Chapter 1

Object-Oriented Programming and Class Hierarchies

Chapter Objectives

- To learn about interfaces and their role in Java
- To understand inheritance and how it facilitates code reuse
- To understand how Java determines which method to execute when there are multiple methods with the same name in a class hierarchy
- To become familiar with the Exception hierarchy and the difference between checked and unchecked exceptions
- To learn how to define and use abstract classes as base classes in a hierarchy
- To learn the role of abstract data types and how to specify them using interfaces
- To study class `Object` and its methods and to learn how to override them
- To become familiar with a class hierarchy for shapes
- To understand how to create packages and to learn more about visibility

This chapter describes important features of Java that support Object-Oriented Programming (OOP). Object-oriented languages allow you to build and exploit hierarchies of classes in order to write code that may be more easily reused in new applications. You will learn how to extend an existing Java class to define a new class that inherits all the attributes of the original, as well as having additional attributes of its own. Because there may be many versions of the same method in a class hierarchy, we show how polymorphism enables Java to determine which version to execute at any given time.

We introduce interfaces and abstract classes and describe their relationship with each other and with actual classes. We introduce the abstract class `Number`. We also discuss class `Object`, which all classes extend, and we describe several of its methods that may be used in classes you create.

As an example of a class hierarchy and OOP, we describe the `Exception` class hierarchy and explain that the Java Virtual Machine (JVM) creates an `Exception` object whenever an error occurs during program execution. Finally, you will learn how to create packages in Java and about the different kinds of visibility for instance variables (data fields) and methods.

Inheritance and Class Hierarchies

1.1 ADTs, Interfaces, and the Java API

In earlier programming courses, you learned how to write individual classes consisting of attributes and methods (operations). You also learned how to use existing classes (e.g., `String` and `Scanner`) to facilitate your programming. These classes are part of the Java Application Programming Interface (API).

One of our goals is to write code that can be reused in many different applications. One way to make code reusable is to encapsulate the data elements together with the methods that operate on that data. A new program can then use the methods to manipulate an object's data without being concerned about details of the data representation or the method implementations. The encapsulated data together with its methods is called an abstract data type (ADT).

FIGURE 1.1
Diagram of an ADT

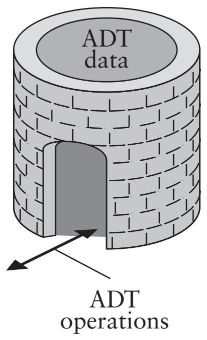


Figure 1.1 shows a diagram of an ADT. The data values stored in the ADT are hidden inside the circular wall. The bricks around this wall are used to indicate that these data values cannot be accessed except by going through the ADT's methods.

A class provides one way to implement an ADT in Java. If the data fields are private, they can be accessed only through public methods. Therefore, the methods control access to the data and determine the manner in which the data is manipulated.

Another goal of this text is to show you how to write and use ADTs in programming. As you progress through this book, you will create a large collection of ADT implementations (classes) in your own program library. You will also learn about ADTs that are available for you to use through the Java API.

Our principal focus will be on ADTs that are used for structuring data to enable you to more easily and efficiently store, organize, and process information. These ADTs are often called *data structures*. We introduce the Java Collections Framework (part of the Java API), which provides implementation of these common data structures, in Chapter 2 and study it throughout the text. Using the classes that are in the Java Collections Framework will make it much easier for you to design and implement new application programs.

Interfaces

A Java interface is a way to specify or describe an ADT to an applications programmer. An interface is like a contract that tells the applications programmer precisely what methods are available and describes the operations they perform. It also tells the applications programmer what arguments, if any, must be passed to each method and what result the method will return. Of course, in order to make use of these methods, someone else must have written a class that *implements the interface* by providing the code for these methods.

The interface tells the coder precisely what methods must be written, but it does not provide a detailed algorithm or prescription for how to write them. The coder must "program to the interface," which means he or she must develop the methods described in the interface without variation. If each coder does this job well, that ensures that other programmers can use the completed class exactly as it is written, without needing to know the details of how it was coded.

There may be more than one way to implement the methods; hence, several classes may implement the interface, but each must satisfy the contract. One class may be more efficient than the others at performing certain kinds of operations (e.g., retrieving information from a database), so that class will be used if retrieval operations are more likely in a particular application. The important point is that the particular implementation that is used will not affect other classes that interact with it because every implementation satisfies the contract.

Besides providing the complete definition (implementation) of all methods declared in the interface, each implementer of an interface may declare data fields and define other methods not in the interface, including constructors. An interface cannot contain constructors because it cannot be instantiated—that is, one cannot create objects, or instances, of it. However, it can be represented by instances of classes that implement it.

EXAMPLE 1.1 An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must support the following operations.

1. Verify a user's Personal Identification Number (PIN).
2. Allow the user to choose a particular account.
3. Withdraw a specified amount of money.
4. Display the result of an operation.
5. Display an account balance.

A class that implements an ATM must provide a method for each operation. We can write this requirement as the interface `ATM` and save it in file `ATM.java`, shown in Listing 1.1. The keyword `interface` on the header line indicates that an interface is being declared. If you are unfamiliar with the documentation style shown in this listing, read about Java documentation at the end of Section A.7 in Appendix A.

