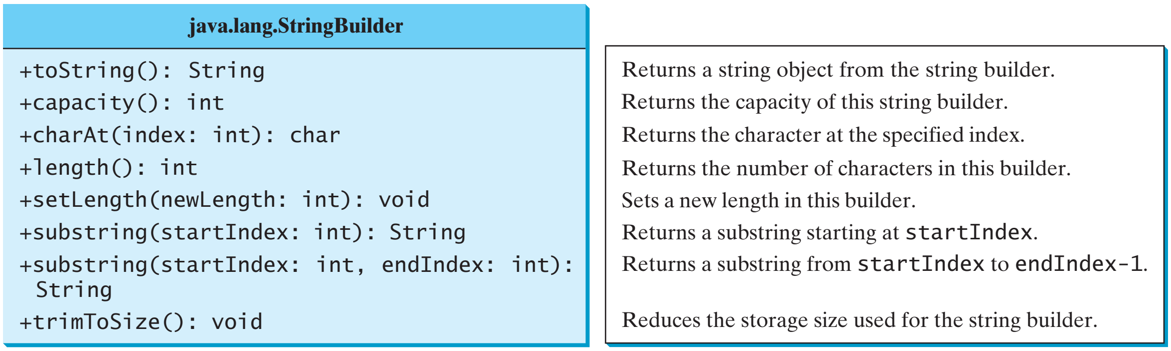
`String` or `StringBuilder`?

Tip

If a string does not require any change, use `String` rather than `StringBuilder`. Java can perform some optimizations for `String`, such as sharing interned strings.

10.11.2 The **toString**, **capacity**, **length**, **setLength**, and **charAt** Methods

The **StringBuilder** class provides the additional methods for manipulating a string builder and obtaining its properties, as shown in Figure 10.20.



java.lang.StringBuilder	
+toString(): String	Returns a string object from the string builder.
+capacity(): int	Returns the capacity of this string builder.
+charAt(index: int): char	Returns the character at the specified index.
+length(): int	Returns the number of characters in this builder.
+setLength(newLength: int): void	Sets a new length in this builder.
+substring(startIndex: int): String	Returns a substring starting at startIndex.
+substring(startIndex: int, endIndex: int): String	Returns a substring from startIndex to endIndex-1.
+trimToSize(): void	Reduces the storage size used for the string builder.

FIGURE 10.20 The **StringBuilder** class contains the methods for modifying string builders.

capacity()

The **capacity()** method returns the current capacity of the string builder. The capacity is the number of characters the string builder is able to store without having to increase its size.

length()

setLength(int)

The **length()** method returns the number of characters actually stored in the string builder. The **setLength(newLength)** method sets the length of the string builder. If the **newLength** argument is less than the current length of the string builder, the string builder is truncated to contain exactly the number of characters given by the **newLength** argument. If the **newLength** argument is greater than or equal to the current length, sufficient null characters (**\u0000**) are appended to the string builder so that **length** becomes the **newLength** argument. The **newLength** argument must be greater than or equal to **0**.

charAt(int)

The **charAt(index)** method returns the character at a specific **index** in the string builder. The index is **0** based. The first character of a string builder is at index **0**, the next at index **1**, and so on. The **index** argument must be greater than or equal to **0**, and less than the length of the string builder.

Note

length and capacity

The length of the string is always less than or equal to the capacity of the builder. The length is the actual size of the string stored in the builder, and the capacity is the current size of the builder. The builder's capacity is automatically increased if more characters are added to exceed its capacity. Internally, a string builder is an array of characters, so the builder's capacity is the size of the array. If the builder's capacity is exceeded, the array is replaced by a new array. The new array size is **2 * (the previous array size + 1)**.

Tip

initial capacity

trimToSize()

You can use **new StringBuilder(initialCapacity)** to create a **StringBuilder** with a specified initial capacity. By carefully choosing the initial capacity, you can make your program more efficient. If the capacity is always larger than the actual length of the builder, the JVM will never need to reallocate memory for the builder. On the other hand, if the capacity is too large, you will waste memory space. You can use the **trimToSize()** method to reduce the capacity to the actual size.

