CHAPTER

13

Abstract Classes and Interfaces

Objectives

- To design and use abstract classes (§13.2).
- To generalize numeric wrapper classes `BigInteger` and `BigDecimal` using the abstract `Number` class (§13.3).
- To process a calendar using the `Calendar` and `GregorianCalendar` classes (§13.4).
- To specify common behavior for objects using interfaces (§13.5).
- To define interfaces and define classes that implement interfaces (§13.5).
- To define a natural order using the `Comparable` interface (§13.6).
- To make objects cloneable using the `Cloneable` interface (§13.7).
- To explore the similarities and differences among concrete classes, abstract classes, and interfaces (§13.8).
- To design the `Rational` class for processing rational numbers (§13.9).
- To design classes that follow the class-design guidelines (§13.10).

13.1 Introduction

Key Point

A superclass defines common behavior for related subclasses. An interface can be used to define common behavior for classes (including unrelated classes).

problem

interface

You can use the **java.util.Arrays.sort** method to sort an array of numbers or strings. Can you apply the same **sort** method to sort an array of geometric objects? In order to write such code, you have to know about interfaces. An *interface* is for defining common behavior for classes (including unrelated classes). Before discussing interfaces, we introduce a closely related subject: abstract classes.

13.2 Abstract Classes

Key Point

An abstract class cannot be used to create objects. An abstract class can contain abstract methods that are implemented in concrete subclasses.

abstract class

In the inheritance hierarchy, classes become more specific and concrete *with each new subclass*. If you move from a subclass back up to a superclass, the classes become more general and less specific. Class design should ensure a superclass contains common features of its subclasses. Sometimes, a superclass is so abstract it cannot be used to create any specific instances. Such a class is referred to as an *abstract class*.

VideoNote

Abstract GeometricObject class

abstract method

abstract modifier

In Chapter 11, **GeometricObject** was defined as the superclass for **Circle** and **Rectangle**. **GeometricObject** models common features of geometric objects. Both **Circle** and **Rectangle** contain the **getArea()** and **getPerimeter()** methods for computing the area and perimeter of a circle and a rectangle. Since you can compute areas and perimeters for all geometric objects, it is better to define the **getArea()** and **getPerimeter()** methods in the **GeometricObject** class. However, these methods cannot be implemented in the **GeometricObject** class because their implementation depends on the specific type of geometric object. Such methods are referred to as *abstract methods* and are denoted using the **abstract** modifier in the method header. After you define the methods in **GeometricObject**, it becomes an abstract class. Abstract classes are denoted using the **abstract** modifier in the class header. In UML graphic notation, the names of abstract classes and their abstract methods are italicized, as shown in Figure 13.1. Listing 13.1 gives the source code for the new **GeometricObject** class.

LISTING 13.1 GeometricObject.java

abstract class

```
public abstract class GeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

  /** Construct a default geometric object */
  protected GeometricObject() {
    dateCreated = new java.util.Date();
  }

  /** Construct a geometric object with color and filled value */
  protected GeometricObject(String color, boolean filled) {
    dateCreated = new java.util.Date();
    this.color = color;
    this.filled = filled;
  }

  /** Return color */
  public String getColor() {
    return color;
```

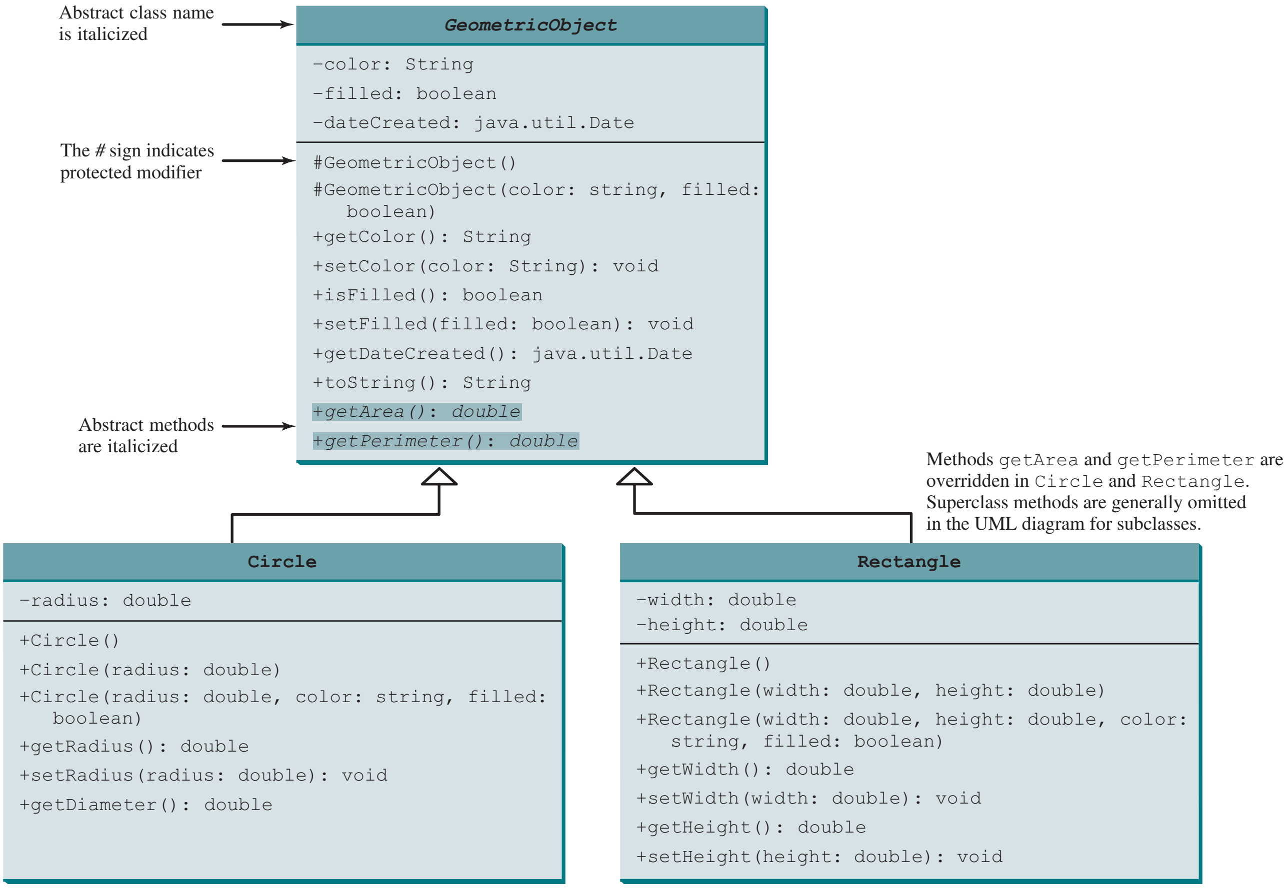

FIGURE 13.1 The new `GeometricObject` class contains abstract methods.

```java
  }

  /** Set a new color */
  public void setColor(String color) {
    this.color = color;
  }

  /** Return filled. Since filled is boolean,
   *  the getter method is named isFilled */
  public boolean isFilled() {
    return filled;
  }

  /** Set a new filled */
  public void setFilled(boolean filled) {
    this.filled = filled;
  }

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

abstract method

abstract method

```java

  @Override
  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;
  }

  /** Abstract method getArea */
  public abstract double getArea();

  /** Abstract method getPerimeter */
  public abstract double getPerimeter();
}
```

Abstract classes are like regular classes, but you cannot create instances of abstract classes using the `new` operator. An abstract method is defined without implementation. Its implementation is provided by the subclasses. A class that contains abstract methods must be defined as abstract.

why protected constructor?

The constructor in the abstract class is defined as protected because it is used only by subclasses. When you create an instance of a concrete subclass, its superclass's constructor is invoked to initialize data fields defined in the superclass.

The `GeometricObject` abstract class defines the common features (data and methods) for geometric objects and provides appropriate constructors. Because you don't know how to compute areas and perimeters of geometric objects, `getArea()` and `getPerimeter()` are defined as abstract methods. These methods are implemented in the subclasses. The implementation of `Circle` and `Rectangle` is the same as in Listings 11.2 and 11.3, except they extend the `GeometricObject` class defined in this chapter. You can see the complete code for these two programs at liveexample.pearsoncmg.com/html/Circle.html and liveexample.pearsoncmg.com/html/Rectangle.html, respectively.

implement `Circle`
implement `Rectangle`

LISTING 13.2 Circle.java

extends abstract `GeometricObject`

```java
public class Circle extends GeometricObject {
  // Same as lines 2-47 in Listing 11.2, so omitted
}
```

LISTING 13.3 Rectangle.java

extends abstract `GeometricObject`

```java
public class Rectangle extends GeometricObject {
  // Same as lines 2-49 in Listing 11.3, so omitted
}
```

13.2.1 Why Abstract Methods?

You may be wondering what advantage is gained by defining the methods `getArea()` and `getPerimeter()` as abstract in the `GeometricObject` class. The example in Listing 13.4 shows the benefits of defining them in the `GeometricObject` class. The program creates two geometric objects, a circle and a rectangle, invokes the `equalArea` method to check whether they have equal areas, and invokes the `displayGeometricObject` method to display them.

LISTING 13.4 TestGeometricObject.java

create a circle
create a rectangle

```java
public class TestGeometricObject {
  /** Main method */
  public static void main(String[] args) {
    // Create two geometric objects
    GeometricObject geoObject1 = new Circle(5);
    GeometricObject geoObject2 = new Rectangle(5, 3);
```

```
 7
 8       System.out.println("The two objects have the same area? " +
 9         equalArea(geoObject1, geoObject2));
10
11       // Display circle
12       displayGeometricObject(geoObject1);
13
14       // Display rectangle
15       displayGeometricObject(geoObject2);
16     }
17
18     /** A method for comparing the areas of two geometric objects */
19     public static boolean equalArea(GeometricObject object1,
20         GeometricObject object2) {
21       return object1.getArea() == object2.getArea();
22     }
23
24     /** A method for displaying a geometric object */
25     public static void displayGeometricObject(GeometricObject object) {
26       System.out.println();
27       System.out.println("The area is " + object.getArea());
28       System.out.println("The perimeter is " + object.getPerimeter());
29     }
30   }
```

equalArea

displayGeometricObject

```
The two objects have the same area? false

The area is 78.53981633974483
The perimeter is 31.41592653589793

The area is 15.0
The perimeter is 16.0
```

The methods `getArea()` and `getPerimeter()` defined in the `GeometricObject` class are overridden in the `Circle` class and the `Rectangle` class. The statements (lines 5–6)

```
GeometricObject geoObject1 = new Circle(5);
GeometricObject geoObject2 = new Rectangle(5, 3);
```

create a new circle and rectangle and assign them to the variables `geoObject1` and `geoObject2`. These two variables are of the `GeometricObject` type.

When invoking `equalArea(geoObject1, geoObject2)` (line 9), the `getArea()` method defined in the `Circle` class is used for `object1.getArea()`, since `geoObject1` is a circle, and the `getArea()` method defined in the `Rectangle` class is used for `object2.getArea()`, since `geoObject2` is a rectangle.

Similarly, when invoking `displayGeometricObject(geoObject1)` (line 12), the methods `getArea()` and `getPerimeter()` defined in the `Circle` class are used, and when invoking `displayGeometricObject(geoObject2)` (line 15), the methods `getArea` and `getPerimeter` defined in the `Rectangle` class are used. The JVM dynamically determines which of these methods to invoke at runtime, depending on the actual object that invokes the method.

Note you could not define the `equalArea` method for comparing whether two geometric objects have the same area if the `getArea` method was not defined in `GeometricObject`, since `object1` and `object2` are of the `GeometricObject` type. Now you have seen the benefits of defining the abstract methods in `GeometricObject`.

why abstract methods?

13.2.2 Interesting Points about Abstract Classes

The following points about abstract classes are worth noting:

abstract method in abstract class

- An abstract method cannot be contained in a nonabstract class. If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined as abstract. In other words, in a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented. Also note abstract methods are nonstatic.

object cannot be created from abstract class

- An abstract class cannot be instantiated using the `new` operator, but you can still define its constructors, which are invoked in the constructors of its subclasses. For instance, the constructors of `GeometricObject` are invoked in the `Circle` class and the `Rectangle` class.

abstract class without abstract method

- A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that doesn't contain any abstract methods. This abstract class is used as a base class for defining subclasses.

concrete method overridden to be abstract

- A subclass can override a method from its superclass to define it as abstract. This is *very unusual*, but it is useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined as abstract.

concrete method overridden to be abstract

- A subclass can be abstract even if its superclass is concrete. For example, the `Object` class is concrete, but its subclasses, such as `GeometricObject`, may be abstract.

abstract class as type

- You cannot create an instance from an abstract class using the `new` operator, but an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of the `GeometricObject` type, is correct:

  ```
  GeometricObject[] objects = new GeometricObject[10];
  ```

  You can then create an instance of `GeometricObject` and assign its reference to the array like this:

  ```
  objects[0] = new Circle();
  ```

Check Point

13.2.1 Which of the following classes defines a legal abstract class?