LISTING 1.1
Interface `ATM.java`

```
/** The interface for an ATM. */
public interface ATM {

    /** Verifies a user's PIN.
        @param pin The user's PIN
        @return Whether or not the User's PIN is verified
     */
    boolean verifyPIN(String pin);

    /** Allows the user to select an account.
        @return a String representing the account selected
     */
    String selectAccount();

    /** Withdraws a specified amount of money
        @param account The account from which the money comes
        @param amount The amount of money withdrawn
        @return Whether or not the operation is successful
     */
    boolean withdraw(String account, double amount);

    /** Displays the result of an operation
        @param account The account for the operation
        @param amount The amount of money
        @param success Whether or not the operation was successful
     */
    void display(String account, double amount, boolean success);

    /** Displays the result of a PIN verification
        @param pin The user's pin
        @param success Whether or not the PIN was valid
     */
    void display(String pin, boolean success);

    /** Displays an account balance
        @param account The account selected
     */
    void showBalance(String account);
}
```

The interface definition shows the heading only for several methods. Because only the headings are shown, they are considered *abstract methods*. Each actual method with its body must be defined in a class that implements the interface. Therefore, a class that implements this interface must provide a `void` method called `verifyPIN` with an argument of type `String`. There are also two display methods with different signatures. The first is used to display the result of a withdrawal, and the second is used to display the result of a PIN verification. The keywords `public abstract` are optional (and usually omitted) in an interface because all interface methods are public abstract by default.

SYNTAX Interface Definition

FORM:

```
public interface interfaceName {
    abstract method declarations
    constant declarations
}
```

EXAMPLE:

```
public interface Payable {
    public abstract double calcSalary();
    public abstract boolean salaried();
    public static final double DEDUCTIONS = 25.5;
}
```

MEANING:

Interface *interfaceName* is defined. The interface body provides headings for abstract methods and constant declarations. Each abstract method must be defined in a class that implements the interface. Constants defined in the interface (e.g., `DEDUCTIONS`) are accessible in classes that implement the interface or in the same way as static fields and methods in classes (see Section A.4).

NOTES:

The keywords `public` and `abstract` are implicit in each abstract method declaration, and the keywords `public static final` are implicit in each constant declaration. We show them in the example here, but we will omit them from now on.

Java 8 also allows for static and default methods in interfaces. They are used to add features to existing classes and interfaces while minimizing the impact on existing programs. We will discuss default and static methods when describing where they are used in the API.

The `implements` Clause

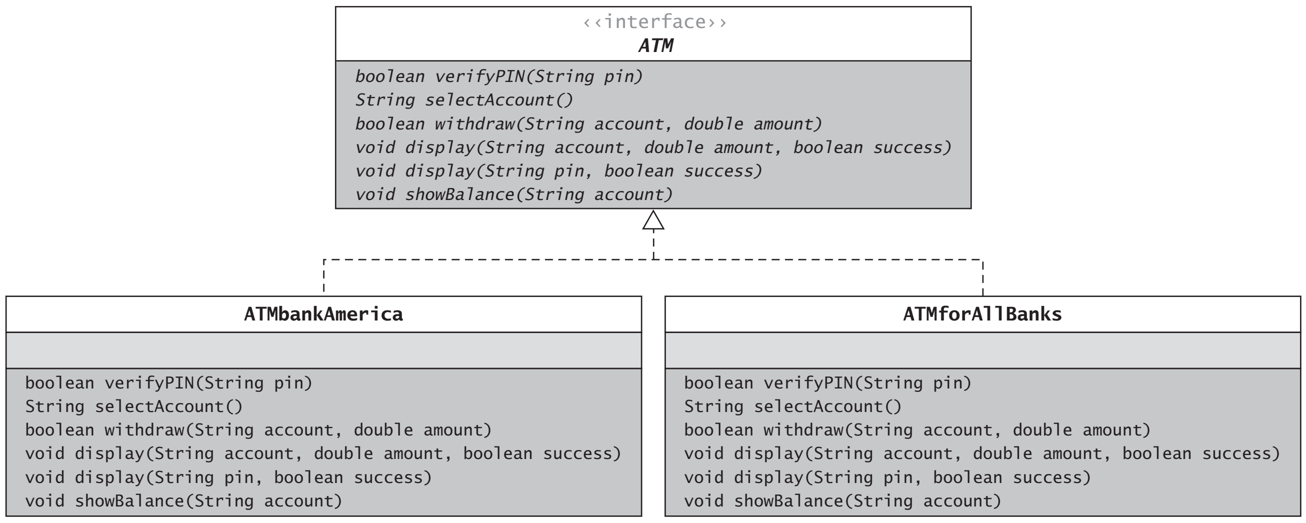
The class headings for two classes that implement interface ATM are

```
public class ATMbankAmerica implements ATM
public class ATMforAllBanks implements ATM
```

Each class heading ends with the clause `implements ATM`. When compiling these classes, the Java compiler will verify that they define the required methods in the way specified by the interface. If a class implements more than one interface, list them all after `implements`, with commas as separators.

Figure 1.2 is a UML (Unified Modeling Language) diagram that shows the ATM interface and these two implementing classes. Note that a dashed line from the class to the interface is used to indicate that the class implements the interface. We will use UML diagrams throughout this text to show relationships between classes and interfaces. Appendix B provides detailed coverage of UML diagrams.