10.11.3 Case Study: Ignoring Nonalphanumeric Characters When Checking Palindromes

Listing 5.14, Palindrome.java, considered all the characters in a string to check whether it is a palindrome. Write a new program that ignores nonalphanumeric characters in checking whether a string is a palindrome.

Here are the steps to solve the problem:

1. Filter the string by removing the nonalphanumeric characters. This can be done by creating an empty string builder, adding each alphanumeric character in the string to a string builder, and returning the string from the string builder. You can use the **isLetterOrDigit(ch)** method in the **Character** class to check whether character **ch** is a letter or a digit.
2. Obtain a new string that is the reversal of the filtered string. Compare the reversed string with the filtered string using the **equals** method.

The complete program is shown in Listing 10.10.

LISTING 10.10 PalindromeIgnoreNonAlphanumeric.java

```java
import java.util.Scanner;

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

    // Prompt the user to enter a string
    System.out.print("Enter a string: ");
    String s = input.nextLine();

    // Display result
    System.out.println("Ignoring nonalphanumeric characters, \nis "
      + s + " a palindrome? " + isPalindrome(s));
  }

  /** Return true if a string is a palindrome */
  public static boolean isPalindrome(String s) {
    // Create a new string by eliminating nonalphanumeric chars
    String s1 = filter(s);

    // Create a new string that is the reversal of s1
    String s2 = reverse(s1);

    // Check if the reversal is the same as the original string
    return s2.equals(s1);
  }

  /** Create a new string by eliminating nonalphanumeric chars */
  public static String filter(String s) {
    // Create a string builder
    StringBuilder stringBuilder = new StringBuilder();

    // Examine each char in the string to skip alphanumeric char
    for (int i = 0; i < s.length(); i++) {
      if (Character.isLetterOrDigit(s.charAt(i))) {
         stringBuilder.append(s.charAt(i));
      }
```

check palindrome

add letter or digit

```
40       }
41
42       // Return a new filtered string
43       return stringBuilder.toString();
44     }
45
46     /** Create a new string by reversing a specified string */
47     public static String reverse(String s) {
48       StringBuilder stringBuilder = new StringBuilder(s);
49       stringBuilder.reverse(); // Invoke reverse in StringBuilder
50       return stringBuilder.toString();
51     }
52   }
```

```
Enter a string: ab<c>cb?a ↵Enter
Ignoring nonalphanumeric characters,
is ab<c>cb?a a palindrome? true
```

```
Enter a string: abcc><?cab ↵Enter
Ignoring nonalphanumeric characters,
is abcc><?cab a palindrome? false
```

The **filter(String s)** method (lines 31–44) examines each character in string **s** and copies it to a string builder if the character is a letter or a numeric character. The **filter** method returns the string in the builder. The **reverse(String s)** method (lines 47–51) creates a new string that reverses the specified string **s**. The **filter** and **reverse** methods both return a new string. The original string is not changed.

The program in Listing 5.14 checks whether a string is a palindrome by comparing pairs of characters from both ends of the string. Listing 10.10 uses the **reverse** method in the **StringBuilder** class to reverse the string, then compares whether the two strings are equal to determine whether the original string is a palindrome.

Check Point

10.26 What is the difference between **StringBuilder** and **StringBuffer**?

10.27 How do you create a string builder from a string? How do you return a string from a string builder?

10.28 Write three statements to reverse a string **s** using the **reverse** method in the **StringBuilder** class.

10.29 Write three statements to delete a substring from a string **s** of **20** characters, starting at index **4** and ending with index **10**. Use the **delete** method in the **StringBuilder** class.

10.30 What is the internal storage for characters in a string and a string builder?

10.31 Suppose that **s1** and **s2** are given as follows:

```
StringBuilder s1 = new StringBuilder("Java");
StringBuilder s2 = new StringBuilder("HTML");
```

Show the value of **s1** after each of the following statements. Assume that the statements are independent.

a. `s1.append(" is fun");`

b. `s1.append(s2);`