```
class A {
  abstract void unfinished() {
  }
}
```

(a)

```
public class abstract A {
  abstract void unfinished();
}
```

(b)

```
class A {
  abstract void unfinished();
}
```

(c)

```
abstract class A {
  protected void unfinished();
}
```

(d)

```
abstract class A {
  abstract void unfinished();
}
```

(e)

```
abstract class A {
  abstract int unfinished();
}
```

(f)

13.2.2 The `getArea()` and `getPerimeter()` methods may be removed from the `GeometricObject` class. What are the benefits of defining `getArea()` and `getPerimeter()` as abstract methods in the `GeometricObject` class?

13.2.3 True or false?

a. An abstract class can be used just like a nonabstract class except that you cannot use the `new` operator to create an instance from the abstract class.

b. An abstract class can be extended.

c. A subclass of a nonabstract superclass cannot be abstract.

d. A subclass cannot override a concrete method in a superclass to define it as abstract.

e. An abstract method must be nonstatic.

13.3 Case Study: The Abstract `Number` Class

Key Point

`Number` is an abstract superclass for numeric wrapper classes `BigInteger` and `BigDecimal`.

Section 10.7 introduced numeric wrapper classes and Section 10.9 introduced the `BigInteger` and `BigDecimal` classes. These classes have common methods `byteValue()`, `shortValue()`, `intValue()`, `longValue()`, `floatValue()`, and `doubleValue()` for returning a `byte`, `short`, `int`, `long`, `float`, and `double` value from an object of these classes. These common methods are actually defined in the `Number` class, which is a superclass for the numeric wrapper classes `BigInteger` and `BigDecimal`, as shown in Figure 13.2.

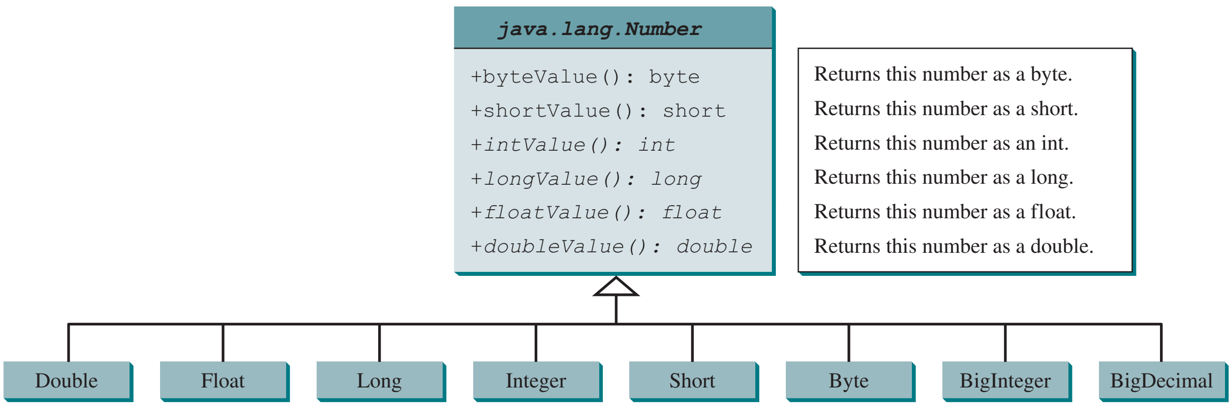

FIGURE 13.2 The `Number` class is an abstract superclass for `Double`, `Float`, `Long`, `Integer`, `Short`, `Byte`, `BigInteger`, and `BigDecimal`.

Since the `intValue()`, `longValue()`, `floatValue()`, and `doubleValue()` methods cannot be implemented in the `Number` class, they are defined as abstract methods in the `Number` class. The `Number` class is therefore an abstract class. The `byteValue()` and `shortValue()` methods are implemented from the `intValue()` method as follows:

```
public byte byteValue() {
  return (byte)intValue();
}

public short shortValue() {
  return (short)intValue();
}
```

With **Number** defined as the superclass for the numeric classes, we can define methods to perform common operations for numbers. Listing 13.5 gives a program that finds the largest number in a list of **Number** objects.

LISTING 13.5 LargestNumber.java

create an array list
add number to list
invoke getLargestNumber
doubleValue

```
1  import java.util.ArrayList;
2  import java.math.*;
3
4  public class LargestNumber {
5    public static void main(String[] args) {
6      ArrayList<Number> list = new ArrayList<>();
7      list.add(45); // Add an integer
8      list.add(3445.53); // Add a double
9      // Add a BigInteger
10     list.add(new BigInteger("3432323234344343101"));
11     // Add a BigDecimal
12     list.add(new BigDecimal("2.0909090989091343433344343"));
13
14     System.out.println("The largest number is " +
15       getLargestNumber(list));
16   }
17
18   public static Number getLargestNumber(ArrayList<Number> list) {
19     if (list == null || list.size() == 0)
20       return null;
21
22     Number number = list.get(0);
23     for (int i = 1; i < list.size(); i++)
24       if (number.doubleValue() < list.get(i).doubleValue())
25         number = list.get(i);
26
27     return number;
28   }
29 }
```

```
The largest number is 3432323234344343101
```

The program creates an **ArrayList** of **Number** objects (line 6). It adds an **Integer** object, a **Double** object, a **BigInteger** object, and a **BigDecimal** object to the list (lines 7–12). Note **45** is automatically converted into an **Integer** object and added to the list in line 7, and **3445.53** is automatically converted into a **Double** object and added to the list in line 8 using autoboxing.

Invoking the **getLargestNumber** method returns the largest number in the list (line 15). The **getLargestNumber** method returns **null** if the list is **null** or the list size is **0** (lines 19 and 20). To find the largest number in the list, the numbers are compared by invoking their **doubleValue()** method (line 24). The **doubleValue()** method is defined in the **Number** class and implemented in the concrete subclass of **Number**. If a number is an **Integer** object, the **Integer**'s **doubleValue()** is invoked. If a number is a **BigDecimal** object, the **BigDecimal**'s **doubleValue()** is invoked.

If the **doubleValue()** method was not defined in the **Number** class, you will not be able to find the largest number among different types of numbers using the **Number** class.

Check Point

13.3.1 Why do the following two lines of code compile but cause a runtime error?

```
Number numberRef = Integer.valueOf(0);
Double doubleRef = (Double)numberRef;
```

13.3.2 Why do the following two lines of code compile but cause a runtime error?

```
Number[] numberArray = Integer[2];
numberArray[0] = Double.valueOf(1.5);
```

13.3.3 Show the output of the following code:

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

13.3.4 What is wrong in the following code? (Note the `compareTo` method for the `Integer` and `Double` classes was introduced in Section 10.7.)

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

13.3.5 What is wrong in the following code?

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

13.4 Case Study: `Calendar` and `GregorianCalendar`

`GregorianCalendar` is a concrete subclass of the abstract class `Calendar`.

Key Point

VideoNote
Calendar and GregorianCalendar classes

An instance of `java.util.Date` represents a specific instant in time with millisecond precision. `java.util.Calendar` is an abstract base class for extracting detailed calendar information, such as the year, month, date, hour, minute, and second. Subclasses of `Calendar` can implement specific calendar systems, such as the Gregorian calendar, the lunar calendar, and the Jewish calendar. Currently, `java.util.GregorianCalendar` for the Gregorian calendar is supported in Java, as shown in Figure 13.3. The `add` method is abstract in the `Calendar` class because its implementation is dependent on a concrete calendar system.

abstract `add` method

construct calendar

You can use `new GregorianCalendar()` to construct a default `GregorianCalendar` with the current time and `new GregorianCalendar(year, month, date)` to construct a `GregorianCalendar` with the specified `year`, `month`, and `date`. The `month` parameter is `0`-based—that is, `0` is for January.

`get(field)`

The `get(int field)` method defined in the `Calendar` class is useful for extracting the date and time information from a `Calendar` object. The fields are defined as constants, as shown in Table 13.1.

Listing 13.6 gives an example that displays the date and time information for the current time.

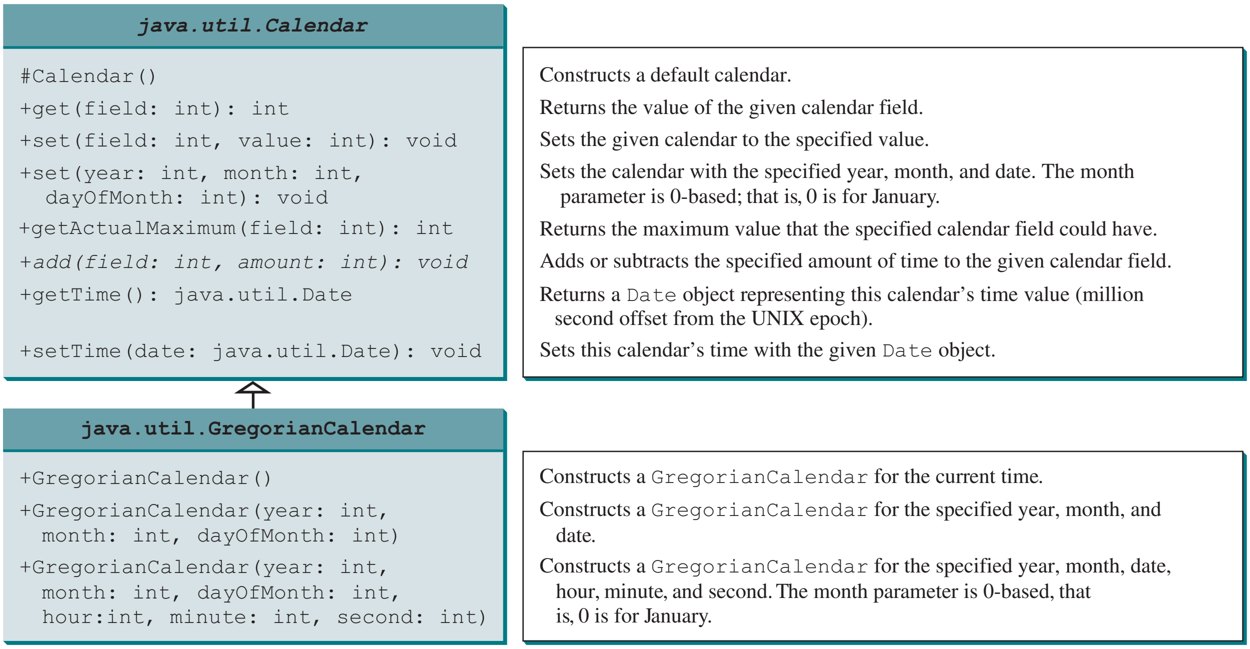

FIGURE 13.3 The abstract `Calendar` class defines common features of various calendars.

TABLE 13.1 Field Constants in the `Calendar` Class

Constant	*Description*
`YEAR`	The year of the calendar.
`MONTH`	The month of the calendar, with 0 for January.
`DATE`	The day of the calendar.
`HOUR`	The hour of the calendar (12-hour notation).
`HOUR_OF_DAY`	The hour of the calendar (24-hour notation).
`MINUTE`	The minute of the calendar.
`SECOND`	The second of the calendar.
`DAY_OF_WEEK`	The day number within the week, with 1 for Sunday.
`DAY_OF_MONTH`	Same as DATE.
`DAY_OF_YEAR`	The day number in the year, with 1 for the first day of the year.
`WEEK_OF_MONTH`	The week number within the month, with 1 for the first week.
`WEEK_OF_YEAR`	The week number within the year, with 1 for the first week.
`AM_PM`	Indicator for AM or PM (0 for AM and 1 for PM).

LISTING 13.6 TestCalendar.java

calendar for current time

extract fields in calendar