FIGURE 1.2
UML Diagram Showing the ATM Interface and Its Implementing Classes



‹‹interface››
ATM
boolean verifyPIN(String pin)
String selectAccount()
boolean withdraw(String account, double amount)
void display(String account, double amount, boolean success)
void display(String pin, boolean success)
void showBalance(String account)
ATMbankAmerica
boolean verifyPIN(String pin)
String selectAccount()
boolean withdraw(String account, double amount)
void display(String account, double amount, boolean success)
void display(String pin, boolean success)
void showBalance(String account)
ATMforAllBanks
boolean verifyPIN(String pin)
String selectAccount()
boolean withdraw(String account, double amount)
void display(String account, double amount, boolean success)
void display(String pin, boolean success)
void showBalance(String account)

PITFALL

Not Properly Defining a Method to Be Implemented

If you neglect to define method `verifyPIN` in class `ATMforAllBanks` or if you use a different method signature, you will get the following syntax error:

```
class ATMforAllBanks should be declared abstract; it does not define method
verifyPIN(String) in interface ATM.
```

The above error indicates that the method `verifyPin` was not properly defined. Because it contains an abstract method that is not defined, Java incorrectly believes that `ATM` should be declared to be an abstract class. If you use a result type other than `boolean`, you will also get a syntax error.

PITFALL

Instantiating an Interface

An interface is not a class, so you cannot instantiate an interface. The statement

```
  ATM anATM = new ATM();  // invalid statement
```

will cause the following syntax error:

```
  interface ATM is abstract; cannot be instantiated.
```

Declaring a Variable of an Interface Type

In the previous programming pitfall, we mentioned that you cannot instantiate an interface. However, you may want to declare a variable that has an interface type and use it to reference an actual object. This is permitted if the variable references an object of a class type that implements the interface. After the following statements execute, variable `ATM1` references an `ATMbankAmerica` object, and variable `ATM2` references an `ATMforAllBanks` object, but both `ATM1` and `ATM2` are type `ATM`.

```
ATM ATM1 = new ATMbankAmerica();  // valid statement
ATM ATM2 = new ATMforAllBanks();  // valid statement
```

EXERCISES FOR SECTION 1.1

SELF-CHECK

1. What are the two parts of an ADT? Which part is accessible to a user and which is not? Explain the relationships between an ADT and a class, between an ADT and an interface, and between an interface and classes that implement the interface.
2. Correct each of the following statements that is incorrect, assuming that class `PDGUI` and class `PDConsoleUI` implement interface `PDUserInterface`.
   a. `PDGUI p1 = new PDConsoleUI();`
   b. `PDGUI p2 = new PDUserInterface();`

c. `PDUserInterface p3 = new PDUserInterface();`

d. `PDUserInterface p4 = new PDConsoleUI();`

e. `PDGUI p5 = new PDUserInterface();`
   `PDUserInterface p6 = p5;`

f. `PDUserInterface p7;`
   `p7 = new PDConsoleUI();`

3. Explain how an interface is like a contract.
4. What are two different uses of the term *interface* in programming?

PROGRAMMING

1. Define an interface named `Resizable` with just one abstract method, `resize`, that is a `void` method with no parameter.
2. Write a Javadoc comment for the following method of a class `Person`. Assume that class `Person` has two `String` data fields `familyName` and `givenName` with the obvious meanings. Provide preconditions and postconditions if needed.

```
public int compareTo(Person per) {
    if (familyName.compareTo(per.familyName) == 0)
        return givenName.compareTo(per.givenName);
    else
        return familyName.compareTo(per.familyName);
}
```

3. Write a Javadoc comment for the following method of class `Person`. Provide preconditions and postconditions if needed.

```
public void changeFamilyName(boolean justMarried, String newFamily) {
    if (justMarried)
        familyName = newFamily;
}
```

4. Write method `verifyPIN` for class `ATMbankAmerica` assuming this class has a data field `pin` (type `String`).

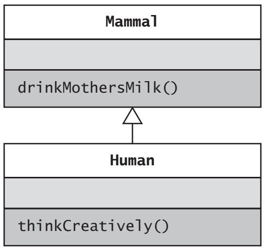
1.2 Introduction to Object-Oriented Programming (OOP)

In this course, you will learn to use features of Java that facilitate the practice of OOP. A major reason for the popularity of OOP is that it enables programmers to reuse previously written code saved as classes, reducing the time required to code new applications. Because previously written code has already been tested and debugged, the new applications should also be more reliable and therefore easier to test and debug.

However, OOP provides additional capabilities beyond the reuse of existing classes. If an application needs a new class that is similar to an existing class but not exactly the same, the programmer can create it by extending, or inheriting from, the existing class. The new class (called the subclass) can have additional data fields and methods for increased functionality. Its objects also inherit the data fields and methods of the original class (called the superclass).

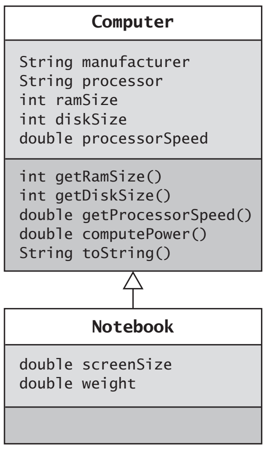
Inheritance in OOP is analogous to inheritance in humans. We all inherit genetic traits from our parents. If we are fortunate, we may even have some earlier ancestors who have left us an inheritance of monetary value. As we grow up, we benefit from our ancestors' resources, knowledge, and experiences, but our experiences will not affect how our parents or ancestors developed. Although we have two parents to inherit from, Java classes can have only one parent.

FIGURE 1.3
Classes `Mammal` and `Human`



Inheritance and hierarchical organization allow you to capture the idea that one thing may be a refinement or an extension of another. For example, an object that is a `Human` is a `Mammal` (the superclass of `Human`). This means that an object of type `Human` has all the data fields and methods defined by class `Mammal` (e.g., method `drinkMothersMilk`), but it may also have more data fields and methods that are not contained in class `Mammal` (e.g., method `thinkCreatively`). Figure 1.3 shows this simple hierarchy. The solid line in the UML class diagram shows that `Human` is a subclass of `Mammal`, and, therefore, `Human` objects can use methods `drinkMothersMilk` and `thinkCreatively`. Objects farther down the hierarchy are more complex and less general than those farther up. For this reason an object that is a `Human` is a `Mammal`, but the converse is not true because every `Mammal` object does not necessarily have the additional properties of a `Human`. Although this seems counterintuitive, the subclass `Human` is actually more powerful than the superclass `Mammal` because it may have additional attributes that are not present in the superclass.

FIGURE 1.4
Classes `NoteBook` and `Computer`



A Superclass and Subclass Example

To illustrate the concepts of inheritance and class hierarchies, let's consider a simple case of two classes: `Computer` and `Notebook`. A `Computer` object has a `manufacturer`, `processor`, `RAM`, and `disk`. A notebook computer is a kind of computer, so it has all the properties of a computer plus some additional features (screen size and weight). There may be other subclasses, such as tablet computer or game computer, but we will ignore them for now. We can define class `Notebook` as a subclass of class `Computer`. Figure 1.4 shows the class hierarchy.

Class `Computer`

Listing 1.2 shows class `Computer.Java`. It is defined like any other class. It contains a constructor, several accessors, a `toString` method, and a method `computePower`, which returns the product of its RAM size and processor speed as a simple measure of its power.

LISTING 1.2
Class Computer.java

```
/** Class that represents a computer. */
public class Computer {
    // Data Fields
    private String manufacturer;
    private String processor;
    private double ramSize;
    private int diskSize;
    private double processorSpeed;

    // Methods
    /** Initializes a Computer object with all properties specified.
        @param man The computer manufacturer
        @param processor The processor type
        @param ram The RAM size
        @param disk The disk size
        @param procSpeed The processor speed
     */
    public Computer(String man, String processor, double ram,
                    int disk, double procSpeed) {
```

```
        manufacturer = man;
        this.processor = processor;
        ramSize = ram;
        diskSize = disk;
        processorSpeed = procSpeed;
    }

    public double computePower() { return ramSize * processorSpeed; }
    public double getRamSize() { return ramSize; }
    public double getProcessorSpeed() { return processorSpeed; }
    public int getDiskSize() { return diskSize; }
    // Insert other accessor and modifier methods here.

    public String toString() {
        String result = "Manufacturer: " + manufacturer +
                        "\nCPU: " + processor +
                        "\nRAM: " + ramSize + " gigabytes" +
                        "\nDisk: " + diskSize + " gigabytes" +
                        "\nProcessor speed: " + processorSpeed + " gigahertz";
        return result;
    }
}
```

Use of `this.`

In the constructor for the `Computer` class, the statement

```
this.processor = processor;
```

sets data field `processor` in the object under construction to reference the same string as parameter processor. The prefix `this.` makes data field `processor` visible in the constructor. This is necessary because the declaration of `processor` as a parameter hides the data field declaration.

PITFALL

Not Using `this.` to Access a Hidden Data Field

If you write the preceding statement as

```
processor = processor; // Copy parameter processor to itself.
```

you will not get an error, but the data field `processor` in the `Computer` object under construction will not be initialized and will retain its default value (`null`). If you later attempt to use data field `processor`, you may get an error or just an unexpected result. Some IDEs will provide a warning if `this.` is omitted.

Class `Notebook`

In the `Notebook` class diagram in Figure 1.4, we show just the data fields declared in class `Notebook`; however, `Notebook` objects also have the data fields that are inherited from class `Computer` (`processor`, `ramSize`, and so forth). The first line in class `Notebook` (Listing 1.3),

```
public class Notebook extends Computer {
```

indicates that class `Notebook` extends class `Computer` and inherits its data and methods. Next, we define any additional data fields

```
// Data Fields
private double screenSize;
private double weight;
```

Initializing Data Fields in a Subclass

The constructor for class `Notebook` must begin by initializing the four data fields inherited from class `Computer`. Because those data fields are private to the superclass, Java requires that they be initialized by a superclass constructor. Therefore, a superclass constructor must be invoked as the first statement in the constructor body using a statement such as

```
super(man, proc, ram, disk, procSpeed);
```

This statement invokes the superclass constructor with the signature `Computer(String, String, double, int, double)`, passing the four arguments listed to the constructor. (A method signature consists of the method's name followed by its parameter types.) The following constructor for `Notebook` also initializes the data fields that are not inherited. Listing 1.3 shows class `Notebook`.

```
public Notebook(String man, String proc, double ram, int disk,
                double procSpeed, double screen, double wei) {
    super(man, proc, ram, disk, procSpeed);
    screenSize = screen;
    weight = wei;
}
```

SYNTAX super(. . .);

FORM:

```
super();
super(argumentList);
```

EXAMPLE:

```
super(man, proc, ram, disk, procSpeed);
```

MEANING:

The **`super()`** call in a class constructor invokes the superclass's constructor that has the corresponding *argumentList*. The superclass constructor initializes the inherited data fields as specified by its *argumentList*. The **`super()`** call must be the first statement in a constructor.