```
import java.util.*;

public class TestCalendar {
  public static void main(String[] args) {
    // Construct a Gregorian calendar for the current date and time
    Calendar calendar = new GregorianCalendar();
    System.out.println("Current time is " + new Date());
    System.out.println("YEAR: " + calendar.get(Calendar.YEAR));
```

```
      System.out.println("MONTH: " + calendar.get(Calendar.MONTH));
      System.out.println("DATE: " + calendar.get(Calendar.DATE));
      System.out.println("HOUR: " + calendar.get(Calendar.HOUR));
      System.out.println("HOUR_OF_DAY: " +
        calendar.get(Calendar.HOUR_OF_DAY));
      System.out.println("MINUTE: " + calendar.get(Calendar.MINUTE));
      System.out.println("SECOND: " + calendar.get(Calendar.SECOND));
      System.out.println("DAY_OF_WEEK: " +
        calendar.get(Calendar.DAY_OF_WEEK));
      System.out.println("DAY_OF_MONTH: " +
        calendar.get(Calendar.DAY_OF_MONTH));
      System.out.println("DAY_OF_YEAR: " +
        calendar.get(Calendar.DAY_OF_YEAR));
      System.out.println("WEEK_OF_MONTH: " +
        calendar.get(Calendar.WEEK_OF_MONTH));
      System.out.println("WEEK_OF_YEAR: " +
        calendar.get(Calendar.WEEK_OF_YEAR));
      System.out.println("AM_PM: " + calendar.get(Calendar.AM_PM));

      // Construct a calendar for December 25, 1997
      Calendar calendar1 = new GregorianCalendar(1997, 11, 25);
      String[] dayNameOfWeek = {"Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"};
      System.out.println("December 25, 1997 is a " +
        dayNameOfWeek[calendar1.get(Calendar.DAY_OF_WEEK) - 1]);
    }
  }
```

create a calendar

```
Current time is Tue Sep 22 12:55:56 EDT 2015
YEAR: 2015
MONTH: 8
DATE: 22
HOUR: 0
HOUR_OF_DAY: 12
MINUTE: 55
SECOND: 56
DAY_OF_WEEK: 3
DAY_OF_MONTH: 22
DAY_OF_YEAR: 265
WEEK_OF_MONTH: 4
WEEK_OF_YEAR: 39
AM_PM: 1
December 25, 1997 is a Thursday
```

`set(field, value)`

The `set(int field, value)` method defined in the `Calendar` class can be used to set a field. For example, you can use `calendar.set(Calendar.DAY_OF_MONTH, 1)` to set the `calendar` to the first day of the month.

`add(field, amount)`

The `add(field, value)` method adds the specified amount to a given field. For example, `add(Calendar.DAY_OF_MONTH, 5)` adds five days to the current time of the calendar. `add(Calendar.DAY_OF_MONTH, -5)` subtracts five days from the current time of the calendar.

`getActualMaximum(field)`

To obtain the number of days in a month, use `calendar.getActualMaximum(Calendar.DAY_OF_MONTH)`. For example, if the `calendar` was for March, this method would return `31`.

setTime(date)
getTime()

You can set a time represented in a **Date** object for the **calendar** by invoking **calendar.setTime(date)** and retrieve the time by invoking **calendar.getTime()**.

Check Point

13.4.1 Can you create a **Calendar** object using the **Calendar** class?

13.4.2 Which method in the **Calendar** class is abstract?

13.4.3 How do you create a **Calendar** object for the current time?

13.4.4 For a **Calendar** object **c**, how do you get its year, month, date, hour, minute, and second?

13.5 Interfaces

Key Point

An interface is a class-like construct for defining common operations for objects.

VideoNote
The concept of interface

In many ways an interface is similar to an abstract class, but its intent is to specify common behavior for objects of related classes or unrelated classes. For example, using appropriate interfaces, you can specify that the objects are comparable, edible, and/or cloneable.

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
modifier interface InterfaceName {
  /** Constant declarations */
  /** Abstract method signatures */
}
```

Here is an example of an interface:

```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class. You can use an interface more or less the same way you use an abstract class. For example, you can use an interface as a data type for a reference variable, as the result of casting, and so on. As with an abstract class, you cannot create an instance from an interface using the **new** operator.

You can use the **Edible** interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the **implements** keyword. For example, the classes **Chicken** and **Fruit** in Listing 13.7 (lines 30 and 49) implement the **Edible** interface. The relationship between the class and the interface is known as *interface inheritance*. Since interface inheritance and class inheritance are essentially the same, we will simply refer to both as *inheritance*.

interface inheritance

LISTING 13.7 TestEdible.java

```
1  public class TestEdible {
2    public static void main(String[] args) {
3      Object[] objects = {new Tiger(), new Chicken(), new Apple()};
4      for (int i = 0; i < objects.length; i++) {
5        if (objects[i] instanceof Edible)
6          System.out.println(((Edible)objects[i]).howToEat());
7
8        if (objects[i] instanceof Animal) {
9          System.out.println(((Animal)objects[i]).sound());
10       }
11     }
12   }
13 }
```

```

abstract class Animal {
  private double weight;

  public double getWeight() {
    return weight;
  }

  public void setWeight(double weight) {
    this.weight = weight;
  }

  /** Return animal sound */
  public abstract String sound();
}

class Chicken extends Animal implements Edible {
  @Override
  public String howToEat() {
    return "Chicken: Fry it";
  }

  @Override
  public String sound() {
    return "Chicken: cock-a-doodle-doo";
  }
}

class Tiger extends Animal {
  @Override
  public String sound() {
    return "Tiger: RROOAARR";
  }
}

abstract class Fruit implements Edible {
  // Data fields, constructors, and methods omitted here
}

class Apple extends Fruit {
  @Override
  public String howToEat() {
    return "Apple: Make apple cider";
  }
}

class Orange extends Fruit {
  @Override
  public String howToEat() {
    return "Orange: Make orange juice";
  }
}
```

Animal class
implements Edible
howToEat()
Tiger class
implements Edible
Apple class
Orange class

```
Tiger: RROOAARR
Chicken: Fry it
Chicken: cock-a-doodle-doo
Apple: Make apple cider
```

This example uses several classes and interfaces. Their inheritance relationship is shown in Figure 13.4.

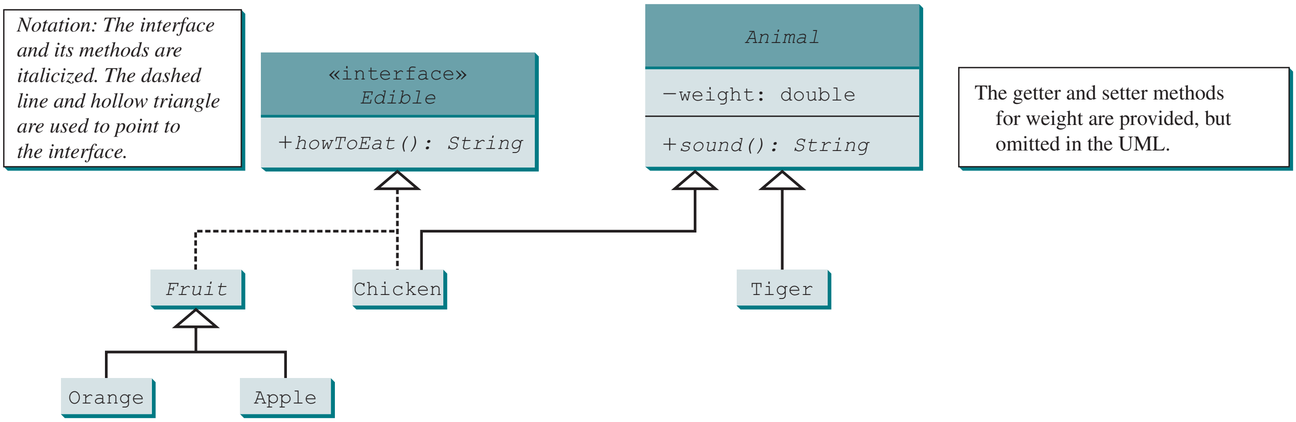

FIGURE 13.4 `Edible` is a supertype for `Chicken` and `Fruit`. `Animal` is a supertype for `Chicken` and `Tiger`. `Fruit` is a supertype for `Orange` and `Apple`.

The `Animal` class defines the `weight` property with its getter and setter methods (lines 16–24) and the `sound` method (line 27). The `sound` method is an abstract method and will be implemented by a concrete `animal` class.

The `Chicken` class implements `Edible` to specify that chickens are edible. When a class implements an interface, it implements all the methods defined in the interface. The `Chicken` class implements the `howToEat` method (lines 32–34). `Chicken` also extends `Animal` to implement the `sound` method (lines 37–39).

The `Fruit` class implements `Edible`. Since it does not implement the `howToEat` method, `Fruit` must be defined as `abstract` (line 49). The concrete subclasses of `Fruit` must implement the `howToEat` method. The `Apple` and `Orange` classes implement the `howToEat` method (lines 55 and 62).

The `main` method creates an array with three objects for `Tiger`, `Chicken`, and `Apple` (line 3) and invokes the `howToEat` method if the element is edible (line 6), and the `sound` method if the element is an animal (line 9).

common behavior

In essence, the `Edible` interface defines common behavior for edible objects. All edible objects have the `howToEat` method.

omit modifiers

Note

The modifiers ***`public static final`*** on data fields and the modifiers ***`public abstract`*** on methods can be omitted in an interface. Therefore, the following interface definitions are equivalent:

```
public interface T {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T {
  int K = 1;

  void p();
}
```

Although the `public` modifier may be omitted for a method defined in the interface, the method must be defined `public` when it is implemented in a subclass.

default methods

Note

Java 8 introduced default interface methods using the keyword `default`. A default method provides a default implementation for the method in the interface. A class that implements the interface may simply use the default implementation for the method or override the method with a new implementation. This feature enables you to add a new method to an existing interface with a default implementation without having to rewrite the code for the existing classes that implement this interface.

Java 8 also permits public static methods in an interface. A public static method in an interface can be used just like a public static method in a class.

In Java 9, you can also use private methods in an interface. These methods are used for implementing the default methods and public static methods. Here is an example of defining default methods, static methods, and private methods in an interface:

```
public interface Java89Interface {
  /** default method in Java 8*/
  public default void doSomething() {
    System.out.println("Do something");
  }

  /** static method in Java 8*/
  public static int getAValue() {
    return 0;
  }

  /** private static method Java 9 */
  private static int getAStaticValue() {
    return 0;
  }

  /** private instance method Java 9 */
  private void performPrivateAction() {
  }
}
```

Check Point

13.5.1 Suppose **A** is an interface. Can you create an instance using **new A()**?

13.5.2 Suppose **A** is an interface. Can you declare a reference variable **x** with type **A** like this?

```
A x;
```

13.5.3 Which of the following is a correct interface?