LISTING 1.3
Class `Notebook`

```
/** Class that represents a notebook computer. */
public class Notebook extends Computer {
    // Data Fields
    private double screenSize;
    private double weight;

    // Methods
    /** Initializes a Notebook object with all properties specified.
        @param man The computer manufacturer
        @param proc The processor type
        @param ram The RAM size
        @param disk The disk size
        @param procSpeed The processor speed
        @param screen The screen size
        @param wei The weight
     */
    public Notebook(String man, String proc, double ram, int disk,
                    double procSpeed, double screen, double wei) {
        super(man, proc, ram, disk, procSpeed);
        screenSize = screen;
        weight = wei;
    }
}
```

The No-Parameter Constructor

If the execution of any constructor in a subclass does not invoke a superclass constructor, Java automatically invokes the no-parameter constructor for the superclass. Java does this to initialize that part of the object inherited from the superclass before the subclass starts to initialize its part of the object. Otherwise, the part of the object that is inherited would remain uninitialized.

PITFALL

Not Defining the No-Parameter Constructor

If no constructors are defined for a class, the no-parameter constructor for that class will be provided by default. However, if any constructors are defined, the no-parameter constructor must also be defined explicitly if it needs to be invoked. Java does not provide it automatically because it may make no sense to create a new object of that type without providing initial data field values. (It was not defined in class `Notebook` or `Computer` because we want the client to specify some information about a `Computer` object when that object is created.) If the no-parameter constructor is defined in a subclass but is not defined in the superclass, you will get a syntax error `constructor not defined`. You can also get this error if a subclass constructor does not explicitly call a superclass constructor. There will be an implicit call to the no-parameter superclass constructor, so it must be defined.

Protected Visibility for Superclass Data Fields

The data fields inherited from class `Computer` have private visibility. Therefore, they can be accessed only within class `Computer`. Because it is fairly common for a subclass method to reference data fields declared in its superclass, Java provides a less restrictive form of visibility called *protected visibility*. A data field (or method) with protected visibility can be accessed in the class defining it, in any subclass of that class, or in any class in the same package. Therefore, if we had used the declaration

```
protected String manufacturer;
```

in class `Computer`, the following assignment statement would be valid in class `Notebook`:

```
manufacturer = man;
```

We will use protected visibility on occasion when we are writing a class that we intend to extend. However, in general, it is better to use private visibility because subclasses may be written by different programmers, and it is always a good practice to restrict and control access to the superclass data fields. We discuss visibility further in Section 1.7.

Is-a versus *Has-a* Relationships

One misuse of inheritance is confusing: the *has-a* relationship with the *is-a* relationship. The *is-a* relationship between classes means that one class is a subclass of the other class. For example, a game computer is a computer with specific attributes that make it suitable for gaming applications (enhanced graphics, fast processor) and is a subclass of the `Computer` class. The *is-a* relationship is achieved by extending a class.

The *has-a* relationship between classes means that one class has the second class as an attribute. For example, a game box is not really a computer (it is a kind of entertainment device), but it has a computer as a component. The *has-a* relationship is achieved by declaring a `Computer` data field in the game box class.

Another issue that sometimes arises is determining whether to define a new class in a hierarchy or whether a new object is a member of an existing class. For example, netbook computers have recently become very popular. They are smaller portable computers that can be used for general-purpose computing but are also used extensively for Web browsing. Should we define a separate class `NetBook`, or is a netbook computer a `Notebook` object with a small screen and low weight?

EXERCISES FOR SECTION 1.2

SELF-CHECK

1. Explain the effect of each valid statement in the following fragment. Indicate any invalid statements.

```
Computer c1 = new Computer();
Computer c2 = new Computer("Ace", "AMD", 8.0, 500, 3.5);
Notebook c3 = new Notebook("Ace", "AMD", 4.0, 500, 3.0);
Notebook c4 = new Notebook("Bravo", "Intel", 4.0, 750, 3.0, 15.5, 5.5);
System.out.println(c2.manufacturer + ", " + c4.processor);
System.out.println(c2.getDiskSize() + ", " + c4.getRamSize());
System.out.println(c2.toString() + "\n" + c4.toString());
```

2. Indicate where in the hierarchy you might want to add data fields for the following and the kind of data field you would add.

   Cost
   The battery identification
   Time before battery discharges
   Number of expansion slots
   Wireless Internet available

3. Can you add the following constructor to class `Notebook`? If so, what would you need to do to class `Computer`?

```
public Notebook() {}
```

PROGRAMMING

1. Write accessor and modifier methods for class `Computer`.

2. Write accessor and modifier methods for class `Notebook`.

1.3 Method Overriding, Method Overloading, and Polymorphism

In the preceding section, we discussed inherited data fields. We found that we could not access an inherited data field in a subclass object if its visibility was private. Next, we consider inherited methods. Methods generally have public visibility, so we should be able to access a method that is inherited. However, what if there are multiple methods with the same name in a class hierarchy? How does Java determine which one to invoke? We answer this question next.

Method Overriding

Let's use the following main method to test our class hierarchy.

```
/** Tests classes Computer and Notebook. Creates an object of each and
    displays them.
    @param args[] No control parameters
 */
public static void main(String[] args) {
    Computer myComputer =
        new Computer("Acme", "Intel", 4, 750, 3.5);
    Notebook yourComputer =
        new Notebook("DellGate", "AMD", 4, 500,
                     2.4, 15.0, 7.5);
    System.out.println("My computer is:\n" + myComputer.toString());
    System.out.println("\nYour computer is:\n" +
                       yourComputer.toString());
}
```

In the second call to `println`, the method call

```
yourComputer.toString()
```

applies method `toString` to object `yourComputer` (type `Notebook`). Because class `Notebook` doesn't define its own `toString` method, class `Notebook` inherits the `toString` method defined in class `Computer`. Executing this method displays the following output lines:

```
My computer is:
Manufacturer: Acme
CPU: Intel
RAM: 4.0 gigabytes
Disk: 750 gigabytes
Speed: 3.5 gigahertz

Your computer is:
Manufacturer: DellGate
CPU: AMD
RAM: 4.0 gigabytes
Disk: 500 gigabytes
Speed: 2.4 gigahertz
```