```
interface A {
  void print() { }
}
```

(a)

```
abstract interface A {
  abstract void print() { }
}
```

(b)

```
abstract interface A {
  print();
}
```

(c)

```
interface A {
  void print();
}
```

(d)

```
interface A {
  default void print() {
  }
}
```

(e)

```
interface A {
  static int get() {
    return 0;
  }
}
```

(f)

13.5.4 Show the error in the following code:

```
interface A {
  void m1();
}

class B implements A {
  void m1() {
    System.out.println("m1");
  }
}
```

13.6 The `Comparable` Interface

Key Point

The `Comparable` interface defines the `compareTo` method for comparing objects.

Suppose you want to design a generic method to find the larger of two objects of the same type, such as two students, two dates, two circles, two rectangles, or two squares. In order to accomplish this, the two objects must be comparable, so the common behavior for the objects must be comparable. Java provides the `Comparable` interface for this purpose. The interface is defined as follows:

java.lang.Comparable

```
// Interface for comparing objects, defined in java.lang
package java.lang;

public interface Comparable<E> {
  public int compareTo(E o);
}
```

The `compareTo` method determines the order of this object with the specified object `o` and returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than `o`.

The `Comparable` interface is a generic interface. The generic type `E` is replaced by a concrete type when implementing this interface. Many classes in the Java library implement `Comparable` to define a natural order for objects. The classes `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `BigInteger`, `BigDecimal`, `Calendar`, `String`, and `Date` all implement the `Comparable` interface. For example, the `Integer`, `BigInteger`, `String`, and `Date` classes are defined as follows in the Java API:

```
public final class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```

```
public class BigInteger extends Number
    implements Comparable<Biginteger> {
  // class body omitted

  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```

```
public final class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```

```
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```

Thus, numbers are comparable, strings are comparable, and so are dates. You can use the `compareTo` method to compare two numbers, two strings, and two dates. For example, the following code:

```
System.out.println(Integer.valueOf(3).compareTo(5));
System.out.println("ABC".compareTo("ABC"));
java.util.Date date1 = new java.util.Date(2013, 1, 1);
java.util.Date date2 = new java.util.Date(2012, 1, 1);
System.out.println(date1.compareTo(date2));
```

displays

```
-1
 0
 1
```

Line 1 displays a negative value since **3** is less than **5**. Line 2 displays zero since **ABC** is equal to **ABC**. Line 5 displays a positive value since **date1** is greater than **date2**.

Let **n** be an **Integer** object, **s** be a **String** object, and **d** be a **Date** object. All the following expressions are **true**:

`n instanceof Integer`	`s instanceof String`	`d instanceof Date`
`n instanceof Object`	`s instanceof Object`	`d instanceof Object`
`n instanceof Comparable`	`s instanceof Comparable`	`d instanceof Comparable`

Since all **Comparable** objects have the **compareTo** method, the **java.util.Arrays.sort(Object[])** method in the Java API uses the **compareTo** method to compare and sorts the objects in an array, provided the objects are instances of the **Comparable** interface. Listing 13.8 gives an example of sorting an array of strings and an array of **BigInteger** objects.

LISTING 13.8 SortComparableObjects.java

```
1  import java.math.*;
2
3  public class SortComparableObjects {
4    public static void main(String[] args) {
5      String[] cities = {"Savannah", "Boston", "Atlanta", "Tampa"};
6      java.util.Arrays.sort(cities);
7      for (String city: cities)
8        System.out.print(city + " ");
9      System.out.println();
10
11     BigInteger[] hugeNumbers = {new BigInteger("2323231092923992"),
12       new BigInteger("432232323239292"),
13       new BigInteger("54623239292")};
14     java.util.Arrays.sort(hugeNumbers);
15     for (BigInteger number: hugeNumbers)
16       System.out.print(number + " ");
17   }
18 }
```

create an array
sort the array
create an array
sort the array

```
Atlanta Boston Savannah Tampa
54623239292 432232323239292 2323231092923992
```

The program creates an array of strings (line 5) and invokes the **sort** method to sort the strings (line 6). The program creates an array of **BigInteger** objects (lines 11–13) and invokes the **sort** method to sort the **BigInteger** objects (line 14).

You cannot use the **sort** method to sort an array of **Rectangle** objects because **Rectangle** does not implement **Comparable**. However, you can define a new rectangle class that implements **Comparable**. The instances of this new class are comparable. Let this new class be named **ComparableRectangle**, as shown in Listing 13.9.

LISTING 13.9 ComparableRectangle.java

```
public class ComparableRectangle extends Rectangle
    implements Comparable<ComparableRectangle> {
  /** Construct a ComparableRectangle with specified properties */
  public ComparableRectangle(double width, double height) {
    super(width, height);
  }

  @Override // Implement the compareTo method defined in Comparable
```

implements **Comparable**

implement compareTo

implement toString

```
  public int compareTo(ComparableRectangle o) {
    if (getArea() > o.getArea())
      return 1;
    else if (getArea() < o.getArea())
      return -1;
    else
      return 0;
  }

  @Override // Implement the toString method in GeometricObject
  public String toString() {
    return "Width: " + getWidth() + " Height: " + getHeight() +
      "Area: " + getArea();
  }
}
```

ComparableRectangle extends **Rectangle** and implements **Comparable**, as shown in Figure 13.5. The keyword **implements** indicates that **ComparableRectangle** inherits all the constants from the **Comparable** interface and implements the methods in the interface. The **compareTo** method compares the areas of two rectangles. An instance of **ComparableRectangle** is also an instance of **Rectangle**, **GeometricObject**, **Object**, and **Comparable**.

Notation:
The interface name and the method names are italicized. The dashed line and hollow triangle are used to point to the interface.

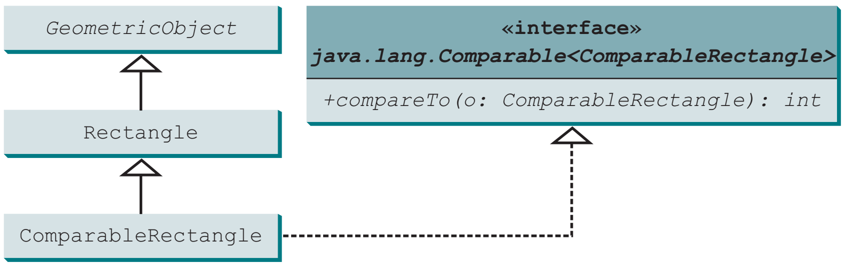

FIGURE 13.5 **ComparableRectangle** extends **Rectangle** and implements **Comparable**.

You can now use the **sort** method to sort an array of **ComparableRectangle** objects, as in Listing 13.10.

LISTING 13.10 SortRectangles.java

create an array

sort the array

```
public class SortRectangles {
  public static void main(String[] args) {
    ComparableRectangle[] rectangles = {
      new ComparableRectangle(3.4, 5.4),
      new ComparableRectangle(13.24, 55.4),
      new ComparableRectangle(7.4, 35.4),
      new ComparableRectangle(1.4, 25.4)};
    java.util.Arrays.sort(rectangles);
    for (Rectangle rectangle: rectangles) {
      System.out.print(rectangle + " ");
      System.out.println();
    }
  }
}
```

```
Width: 3.4 Height: 5.4 Area: 18.36
Width: 1.4 Height: 25.4 Area: 35.559999999999995
Width: 7.4 Height: 35.4 Area: 261.96
Width: 13.24 Height: 55.4 Area: 733.496
```

An interface provides another form of generic programming. It would be difficult to use a generic `sort` method to sort the objects without using an interface in this example, because multiple inheritance would be necessary to inherit `Comparable` and another class, such as `Rectangle`, at the same time.

benefits of interface

The `Object` class contains the `equals` method, which is intended for the subclasses of the `Object` class to override in order to compare whether the contents of the objects are the same. Suppose the `Object` class contains the `compareTo` method, as defined in the `Comparable` interface; the `sort` method can be used to compare a list of *any* objects. Whether a `compareTo` method should be included in the `Object` class is debatable. Since the `compareTo` method is not defined in the `Object` class, the `Comparable` interface is defined in Java to enable objects to be compared if they are instances of the `Comparable` interface. `compareTo` should be consistent with `equals`. That is, for two objects `o1` and `o2`, `o1.compareTo(o2) == 0` if and only if `o1.equals(o2)` is `true`. Therefore, you should also override the `equals` method in the `ComparableRectangle` class to return `true` if two rectangles have the same area.

Check Point

13.6.1 True or false? If a class implements `Comparable`, the object of the class can invoke the `compareTo` method.

13.6.2 Which of the following is the correct method header for the `compareTo` method in the `String` class?

```
public int compareTo(String o)
public int compareTo(Object o)
```

13.6.3 Can the following code be compiled? Why?

```
Integer n1 = 3;
Object n2 = 4;
System.out.println(n1.compareTo(n2));
```

13.6.4 You can define the `compareTo` method in a class without implementing the `Comparable` interface. What are the benefits of implementing the `Comparable` interface?

13.6.5 What is wrong in the following code?

```
public class Test {
  public static void main(String[] args) {
    Person[] persons = {new Person(3), new Person(4), new Person(1)};
    java.util.Arrays.sort(persons);
  }
}

class Person {
  private int id;

  Person(int id) {
    this.id = id;
  }
}
```

13.6.6 Simplify the code in lines 10–15 in Listing 13.9 using one line of code. Also override the `equals` method in this class.

13.6.7 Listing 13.5 has an error. If you add `list.add(new BigInteger("3432323234344343102"));` in line 11, you will see the result is incorrect. This is due to the fact that a double value can have up to 17 significant digits. When invoking `doubleValue()` on a `BigInteger` object in line 24, precision is lost. Fix the error by converting the numbers into `BigDecimal`, and compare them using the `compareTo` method in line 24.

13.7 The `Cloneable` Interface

Key Point

The `Cloneable` interface specifies that an object can be cloned.

Often, it is desirable to create a copy of an object. To do this, you need to use the `clone` method and understand the `Cloneable` interface.

An interface contains constants and abstract methods, but the `Cloneable` interface is a special case. The `Cloneable` interface in the `java.lang` package is defined as follows:

java.lang.Cloneable

```
package java.lang;

public interface Cloneable {
}
```

marker interface

This interface is empty. An interface with an empty body is referred to as a *marker interface*. A marker interface is used to denote that a class possesses certain desirable properties. A class that implements the `Cloneable` interface is marked cloneable, and its objects can be cloned using the `clone()` method defined in the `Object` class.

Many classes in the Java library (e.g., `Date`, `Calendar` and `ArrayList`) implement `Cloneable`. Thus, the instances of these classes can be cloned. For example, the following code:

```
1  Calendar calendar = new GregorianCalendar(2013, 2, 1);
2  Calendar calendar1 = calendar;
3  Calendar calendar2 = (Calendar)calendar.clone();
4  System.out.println("calendar == calendar1 is " +
5    (calendar == calendar1));
6  System.out.println("calendar == calendar2 is " +
7    (calendar == calendar2));
8  System.out.println("calendar.equals(calendar2) is " +
9    calendar.equals(calendar2));
```

displays

```
calendar == calendar1 is true
calendar == calendar2 is false
calendar.equals(calendar2) is true
```

In the preceding code, line 2 copies the reference of `calendar` to `calendar1`, so `calendar` and `calendar1` point to the same `Calendar` object. Line 3 creates a new object that is the clone of `calendar` and assigns the new object's reference to `calendar2`. `calendar2` and `calendar` are different objects with the same contents.

The following code:

```
ArrayList<Double> list1 = new ArrayList<>();
list1.add(1.5);
list1.add(2.5);
list1.add(3.5);
ArrayList<Double> list2 = (ArrayList<Double>)list1.clone();
ArrayList<Double> list3 = list1;
list2.add(4.5);
list3.remove(1.5);
System.out.println("list1 is " + list1);
System.out.println("list2 is " + list2);
System.out.println("list3 is " + list3);
```

displays

```
list1 is [2.5, 3.5]
list2 is [1.5, 2.5, 3.5, 4.5]
list3 is [2.5, 3.5]
```

In the preceding code, line 5 creates a new object that is the clone of **list1** and assigns the new object's reference to **list2**. **list2** and **list1** are different objects with the same contents. Line 6 copies the reference of **list1** to **list3**, so **list1** and **list3** point to the same **ArrayList** object. Line 7 adds **4.5** into **list2**. Line 8 removes **1.5** from **list3**. Since **list1** and **list3** point to the same **ArrayList**, lines 9 and 11 display the same content.

clone arrays

You can clone an array using the **clone** method. For example, the following code:

```
int[] list1 = {1, 2};
int[] list2 = list1.clone();
list1[0] = 7;
list2[1] = 8;
System.out.println("list1 is " + list1[0] + ", " + list1[1]);
System.out.println("list2 is " + list2[0] + ", " + list2[1]);
```

displays

```
list1 is 7, 2
list2 is 1, 8
```

Note the return type of the **clone()** method for an array is the same as the type of the array. For example, the return type for **list1.clone()** is **int[]** since **list1** is of the type **int[]**.

how to implement Cloneable

To define a custom class that implements the **Cloneable** interface, the class must override the **clone()** method in the **Object** class. Listing 13.11 defines a class named **House** that implements **Cloneable** and **Comparable**.

LISTING 13.11 House.java