Unfortunately, this output doesn't show the complete state of object `yourComputer`. To show the complete state of a notebook computer, we need to define a `toString` method for class `Notebook`. If class `Notebook` has its own `toString` method, it will override the inherited method and will be invoked by the method call `yourComputer.toString()`. We define method `toString` for class `Notebook` next.

```
public String toString() {
    String result = super.toString() +
                    "\nScreen size: " + screenSize + " inches" +
                    "\nWeight: " + weight + " pounds";
    return result;
}
```

This method `Notebook.toString` returns a string representation of the state of a `Notebook` object. The first line

```
String result = super.toString()
```

uses method call **`super.`**`toString()` to invoke the `toString` method of the superclass (method `Computer.toString`) to get the string representation of the four data fields that are inherited from the superclass. The next two lines append the data fields defined in class `Notebook` to this string.

SYNTAX super.

FORM:

super.*methodName*()
super.*methodName*(*argumentList*)

EXAMPLE:

```
super.toString()
```

MEANING:

Using the prefix **`super.`** in a call to method *methodName* calls the method with that name defined in the superclass of the current class.

PROGRAM STYLE

Calling Method `toString()` Is Optional

In the `println` statement shown earlier,

```
System.out.println("My computer is:\n" + myComputer.toString());
```

the explicit call to method `toString` is not required. The statement could be written as

```
System.out.println("My computer is:\n" + myComputer);
```

Java automatically applies the `toString` method to an object referenced in a `String` expression. Normally, we will not explicitly call `toString`.

PITFALL

Overridden Methods Must Have Compatible Return Types

If you write a method in a subclass that has the same signature as one in the superclass but a different return type, you may get the following error message: `in` *`subclass-name`* `cannot override` *`method-name`* `in` *`superclass-name`*`; attempting to use incompatible return type`. The subclass method return type must be the same as or a subclass of the superclass method's return type.

Method Overloading

Let's assume we have decided to standardize and purchase our notebook computers from only one manufacturer. We could then introduce a new constructor with one less parameter for class `Notebook`.

```
public Notebook(String proc, int ram, int disk, double procSpeed,
                double screen, double wei) {
    this(DEFAULT_NB_MAN, proc, ram, disk, procSpeed, screen, wei);
}
```

The method call

```
this(DEFAULT_NB_MAN, proc, ram, disk, procSpeed, screen, wei);
```

invokes the six-parameter constructor (see Listing 1.3), passing on the five arguments it receives and the constant string `DEFAULT_NB_MAN` (defined in class `Notebook`). The six-parameter constructor begins by calling the superclass constructor, satisfying the requirement that it be called first. We now have two constructors with different signatures in class `Notebook`. Having multiple methods with the same name but different signatures in a class is called *method overloading*.

Now we have two ways to create new `Notebook` objects. Both of the following statements are valid:

```
Notebook lTP1 = new Notebook("Intel", 4, 500, 1.8, 14, 6.5);
Notebook lTP2 = new Notebook("MicroSys", "AMD", 4, 750, 3.0, 15, 7.5);
```

The manufacturer of `lTP1` is `DEFAULT_NB_MAN`.

SYNTAX **`this( . . . );`**

FORM:
`this(`*`argumentList`*`);`

EXAMPLE:
`this(DEFAULT_NB_MAN, proc, ram, disk, procSpeed);`

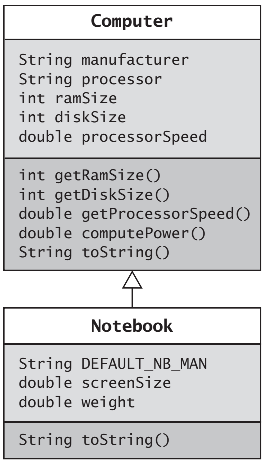
MEANING:
The call to **`this()`** invokes the constructor for the current class whose parameter list matches the argument list. The constructor initializes the new object as specified by its arguments. The invocation of another constructor (through either **`this()`** or **`super()`**) must be the first statement in a constructor.

Listing 1.4 shows the complete class `Notebook`. Figure 1.5 shows the UML diagram, revised to show that `Notebook` has a `toString` method and a constant data field. The next Pitfall discusses the reason for the `@Override` annotation preceding method `toString`.

LISTING 1.4
Complete Class `Notebook` with Method `toString`

```
/** Class that represents a notebook computer. */
public class Notebook extends Computer {
    // Data Fields
    private static final String DEFAULT_NB_MAN = "MyBrand";
    private double screenSize;
    private double weight;
```

FIGURE 1.5
Revised UML Diagram for `Computer` Class Hierarchy



```
    /** Initializes a Notebook object with all properties specified.
        @param man The computer manufacturer
        @param proc The processor type
        @param ram The RAM size
        @param disk The disk size
        @param screen The screen size
        @param wei The weight
     */
    public Notebook(String man, String proc, int ram, int disk,
                    double procSpeed, double screen, double wei) {
        super(man, proc, ram, disk, procSpeed);
        screenSize = screen;
        weight = wei;
    }

    /** Initializes a Notebook object with 6 properties specified. */
    public Notebook(String proc, int ram, int disk,
                    double procSpeed, double screen, double wei) {
        this(DEFAULT_NB_MAN, proc, ram, disk, procSpeed, screen, wei);
    }

    @Override
    public String toString() {
        String result = super.toString() +
                        "\nScreen size: " + screenSize + " inches" +
                        "\nWeight: " + weight + " pounds";
        return result;
    }
}
```

PITFALL

Overloading a Method When Intending to Override It

To override a method, you must use the same name and the same number and types of the parameters as the superclass method that is being overridden. If the name is the same but the number or types of the parameters are different, then the method is overloaded instead. Normally, the compiler will not detect this as an error. However, it is a sufficiently common error that a feature was added to the Java compiler so that programmers can indicate that they intend to override a method. If you precede the declaration of the method with the annotation `@Override`, the compiler will issue an error message if the method is overloaded instead of overridden.