```
public class House implements Cloneable, Comparable<House> {
  private int id;
  private double area;
  private java.util.Date whenBuilt;

  public House(int id, double area) {
    this.id = id;
    this.area = area;
    whenBuilt = new java.util.Date();
  }

  public int getId() {
    return id;
  }

  public double getArea() {
    return area;
  }

  public java.util.Date getWhenBuilt() {
    return whenBuilt;
  }
```

This exception is thrown if House does not implement Cloneable

```
23
24    @Override /** Override the protected clone method defined in
25      the Object class, and strengthen its accessibility */
26    public Object clone() {
27      try {
28        return super.clone();
29      }
30      catch (CloneNotSupportedException ex) {
31        return null;
32      }
33    }
34
35    @Override // Implement the compareTo method defined in Comparable
36    public int compareTo(House o) {
37      if (area > o.area)
38        return 1;
39      else if (area < o.area)
40        return -1;
41      else
42        return 0;
43    }
44  }
```

The **House** class implements the **clone** method (lines 26–33) defined in the **Object** class. The header for the **clone** method defined in the **Object** class is

```
protected native Object clone() throws CloneNotSupportedException;
```

The keyword **native** indicates that this method is not written in Java, but is implemented in the JVM for the native platform. The keyword **protected** restricts the method to be accessed in the same package or in a subclass. For this reason, the **House** class must override the method and change the visibility modifier to **public** so the method can be used in any package. Since the **clone** method implemented for the native platform in the **Object** class performs the task of cloning objects, the **clone** method in the **House** class simply invokes **super.clone()**.

CloneNotSupported-Exception

The **clone** method defined in the **Object** class throws **CloneNotSupportedException** if the object is not a type of **Cloneable**. Since we catch the exception in the method (lines 30–32), there is no need to declare it in the **clone()** method header.

The **House** class implements the **compareTo** method (lines 36–43) defined in the **Comparable** interface. The method compares the areas of two houses.

You can now create an object of the **House** class and create an identical copy from it, as follows:

```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```

shallow copy

deep copy

house1 and **house2** are two different objects with identical contents. The **clone** method in the **Object** class copies each field from the original object to the target object. If the field is of a primitive type, its value is copied. For example, the value of **area** (**double** type) is copied from **house1** to **house2**. If the field is of an object, the reference of the field is copied. For example, the field **whenBuilt** is of the **Date** class, so its reference is copied into **house2**, as shown in Figure 13.6a. Therefore, **house1.whenBuilt == house2.whenBuilt** is true, although **house1 == house2** is false. This is referred to as a *shallow copy* rather than a *deep copy*, meaning if the field is of an object type, the object's reference is copied rather than its contents.

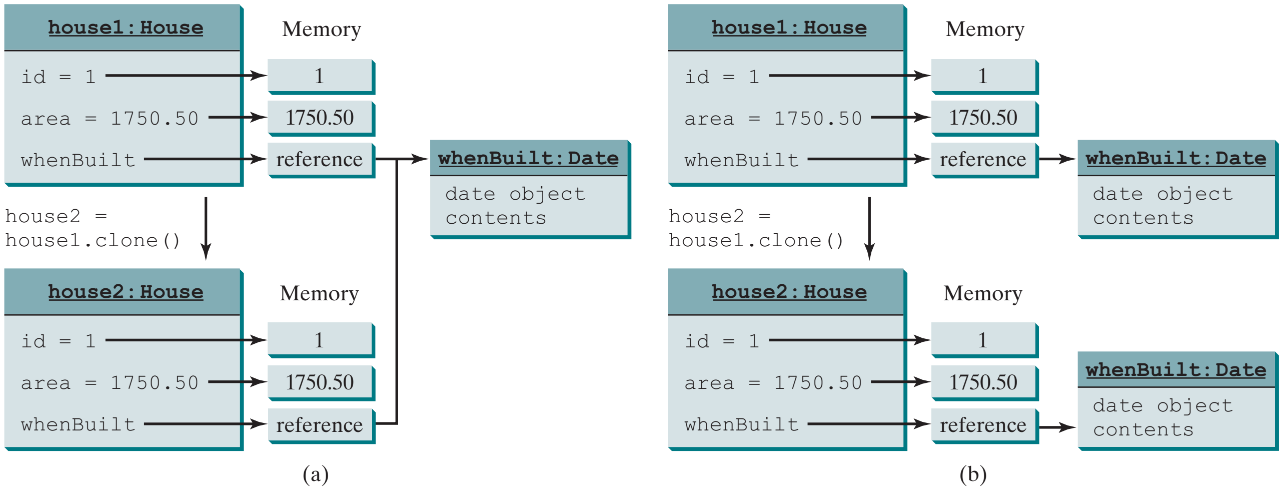

FIGURE 13.6 (a) The default `clone` method performs a shallow copy. (b) The custom `clone` method performs a deep copy.

deep copy

To perform a deep copy for a `House` object, replace the `clone()` method in lines 26–33 with the following code: (For the complete code, see liveexample.pearsoncmg.com/text/House.txt.)

```
public Object clone() throws CloneNotSupportedException {
  // Perform a shallow copy
  House houseClone = (House)super.clone();
  // Deep copy on whenBuilt
  houseClone.whenBuilt = (java.util.Date)(whenBuilt.clone());
  return houseClone;
}
```

or

```
public Object clone() {
  try {
    // Perform a shallow copy
    House houseClone = (House)super.clone();
    // Deep copy on whenBuilt
    houseClone.whenBuilt = (java.util.Date)(whenBuilt.clone());
    return houseClone;
  }
  catch (CloneNotSupportedException ex) {
    return null;
  }
}
```

Now, if you clone a `House` object in the following code:

```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```

`house1.whenBuilt` == `house2.whenBuilt` will be `false`. `house1` and `house2` contain two different `Date` objects, as shown in Figure 13.6b.

Several questions arise from the `clone` method and `Cloneable` interface.

First, why is the `clone` method in the `Object` class defined protected, not public? Not every object can be cloned. The designer of Java purposely forces the subclasses to override it if an object of the subclass is cloneable.

Second, why is the `clone` method not defined in the `Cloneable` interface? Java provides a native method that performs a shallow copy to clone an object. Since a method in an interface is abstract, this native method cannot be implemented in the interface. Therefore, the designer of Java decided to define and implement the native `clone` method in the `Object` class.

Third, why doesn't the `Object` class implement the `Cloneable` interface? The answer is the same as in the first question.

Fourth, what would happen if the `House` class did not implement `Cloneable` in line 1 of Listing 13.11? `house1.clone()` would return `null` because `super.clone()` in line 28 would throw a `CloneNotSupportedException`.

Fifth, you may implement the `clone` method in the `House` class without invoking the clone method in the `Object` class as follows:

```
public Object clone() {
  // Perform a shallow copy
  House houseClone = new House(id, area);

  // Deep copy on whenBuilt
  houseClone.whenBuilt = new Date();
  houseClone.getWhenBuilt().setTime(whenBuilt.getTime());

  return houseClone;
}
```

In this case, the `House` class does not need to implement the `Cloneable` interface, and you have to make sure all the data fields are copied correctly. Using the `clone()` method in the `Object` class relieves you from manually copying the data fields. The `clone` method in the `Object` class automatically performs a shallow copy of all the data fields.

Check Point

13.7.1 Can a class invoke `super.clone()` when implementing the `clone()` method if the class does not implement `java.lang.Cloneable`? Does the `Date` class implement `Cloneable`?

13.7.2 What would happen if the `House` class (defined in Listing 13.11) did not override the `clone()` method or if `House` did not implement `java.lang.Cloneable`?

13.7.3 Show the output of the following code:

```
java.util.Date date = new java.util.Date();
java.util.Date date1 = date;
java.util.Date date2 = (java.util.Date)(date.clone());
System.out.println(date == date1);
System.out.println(date == date2);
System.out.println(date.equals(date2));
```

13.7.4 Show the output of the following code:

```
ArrayList<String> list = new ArrayList<>();
list.add("New York");
ArrayList<String> list1 = list;
ArrayList<String> list2 = (ArrayList<String>)(list.clone());
list.add("Atlanta");
System.out.println(list == list1);
System.out.println(list == list2);
System.out.println("list is " + list);
System.out.println("list1 is " + list1);
System.out.println("list2.get(0) is " + list2.get(0));
System.out.println("list2.size() is " + list2.size());
```

13.7.5 What is wrong in the following code?

```
public class Test {
  public static void main(String[] args) {
    GeometricObject x = new Circle(3);
    GeometricObject y = x.clone();
    System.out.println(x == y);
  }
}
```

13.7.6 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    House house1 = new House(1, 1750, 50);
    House house2 = (House)house1.clone();
    System.out.println(house1.equals(house2);
  }
}
```

13.8 Interfaces vs. Abstract Classes

A class can implement multiple interfaces, but it can only extend one superclass.

Key Point

An interface can be used more or less the same way as an abstract class, but defining an interface is different from defining an abstract class. Table 13.2 summarizes the differences.

TABLE 13.2 Interfaces vs. Abstract Classes

	Variables	*Constructors*	*Methods*
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be `public static final`.	No constructors. An interface cannot be instantiated using the new operator.	May contain public abstract instance methods, public default, and public static methods.

Java allows only *single inheritance* for class extension, but allows *multiple extensions* for interfaces. For example,

single inheritance
multiple inheritance

```
public class NewClass extends BaseClass
    implements Interface1, ... , InterfaceN {
  ...
}
```

An interface can inherit other interfaces using the `extends` keyword. Such an interface is called a *subinterface*. For example, `NewInterface` in the following code is a subinterface of `Interface1`, . . . , and `InterfaceN`.

subinterface

```
public interface NewInterface extends Interface1, ... , InterfaceN {
  // constants and abstract methods
}
```

A class implementing `NewInterface` must implement the abstract methods defined in `NewInterface`, `Interface1`, . . . , and `InterfaceN`. An interface can extend other interfaces, but not classes. A class can extend its superclass and implement multiple interfaces.

All classes share a single root, the `Object` class, but there is no single root for interfaces. Like a class, an interface also defines a type. A variable of an interface type can reference any instance of the class that implements the interface. If a class implements an interface, the interface is like a superclass for the class. You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa. For example, suppose `c` is an instance of `Class2` in Figure 13.7. `c` is also an instance of `Object`, `Class1`, `Interface1`, `Interface1_1`, `Interface1_2`, `Interface2_1`, and `Interface2_2`.

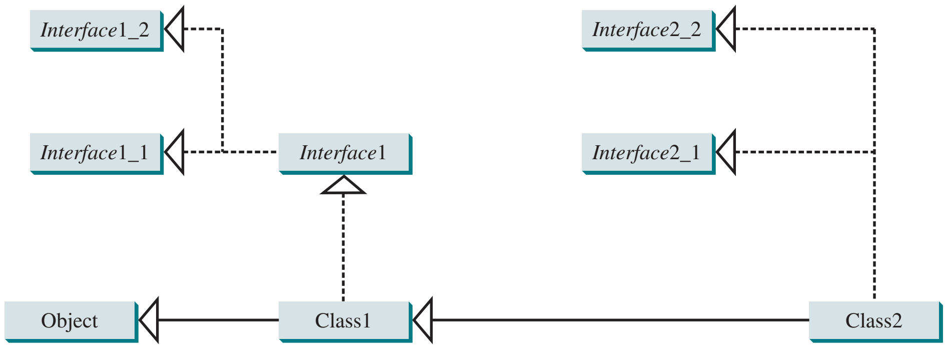

FIGURE 13.7 `Class1` implements `Interface1`; `Interface1` extends `Interface1_1` and `Interface1_2`. `Class2` extends `Class1` and implements `Interface2_1` and `Interface2_2`.

naming convention

Note

Class names are nouns. Interface names may be adjectives or nouns.

is-a relationship
is-kind-of relationship

Design Guide

Abstract classes and interfaces can both be used to specify common behavior of objects. How do you decide whether to use an interface or a class? In general, a *strong is-a relationship* that clearly describes a parent–child relationship should be modeled using classes. For example, Gregorian calendar is a calendar, so the relationship between the class `java.util.GregorianCalendar` and `java.util.Calendar` is modeled using class inheritance. A *weak is-a relationship*, also known as an *is-kind-of relationship*, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the `String` class implements the `Comparable` interface.

interface preferred

In general, interfaces are preferred over abstract classes because an interface can define a common supertype for unrelated classes. Interfaces are more flexible than classes. Consider the `Animal` class. Suppose the `howToEat` method is defined in the `Animal` class as follows:

`Animal` class

```
abstract class Animal {
  public abstract String howToEat();
}
```

Two subclasses of `Animal` are defined as follows:

`Chicken` class

```
class Chicken extends Animal {
  @Override
  public String howToEat() {
    return "Fry it";
  }
}
```

```
class Duck extends Animal {
  @Override
  public String howToEat() {
    return "Roast it";
  }
}
```

Duck class

Given this inheritance hierarchy, polymorphism enables you to hold a reference to a `Chicken` object or a `Duck` object in a variable of type `Animal`, as in the following code:

```
public static void main(String[] args) {
  Animal animal = new Chicken();
  eat(animal);

  animal = new Duck();
  eat(animal);
}

public static void eat(Animal animal) {
  System.out.println(animal.howToEat());
}
```

The JVM dynamically decides which `howToEat` method to invoke based on the actual object that invokes the method.

You can define a subclass of `Animal`. However, there is a restriction: the subclass must be for another animal (e.g., `Turkey`). Another issue arises: if an animal (e.g., `Tiger`) is not edible, it will not be appropriate to extend the `Animal` class.

Interfaces don't have these problems. Interfaces give you more flexibility than classes because you don't have to make everything fit into one type of class. You may define the `howToEat()` method in an interface, and let it serve as a common supertype for other classes. For example, see the following code:

```
public class DesignDemo {
  public static void main(String[] args) {
    Edible stuff = new Chicken();
    eat(stuff);

    stuff = new Duck();
    eat(stuff);

    stuff = new Broccoli();
    eat(stuff);
  }

  public static void eat(Edible stuff) {
    System.out.println(stuff.howToEat()):
  }
}
```

```
interface Edible {
  public String howToEat();
}
```

Edible interface

```
class Chicken implements Edible {
  @Override
  public String howToEat() {
    return "Fry it";
  }
}
```

Chicken class

Duck class

```
class Duck implements Edible {
  @Override
  public String howToEat() {
    return "Roast it";
  }
}
```

Broccoli class

```
class Broccoli implements Edible {
  @Override
  public String howToEat() {
    return "Stir-fry it";
  }
}
```

To define a class that represents edible objects, simply let the class implement the **Edible** interface. The class is now a subtype of the **Edible** type, and any **Edible** object can be passed to invoke the **howToEat** method.

Check Point

13.8.1 Give an example to show why interfaces are preferred over abstract classes.

13.8.2 Define the terms abstract classes and interfaces. What are the similarities and differences between abstract classes and interfaces?

13.8.3 True or false?

a. An interface is compiled into a separate bytecode file.
b. An interface can have static methods.
c. An interface can extend one or more interfaces.
d. An interface can extend an abstract class.
e. An interface can have default methods.

13.9 Case Study: The Rational Class

Key Point

This section shows how to design the **Rational** *class for representing and processing rational numbers.*

A rational number has a numerator and a denominator in the form **a/b**, where **a** is the numerator and **b** the denominator. For example, **1/3**, **3/4**, and **10/4** are rational numbers.

A rational number cannot have a denominator of **0**, but a numerator of **0** is fine. Every integer **i** is equivalent to a rational number **i/1**. Rational numbers are used in exact computations involving fractions—for example, **1/3 = 0.33333**. . . . This number cannot be precisely represented in floating-point format using either the data type **double** or **float**. To obtain the exact result, we must use rational numbers.

Java provides data types for integers and floating-point numbers, but not for rational numbers. This section shows how to design a class to represent rational numbers.

Since rational numbers share many common features with integers and floating-point numbers, and **Number** is the root class for numeric wrapper classes, it is appropriate to define **Rational** as a subclass of **Number**. Since rational numbers are comparable, the **Rational** class should also implement the **Comparable** interface. Figure 13.8 illustrates the **Rational** class and its relationship to the **Number** class and the **Comparable** interface.

A rational number consists of a numerator and a denominator. There are many equivalent rational numbers—for example, **1/3 = 2/6 = 3/9 = 4/12**. The numerator and the denominator of **1/3** have no common divisor except **1**, so **1/3** is said to be in *lowest terms*.

To reduce a rational number to its lowest terms, you need to find the greatest common divisor (GCD) of the absolute values of its numerator and denominator, then divide both the numerator and denominator by this value. You can use the method for computing the GCD of

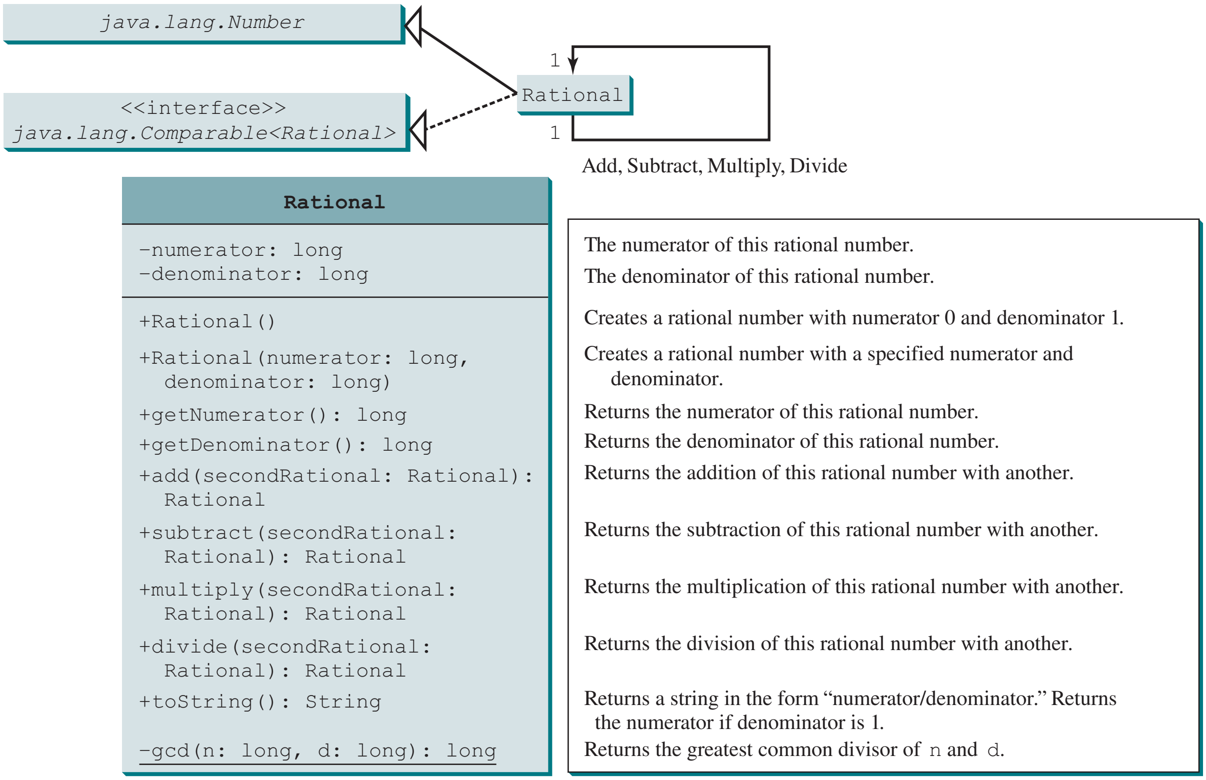

FIGURE 13.8 The properties, constructors, and methods of the `Rational` class are illustrated in UML.

two integers **n** and **d**, as suggested in Listing 5.9, GreatestCommonDivisor.java. The numerator and denominator in a `Rational` object are reduced to their lowest terms.

As usual, let us first write a test program to create two `Rational` objects and test its methods. Listing 13.12 is a test program.

LISTING 13.12 TestRationalClass.java

```java
public class TestRationalClass {
  /** Main method */
  public static void main(String[] args) {
    // Create and initialize two rational numbers r1 and r2
    Rational r1 = new Rational(4, 2);
    Rational r2 = new Rational(2, 3);

    // Display results
    System.out.println(r1 + " + " + r2 + " = " + r1.add(r2));
    System.out.println(r1 + " - " + r2 + " = " + r1.subtract(r2));
    System.out.println(r1 + " * " + r2 + " = " + r1.multiply(r2));
    System.out.println(r1 + " / " + r2 + " = " + r1.divide(r2));
    System.out.println(r2 + " is " + r2.doubleValue());
  }
}
```

create a Rational
create a Rational
add

```
2 + 2/3 = 8/3
2 - 2/3 = 4/3
2 * 2/3 = 4/3
2 / 2/3 = 3
2/3 is 0.6666666666666666
```

The `main` method creates two rational numbers, `r1` and `r2` (lines 5 and 6), and displays the results of `r1 + r2`, `r1 – r2`, `r1 x r2`, and `r1 / r2` (lines 9–12). To perform `r1 + r2`, invoke `r1.add(r2)` to return a new `Rational` object. Similarly, invoke `r1.subtract(r2)` for `r1 – r2`, `r1.multiply(r2)` for `r1 x r2`, and `r1.divide(r2)` for `r1 / r2`.

The `doubleValue()` method displays the double value of `r2` (line 13). The `doubleValue()` method is defined in `java.lang.Number` and overridden in `Rational`.

Note when a string is concatenated with an object using the plus sign (`+`), the object's string representation from the `toString()` method is used to concatenate with the string. Thus, `r1 + " + " + r2 + " = " + r1.add(r2)` is equivalent to `r1.toString() + " + " + r2.toString() + " = " + r1.add(r2).toString()`.

The `Rational` class is implemented in Listing 13.13.

LISTING 13.13 Rational.java

```java
public class Rational extends Number implements Comparable<Rational> {
  // Data fields for numerator and denominator
  private long numerator = 0;
  private long denominator = 1;

  /** Construct a rational with default properties */
  public Rational() {
    this(0, 1);
  }

  /** Construct a rational with specified numerator and denominator */
  public Rational(long numerator, long denominator) {
    long gcd = gcd(numerator, denominator);
    this.numerator = (denominator > 0 ? 1 : -1) * numerator / gcd;
    this.denominator = Math.abs(denominator) / gcd;
  }

  /** Find GCD of two numbers */
  private static long gcd(long n, long d) {
    long n1 = Math.abs(n);
    long n2 = Math.abs(d);
    int gcd = 1;

    for (int k = 1; k <= n1 && k <= n2; k++) {
      if (n1 % k == 0 && n2 % k == 0)
        gcd = k;
    }

    return gcd;
  }

  /** Return numerator */
  public long getNumerator() {
    return numerator;
  }

  /** Return denominator */
  public long getDenominator() {
    return denominator;
  }

  /** Add a rational number to this rational */
  public Rational add(Rational secondRational) {
    long n = numerator * secondRational.getDenominator() +
      denominator * secondRational.getNumerator();
```

$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$

```java
    long d = denominator * secondRational.getDenominator();
    return new Rational(n, d);
  }

  /** Subtract a rational number from this rational */
  public Rational subtract(Rational secondRational) {
    long n = numerator * secondRational.getDenominator()
      - denominator * secondRational.getNumerator();
    long d = denominator * secondRational.getDenominator();
    return new Rational(n, d);
  }

  /** Multiply a rational number by this rational */
  public Rational multiply(Rational secondRational) {
    long n = numerator * secondRational.getNumerator();
    long d = denominator * secondRational.getDenominator();
    return new Rational(n, d);
  }

  /** Divide a rational number by this rational */
  public Rational divide(Rational secondRational) {
    long n = numerator * secondRational.getDenominator();
    long d = denominator * secondRational.numerator;
    return new Rational(n, d);
  }

  @Override
  public String toString() {
    if (denominator == 1)
      return numerator + "";
    else
      return numerator + "/" + denominator;
  }

  @Override // Override the equals method in the Object class
  public boolean equals(Object other) {
    if ((this.subtract((Rational)(other))).getNumerator() == 0)
      return true;
    else
      return false;
  }

  @Override // Implement the abstract intValue method in Number
  public int intValue() {
    return (int)doubleValue();
  }

  @Override // Implement the abstract floatValue method in Number
  public float floatValue() {
    return (float)doubleValue();
  }

  @Override // Implement the doubleValue method in Number
  public double doubleValue() {
    return numerator *  1.0 / denominator;
  }

  @Override // Implement the abstract longValue method in Number
  public long longValue() {
    return (long)doubleValue();
```

$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$

$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$

$\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$

```
106     }
107
108     @Override // Implement the compareTo method in Comparable
109     public int compareTo(Rational o) {
110       if (this.subtract(o).getNumerator() > 0)
111         return 1;
112       else if (this.subtract(o).getNumerator() < 0)
113         return -1;
114       else
115         return 0;
116     }
117   }
```

The rational number is encapsulated in a **Rational** object. Internally, a rational number is represented in its lowest terms (line 13) and the numerator determines its sign (line 14). The denominator is always positive (line 15).