PROGRAM STYLE

Precede an Overridden Method with the Annotation `@Override`

Whenever a method is overridden, we recommend preceding it with the annotation `@Override`. Some Java integrated development environments such as Netbeans and Eclipse will either issue a warning or add this annotation automatically.

Polymorphism

An important advantage of OOP is that it supports a feature called *polymorphism*, which means many forms or many shapes. Polymorphism enables the JVM to determine at run time which of the classes in a hierarchy is referenced by a superclass variable or parameter. Next we will see how this simplifies the programming process.

Suppose you are not sure whether a computer referenced in a program will be a notebook or a regular computer. If you declare the reference variable

```
Computer theComputer;
```

you can use it to reference an object of either type because a type `Notebook` object can be referenced by a type `Computer` variable. In Java, a variable of a superclass type (general) can reference an object of a subclass type (specific). `Notebook` objects are `Computer` objects with more features. When the following statements are executed,

```
theComputer = new Computer("Acme", "Intel", 2, 160, 2.6);
System.out.println(theComputer.toString());
```

you would see four output lines, representing the state of the object referenced by `theComputer`.

Now suppose you have purchased a notebook computer instead. What happens when the following statements are executed?

```
theComputer = new Notebook("Bravo", "Intel", 4, 240, 2.4. 15.0, 7.5);
System.out.println(theComputer.toString());
```

Recall that `theComputer` is type `Computer`. Will the `theComputer.toString()` method call return a string with all seven data fields or just the five data fields defined for a `Computer` object? The answer is a string with all seven data fields. The reason is that the type of the object receiving the `toString` message determines which `toString` method is called. Even though variable `theComputer` is type `Computer`, it references a type `Notebook` object, and the `Notebook` object receives the `toString` message. Therefore, the method `toString` for class `Notebook` is the one called.

This is an example of polymorphism. Variable `theComputer` references a `Computer` object at one time and a `Notebook` object another time. At compile time, the Java compiler can't determine what type of object `theComputer` will reference, but at run time, the JVM knows the type of the object that receives the `toString` message and can call the appropriate `toString` method.

EXAMPLE 1.2 If we declare the array `labComputers` as follows:

```
Computer[] labComputers = new Computer[10];
```

each subscripted variable `labComputers[i]` can reference either a `Computer` object or a `Notebook` object because `Notebook` is a subclass of `Computer`. For the method call `labComputers[i].toString()`, polymorphism ensures that the correct `toString` method is called. For each value of subscript `i`, the actual type of the object referenced by `labComputers[i]` determines which `toString` method will execute (`Computer.toString` or `Notebook.toString`).

Methods with Class Parameters

Polymorphism also simplifies programming when we write methods that have class parameters. For example, if we want to compare the power of two computers without polymorphism, we will need to write overloaded `comparePower` methods in class `Computer`, one for each subclass parameter and one with a class `Computer` parameter. However, polymorphism enables us to write just one method with a `Computer` parameter.

EXAMPLE 1.3 Method Computer.comparePowers compares the power of the Computer object it is applied to with the Computer object passed as its argument. It returns −1, 0, or +1 depending on which computer has more power. It does not matter whether this or aComputer references a Computer or a Notebook object.

```
/** Compares power of this computer and its argument computer
    @param aComputer The computer being compared to this computer
    @return -1 if this computer has less power,
            0 if the same, and
            +1 if this computer has more power.
 */
public int comparePower(Computer aComputer) {
    if (this.computePower() < aComputer.computePower())
        return -1;
    else if (this.computePower() == aComputer.computePower())
        return 0;
    else return 1;
}
```

EXERCISES FOR SECTION 1.3

SELF-CHECK

1. Explain the effect of each of the following statements. Which one(s) would you find in class Computer? Which one(s) would you find in class Notebook?

```
super(man, proc, ram, disk, procSpeed);
this(man, proc, ram, disk, procSpeed);
```

2. Indicate whether methods with each of the following signatures and return types (if any) would be allowed and in what classes they would be allowed. Explain your answers.

```
Computer()
Notebook()
int toString()
double getRamSize()
String getRamSize()
String getRamSize(String)
String getProcessor()
double getScreenSize()
```

3. For the loop body in the following fragment, indicate which method is invoked for each value of i. What is printed?

```
Computer comp[] = new Computer[3];
comp[0] = new Computer("Ace", "AMD", 8, 750, 3.5);
comp[1] = new Notebook("Dell", "Intel", 4, 500, 2.2, 15.5, 7.5);
comp[2] = comp[1];
for (int i = 0; i < comp.length; i++) {
    System.out.println(comp[i].getRamSize() + "\n" +
                       comp[i].toString());
}
```

4. When does Java determine which toString method to execute for each value of i in the for statement in the preceding question: at compile time or at run time? Explain your answer.