The **gcd** method (lines 19–30 in the **Rational** class) is private; it is not intended for use by clients. The **gcd** method is only for internal use by the **Rational** class. The **gcd** method is also static, since it is not dependent on any particular **Rational** object.

The **abs(x)** method (lines 20 and 21 in the **Rational** class) is defined in the **Math** class and returns the absolute value of **x**.

Two **Rational** objects can interact with each other to perform add, subtract, multiply, and divide operations. These methods return a new **Rational** object (lines 43–70). The math formula for performing these operations are as follows:

$$a/b + c/d = (ad + bc)/(bd) \text{ (e.g., } 2/3 + 3/4 = (2*4 + 3*3)/(3*4) = 17/12)$$

$$a/b - c/d = (ad - bc)/(bd) \text{ (e.g., } 2/3 - 3/4 = (2*4 - 3*3)/(3*4) = -1/12)$$

$$a/b * c/d = (ac)/(bd) \text{ (e.g., } 2/3*3/4 = (2*3)/(3*4) = 6/12 = 1/2)$$

$$(a/b) / (c/d) = (ad)/(bc) \text{ (e.g., } (2/3) / (3/4) = (2*4)/(3*3) = 8/9)$$

The methods **toString** and **equals** in the **Object** class are overridden in the **Rational** class (lines 72–86). The **toString()** method returns a string representation of a **Rational** object in the form **numerator/denominator**, or simply **numerator** if **denominator** is **1**. The **equals(Object other)** method returns true if this rational number is equal to the other rational number.

The abstract methods **intValue**, **longValue**, **floatValue**, and **doubleValue** in the **Number** class are implemented in the **Rational** class (lines 88–106). These methods return the **int**, **long**, **float**, and **double** value for this rational number.

The **compareTo(Rational other)** method in the **Comparable** interface is implemented in the **Rational** class (lines 108–116) to compare this rational number to the other rational number.

Note

immutable

The getter methods for the properties **numerator** and **denominator** are provided in the **Rational** class, but the setter methods are not provided, so, once a **Rational** object is created, its contents cannot be changed. The **Rational** class is immutable. The **String** class and the wrapper classes for primitive-type values are also immutable.

Note

The numerator and denominator are represented using two variables. It is possible to use an array of two integers to represent the numerator and denominator (see Programming Exercise 13.14). The signatures of the public methods in the **Rational** class are not changed, although the internal representation of a rational number is changed. This is a good example to illustrate the idea that the data fields of a class should be kept private so as to encapsulate the implementation of the class from the use of the class.

encapsulation

overflow

The **Rational** class has serious limitations and can easily overflow. For example, the following code will display an incorrect result, because the denominator is too large:

```
public class Test {
  public static void main(String[] args) {
    Rational r1 = new Rational(1, 123456789);
    Rational r2 = new Rational(1, 123456789);
    Rational r3 = new Rational(1, 123456789);
    System.out.println("r1 * r2 * r3 is " +
      r1.multiply(r2.multiply(r3)));
  }
}
```

```
r1 * r2 * r3 is -1/2204193661661244627
```

To fix it, you can implement the **Rational** class using the **BigInteger** for numerator and denominator (see Programming Exercise 13.15).

Check Point

13.9.1 Show the output of the following code:

```
Rational r1 = new Rational(-2, 6);
System.out.println(r1.getNumerator());
System.out.println(r1.getDenominator());
System.out.println(r1.intValue());
System.out.println(r1.doubleValue());
```

13.9.2 Why is the following code wrong?

```
Rational r1 = new Rational(-2, 6);
Object r2 = new Rational(1, 45);
System.out.println(r2.compareTo(r1));
```

13.9.3 Why is the following code wrong?

```
Object r1 = new Rational(-2, 6);
Rational r2 = new Rational(1, 45);
System.out.println(r2.compareTo(r1));
```

13.9.4 Simplify the code in lines 82–85 in Listing 13.13, Rational.java, using one line of code without using the if statement. Simplify the code in lines 110–115 using a conditional operator.

13.9.5 Trace the program carefully and show the output of the following code:

```
Rational r1 = new Rational(1, 2);
Rational r2 = new Rational(1, -2);
System.out.println(r1.add(r2));
```

13.9.6 The preceding question shows a bug in the **toString** method. Revise the **toString()** method to fix the error.

13.10 Class-Design Guidelines

Class-design guidelines are helpful for designing sound classes.

Key Point

You have learned how to design classes from the preceding example and from many other examples in the previous chapters. This section summarizes some of the guidelines.

13.10.1 Cohesion

A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff are different entities.

coherent purpose

separate responsibilities

A single entity with many responsibilities can be broken into several classes to separate the responsibilities. The classes `String`, `StringBuilder`, and `StringBuffer` all deal with strings, for example, but have different responsibilities. The `String` class deals with immutable strings, the `StringBuilder` class is for creating mutable strings, and the `StringBuffer` class is similar to `StringBuilder`, except that `StringBuffer` contains synchronized methods for updating strings.

13.10.2 Consistency

naming conventions

Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods. A popular style is to place the data declaration before the constructor, and place constructors before methods.

naming consistency

Make the names consistent. It is not a good practice to choose different names for similar operations. For example, the `length()` method returns the size of a `String`, a `StringBuilder`, and a `StringBuffer`. It would be inconsistent if different names were used for this method in these classes.

no-arg constructor

In general, you should consistently provide a public no-arg constructor for constructing a default instance. If a class does not support a no-arg constructor, document the reason. If no constructors are defined explicitly, a public default no-arg constructor with an empty body is assumed.

If you want to prevent users from creating an object for a class, you can declare a private constructor in the class, as is the case for the `Math` class and the `GuessDate` class.

13.10.3 Encapsulation

encapsulate data fields

A class should use the `private` modifier to hide its data from direct access by clients. This makes the class easy to maintain.

Provide a getter method only if you want the data field to be readable and provide a setter method only if you want the data field to be updateable. For example, the `Rational` class provides a getter method for `numerator` and `denominator`, but no setter method, because a `Rational` object is immutable.

13.10.4 Clarity

easy to explain

Cohesion, consistency, and encapsulation are good guidelines for achieving design clarity. In addition, a class should have a clear contract that is easy to explain and easy to understand.

independent methods

Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on how or when the user can use it, design the properties in a way that lets the user set them in any order and with any combination of values, and design methods that function independently of their order of occurrence. For example, the `Loan` class contains the properties `loanAmount`, `numberOfYears`, and `annualInterestRate`. The values of these properties can be set in any order.

intuitive meaning

Methods should be defined intuitively without causing confusion. For example, the `substring(int beginIndex, int endIndex)` method in the `String` class is somewhat confusing. The method returns a substring from `beginIndex` to `endIndex – 1`, rather than to `endIndex`. It would be more intuitive to return a substring from `beginIndex` to `endIndex`.

independent properties

You should not declare a data field that can be derived from other data fields. For example, the following `Person` class has two data fields: `birthDate` and `age`. Since `age` can be derived from `birthDate`, `age` should not be declared as a data field.

```
public class Person {
  private java.util.Date birthDate;
  private int age;

  ...
}
```

13.10.5 Completeness

Classes are 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 properties and methods. For example, the `String` class contains more than 40 methods that are useful for a variety of applications.

13.10.6 Instance vs. Static

A variable or method that is dependent on a specific instance of the class must be an instance variable or method. A variable that is shared by all the instances of a class should be declared static. For example, the variable `numberOfObjects` in `Circle` in Listing 9.8 is shared by all the objects of the `Circle` class, and therefore is declared static. A method that is not dependent on a specific instance should be defined as a static method. For instance, the `getNumberOfObjects()` method in `Circle` is not tied to any specific instance and therefore is defined as a static method.

Always reference static variables and methods from a class name (rather than a reference variable) to improve readability and avoid errors.

Do not pass a parameter from a constructor to initialize a static data field. It is better to use a setter method to change the static data field. Thus, the following class in (a) is better replaced by (b):

```
public class SomeThing {
  private int t1;
  private static int t2;

  public SomeThing(int t1, int t2) {
    ...
  }
}
```

(a)

```
public class SomeThing {
  private int t1;
  private static int t2;

  public SomeThing(int t1) {
    ...
  }

  public static void setT2(int t2) {
    SomeThing.t2 = t2;
  }
}
```

(b)

Instance and static are integral parts of object-oriented programming. A data field or method is either instance or static. Do not mistakenly overlook static data fields or methods. It is a common design error to define an instance method that should have been static. For example, the `factorial(int n)` method for computing the factorial of `n` should be defined static because it is independent of any specific instance. common design error

A constructor is always instance because it is used to create a specific instance. A static variable or method can be invoked from an instance method, but an instance variable or method cannot be invoked from a static method.

13.10.7 Inheritance vs. Aggregation

The difference between inheritance and aggregation is the difference between an is-a and a has-a relationship. For example, an apple is a fruit; thus, you would use inheritance to model the relationship between the classes `Apple` and `Fruit`. A person has a name; thus, you would use aggregation to model the relationship between the classes `Person` and `Name`.

13.10.8 Interfaces vs. Abstract Classes

Both interfaces and abstract classes can be used to specify common behavior for objects. How do you decide whether to use an interface or a class? In general, a strong is-a relationship that clearly describes a parent–child relationship should be modeled using classes. For example, since an orange is a fruit, their relationship should be modeled using class inheritance. A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the `String` class implements the `Comparable` interface. A circle or a rectangle is a geometric object, so `Circle` can be designed as a subclass of `GeometricObject`. Circles are different and comparable based on their radii, so `Circle` can implement the `Comparable` interface.

Interfaces are more flexible than abstract classes because a subclass can extend only one superclass, but can implement any number of interfaces. However, interfaces cannot contain data fields. In Java 8, interfaces can contain default methods and static methods, which are very useful to simplify class design. We will give examples of this type of design in Chapter 20, Lists, Stacks, Queues, and Priority Queues.

13.10.1 Describe class-design guidelines.

Check Point

Key Terms

abstract class 522
abstract method 522
deep copy 542
interface 532
marker interface 540
shallow copy 542
subinterface 545

Chapter Summary

1. *Abstract classes* are like regular classes with data and methods, but you cannot create instances of abstract classes using the `new` operator.
2. An *abstract method* cannot be contained in a nonabstract class. If a subclass of an abstract superclass does not implement all the inherited abstract methods of the superclass, the subclass must be defined as abstract.
3. A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that doesn't contain any abstract methods.
4. A subclass can be abstract even if its superclass is concrete.
5. An *interface* is a class-like construct that contains only constants, abstract methods, default methods, and static methods. In many ways, an interface is similar to an abstract class, but an abstract class can contain data fields.
6. An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class.
7. The `java.lang.Comparable` interface defines the `compareTo` method. Many classes in the Java library implement `Comparable`.

8. The `java.lang.Cloneable` interface is a *marker interface*. An object of the class that implements the `Cloneable` interface is cloneable.

9. A class can extend only one superclass but can implement one or more interfaces.

10. An interface can extend one or more interfaces.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

MyProgrammingLab™

Sections 13.2 and 13.3

****13.1** (*Triangle class*) Design a new `Triangle` class that extends the abstract `GeometricObject` class. Draw the UML diagram for the classes `Triangle` and `GeometricObject` then implement the `Triangle` class. Write a test program that prompts the user to enter three sides of the triangle, a color, and a Boolean value to indicate whether the triangle is filled. The program should create a `Triangle` object with these sides, and set the `color` and `filled` properties using the input. The program should display the area, perimeter, color, and true or false to indicate whether it is filled or not.

***13.2** (*Average ArrayList*) Write the following method that averages an `ArrayList` of integers:

```
public static void average (ArrayList<Integer> list)
```

***13.3** (*Sort ArrayList*) Write the following method that sorts an `ArrayList` of numbers:

```
public static void sort (ArrayList<Number> list)
```

****13.4** (*Display calendars*) Rewrite the `PrintCalendar` class in Listing 6.12 to display a calendar for a specified month using the `Calendar` and `GregorianCalendar` classes. Your program receives the month and year from the command line. For example:

```
java Exercise13_04 5 2016
```

This displays the calendar shown in Figure 13.9.

```
Command Prompt
c:\exercise>java Exercise13_04 10 2019
         October 2019
-----------------------------
 Sun Mon Tue Wed Thu Fri Sat
               1   2   3   4   5
   6   7   8   9  10  11  12
  13  14  15  16  17  18  19
  20  21  22  23  24  25  26
  27  28  29  30  31

c:\exercise>
```

Figure 13.9 The program displays a calendar for May 2016.

You can also run the program without the year. In this case, the year is the current year. If you run the program without specifying a month and a year, the month is the current month.

Sections 13.4–13.8v

***13.5** (*Enable `GeometricObject` comparable*) Modify the `GeometricObject` class to implement the `Comparable` interface and define a static `max` method in the `GeometricObject` class for finding the larger of two `GeometricObject` objects. Draw the UML diagram and implement the new `GeometricObject` class. Write a test program that uses the `max` method to find the larger of two circles, the larger of two rectangles.

***13.6** (*The `ComparableCircle` class*) Define a class named `ComparableCircle` that extends `Circle` and implements `Comparable`. Draw the UML diagram and implement the `compareTo` method to compare the circles on the basis of area. Write a test class to find the larger of two instances of `ComparableCircle` objects, and the larger between a circle and a rectangle.

***13.7** (*The `Colorable` interface*) Design an interface named `Colorable` with a `void` method named `howToColor()`. Every class of a colorable object must implement the `Colorable` interface. Design a class named `Square` that extends `GeometricObject` and implements `Colorable`. Implement `howToColor` to display the message `Color all four sides`. The `Square` class contains a data field `side` with getter and setter methods, and a constructor for constructing a `Square` with a specified side. The `Square` class has a private double data field named `side` with its getter and setter methods. It has a no-arg constructor to create a `Square` with side 0, and another constructor that creates a `Square` with the specified side.

Draw a UML diagram that involves `Colorable`, `Square`, and `GeometricObject`. Write a test program that creates an array of five `GeometricObject`s. For each object in the array, display its area and invoke its `howToColor` method if it is colorable.

***13.8** (*Revise the `MyStack` class*) Rewrite the `MyStack` class in Listing 11.10 to perform a deep copy of the `list` field.

***13.9** (*Enable `Circle` comparable*) Rewrite the `Circle` class in Listing 13.2 to extend `GeometricObject` and implement the `Comparable` interface. Override the `equals` method in the `Object` class. Two `Circle` objects are equal if their radii are the same. Draw the UML diagram that involves `Circle`, `GeometricObject`, and `Comparable`.

***13.10** (*Enable `Rectangle` comparable*) Rewrite the `Rectangle` class in Listing 13.3 to extend `GeometricObject` and implement the `Comparable` interface. Override the `equals` method in the `Object` class. Two `Rectangle` objects are equal if their areas are the same. Draw the UML diagram that involves `Rectangle`, `GeometricObject`, and `Comparable`.

***13.11** (*The `Octagon` class*) Write a class named `Octagon` that extends `GeometricObject` and implements the `Comparable` and `Cloneable` interfaces. Assume all eight sides of the octagon are of equal length. The area can be computed using the following formula:

$$area = \left(2 + 4/\sqrt{2}\right) * side * side$$

The `Octagon` class has a private double data field named side with its getter and setter methods. The class has a no-arg constructor that creates an `Octagon` with side 0, and a constructor to create an `Octagon` with a specified side.

Draw the UML diagram that involves **Octagon**, **GeometricObject**, **Comparable**, and **Cloneable**. Write a test program that creates an **Octagon** object with side value **5** and displays its area and perimeter. Create a new object using the **clone** method, and compare the two objects using the **compareTo** method.

***13.12** (*Sum the areas of geometric objects*) Write a method that sums the areas of all the geometric objects in an array. The method signature is

```
public static double sumArea(GeometricObject[] a)
```

Write a test program that creates an array of four objects (two circles and two rectangles) and computes their total area using the **sumArea** method.

***13.13** (*Enable the **Course** class cloneable*) Rewrite the **Course** class in Listing 10.6 to add a **clone** method to perform a deep copy on the **students** field.

Section 13.9

***13.14** (*Demonstrate the benefits of encapsulation*) Rewrite the **Rational** class in Listing 13.13 using a new internal representation for the numerator and denominator. Create an array of two integers as follows:

```
private long[] r = new long[2];
```

Use **r[0]** to represent the numerator and **r[1]** to represent the denominator. The signatures of the methods in the **Rational** class are not changed, so a client application that uses the previous **Rational** class can continue to use this new **Rational** class without being recompiled.

***13.15** (*Use **BigInteger** for the **Rational** class*) Redesign and implement the **Rational** class in Listing 13.13 using **BigInteger** for the numerator and denominator. Write a test program that prompts the user to enter two rational numbers and display the results as shown in the following sample run:

```
Enter the first rational number: 3 454 ↵Enter
Enter the second rational number: 7 2389 ↵Enter
3/454 + 7/2389 = 10345/1084606
3/454 - 7/2389 = 3989/1084606
3/454 * 7/2389 = 21/1084606
3/454 / 7/2389 = 7167/3178
7/2389 is 0.0029300962745918793
```

***13.16** (*Create a rational-number calculator*) Write a program similar to Listing 7.9, Calculator.java. Instead of using integers, use rationals, as shown in Figure 13.10. You will need to use the **split** method in the **String** class, introduced in Section 10.10.3, Replacing and Splitting Strings, to retrieve the numerator string and denominator string, and convert strings into integers using the **Integer.parseInt** method.

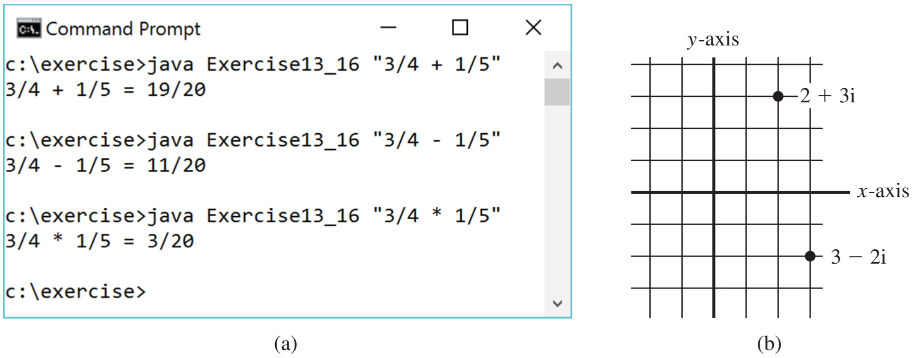

FIGURE 13.10 (a) The program takes a string argument that consists of operand1, operator, and operand2 from the command line and displays the expression and the result of the arithmetic operation. (b) A complex number can be interpreted as a point in a plane.

***13.17** (*Math: The* **`Complex`** *class*) A complex number is a number in the form $a + bi$, where a and b are real numbers and i is $\sqrt{-1}$. The numbers **`a`** and **`b`** are known as the real part and imaginary part of the complex number, respectively. You can perform addition, subtraction, multiplication, and division for complex numbers using the following formulas:

$$a + bi + c + di = (a + c) + (b + d)i$$

$$a + bi - (c + di) = (a - c) + (b - d)i$$

$$(a + bi) * (c + di) = (ac - bd) + (bc + ad)i$$

$$(a + bi)/(c + di) = (ac + bd)/(c^2 + d^2) + (bc - ad)i/(c^2 + d^2)$$

You can also obtain the absolute value for a complex number using the following formula:

$$|a + bi| = \sqrt{a^2 + b^2}$$

(A complex number can be interpreted as a point on a plane by identifying the (a,b) values as the coordinates of the point. The absolute value of the complex number corresponds to the distance of the point to the origin, as shown in Figure 13.10.)

Design a class named **`Complex`** for representing complex numbers and the methods **`add`**, **`subtract`**, **`multiply`**, **`divide`**, and **`abs`** for performing complex-number operations, and override **`toString`** method for returning a string representation for a complex number. The **`toString`** method returns **`(a + bi)`** as a string. If **`b`** is **`0`**, it simply returns **`a`**. Your **`Complex`** class should also implement **`Cloneable`** and **`Comparable`**. Compare two complex numbers using their absolute values.

Provide three constructors **`Complex(a, b)`**, **`Complex(a)`**, and **`Complex()`**. **`Complex()`** creates a **`Complex`** object for number **`0`**, and **`Complex(a)`** creates a **`Complex`** object with **`0`** for **`b`**. Also provide the **`getRealPart()`** and **`getImaginaryPart()`** methods for returning the real part and the imaginary part of the complex number, respectively.

Draw the UML class diagram and implement the class. Use the code at https://liveexample.pearsoncmg.com/test/Exercise13_17.txt to test your implementation. Here is a sample run:

```
Enter the first complex number: 3.5 5.5 ↵Enter
Enter the second complex number: -3.5 1 ↵Enter
(3.5 + 5.5i) + (-3.5 + 1.0i) = 0.0 + 6.5i
(3.5 + 5.5i) - (-3.5 + 1.0i) = 7.0 + 4.5i
(3.5 + 5.5i) * (-3.5 + 1.0i) = -17.75 + -15.75i
(3.5 + 5.5i) / (-3.5 + 1.0i) = -0.5094 + -1.7i
|(3.5 + 5.5i)| = 6.519202405202649
false
3.5
5.5
[-3.5 + 1.0i, 4.0 + -0.5i, 3.5 + 5.5i, 3.5 + 5.5i]
```

13.18 (*Use the* **`Rational`** *class*) Write a program that computes the following summation series using the **`Rational`** class:

$$\frac{1}{2} + \frac{3}{4} + \frac{5}{6} + \cdots + \frac{97}{98} + \frac{99}{100}$$

You will discover that the output is incorrect because of integer overflow (too large). To fix this problem, see Programming Exercise 13.15.

13.19 (*Convert decimals to fractions*) Write a program that prompts the user to enter a decimal number and displays the number in a fraction. (*Hint*: read the decimal number as a string, extract the integer part and fractional part from the string, and use the **`Rational`** class in Listing 13.13 to obtain a rational number for the decimal number.) Here are some sample runs:

```
Enter a decimal number: 3.25 ↵Enter
The fraction number is 13/4
```

```
Enter a decimal number: -0.45452 ↵Enter
The fraction number is -11363/25000
```

13.20 (*Algebra: solve quadratic equations*) Rewrite Programming Exercise 3.1 to obtain imaginary roots if the determinant is less than 0 using the **`Complex`** class in Programming Exercise 13.17. Here are some sample runs:

```
Enter a, b, c: 1 3 1 ↵Enter
The roots are -0.381966 and -2.61803
```

```
Enter a, b, c: 1 2 1 ↵Enter
The root is -1
```

```
Enter a, b, c: 1 2 3 ↵Enter
The roots are -1.0 + 1.4142i and -1.0 + -1.4142i
```

13.21 (*Algebra: vertex form equations*) The equation of a parabola can be expressed in either standard form ($y = ax^2 + bx + c$) or vertex form ($y = a(x - h)^2 + k$). Write a program that prompts the user to enter a, b, and c as integers in standard form and displays $h\left(= \frac{-b}{2a}\right)$ and $k\left(= \frac{4ac - b^2}{4a}\right)$ in the vertex form. Display h and k as rational numbers. Here are some sample runs:

```
Enter a, b, c: 1 3 1 ↵Enter
h is -3/2 k is -5/4
```

```
Enter a, b, c: 2 3 4 ↵Enter
h is -3/4 k is 23/8
```