PROGRAMMING

1. Write constructors for both classes that allow you to specify only the processor, RAM size, and disk size.
2. Complete the accessor and modifier methods for class `Computer`.
3. Complete the accessor and modifier methods for class `Notebook`.

1.4 Abstract Classes

In this section, we introduce another kind of class called an *abstract class*. An abstract class is denoted by the use of the word *abstract* in its heading:

visibility `abstract class` *className*

An abstract class differs from an actual class (sometimes called a concrete class) in two respects:

- An abstract class cannot be instantiated.
- An abstract class may declare abstract methods.

Just as in an interface, an abstract method is declared through a method heading in the abstract class definition. This heading indicates the result type, method name, and parameters, thereby specifying the form that any actual method declaration must take:

visibility `abstract` *resultType methodName(parameterList)*;

However, the complete method definition, including the method body (implementation), does not appear in the abstract class definition.

In order to compile without error, an actual class that is a subclass of an abstract class must provide an implementation for each abstract method of its abstract superclass. The heading for each actual method must match the heading for the corresponding abstract method.

We introduce an abstract class in a class hierarchy when we need a base class for two or more actual classes that share some attributes. We may want to declare some of the attributes and define some of the methods that are common to these base classes. If, in addition, we want to require that the actual subclasses implement certain methods, we can accomplish this by making the base class an abstract class and declaring these methods abstract.

EXAMPLE 1.4 The Food Guide Pyramid provides a recommendation of what to eat each day based on established dietary guidelines. There are six categories of foods in the pyramid: fats, oils, and sweets; meats, poultry, fish, and nuts; milk, yogurt, and cheese; vegetables; fruits; and bread, cereal, and pasta. If we wanted to model the Food Guide Pyramid, we might have each of these as actual subclasses of an abstract class called `Food`:

```
/** Abstract class that models a kind of food. */
public abstract class Food {
    // Data Field
    private double calories;

    // Abstract Methods
    /** Calculates the percent of protein in a Food object. */
```

```
        public abstract double percentProtein();
        /** Calculates the percent of fat in a Food object. */
        public abstract double percentFat();
        /** Calculates the percent of carbohydrates in a Food object. */
        public abstract double percentCarbohydrates();

        // Actual Methods
        public double getCalories() { return calories; }
        public void setCalories(double cal) {
            calories = cal;
        }
    }
```

The three abstract method declarations

```
public abstract double percentProtein();
public abstract double percentFat();
public abstract double percentCarbohydrates();
```

impose the requirement that all actual subclasses implement these three methods. We would expect a different method definition for each kind of food. The keyword **abstract** must appear in all abstract method declarations in an abstract class. Recall that this is not required for abstract method declarations in interfaces.

SYNTAX Abstract Class Definition

FORM:

```
public abstract class className {
    data field declarations
    abstract method declarations
    actual method definitions
}
```

EXAMPLE:

```
public abstract class Food {
    // Data Field
    private double calories;

    // Abstract Methods
    public abstract double percentProtein();
    public abstract double percentFat();
    public abstract double percentCarbohydrates();

    // Actual Methods
    public double getCalories() { return calories; }
    public void setCalories(double cal) {
        calories = cal;
    }
}
```

INTERPRETATION:

Abstract class *className* is defined. The class body may have declarations for data fields and abstract methods as well as actual method definitions. Each abstract method declaration consists of a method heading containing the keyword **abstract**. All of the declaration kinds shown above are optional.

PITFALL

Omitting the Definition of an Abstract Method in a Subclass

If you write class `Vegetable` and forget to define method `percentProtein`, you will get the syntax error class `Vegetable should be declared abstract, it does not define method percentProtein in class Food`. Although this error message is misleading (you did not intend `Vegetable` to be abstract), any class with undefined methods is abstract by definition. The compiler's rationale is that the undefined method is intentional, so `Vegetable` must be an abstract class, with a subclass that defines `percentProtein`.

Referencing Actual Objects

Because class `Food` is abstract, you can't create type `Food` objects. However, you can use a type `Food` variable to reference an actual object that belongs to a subclass of type `Food`. For example, an object of type `Vegetable` can be referenced by a `Vegetable` or `Food` variable because `Vegetable` is a subclass of `Food` (i.e., a `Vegetable` object is also a `Food` object).

EXAMPLE 1.5 The following statement creates a `Vegetable` object that is referenced by variable `mySnack` (type `Food`).

```
    Food mySnack = new Vegetable("carrot sticks");
```

Initializing Data Fields in an Abstract Class

An abstract class can't be instantiated. However, an abstract class can have constructors that initialize its data fields when a new subclass object is created. The subclass constructor will use `super(...)` to call such a constructor.

Abstract Class Number and the Java Wrapper Classes

The abstract class `Number` is predefined in the Java class hierarchy. It has as its subclasses all the wrapper classes for primitive numeric types (e.g., `Byte`, `Double`, `Integer`, and `Short`). A *wrapper class* is used to store a primitive-type value in an object type. Each wrapper class contains constructors to create an object that stores a particular primitive-type value. For example, `Integer(35)` or `Integer("35")` creates a type `Integer` object that stores the `int 35`. A wrapper class also has methods for converting the value stored to a different numeric type.

Figure 1.6 shows a portion of the class hierarchy with base class `Number`. Italicizing the class name `Number` in its class box indicates that `Number` is an abstract class and, therefore, cannot be instantiated. Listing 1.5 shows part of the definition for class `Number`. Two abstract methods are declared `(intValue and doubleValue)`, and one actual method `(byteValue)` is defined.

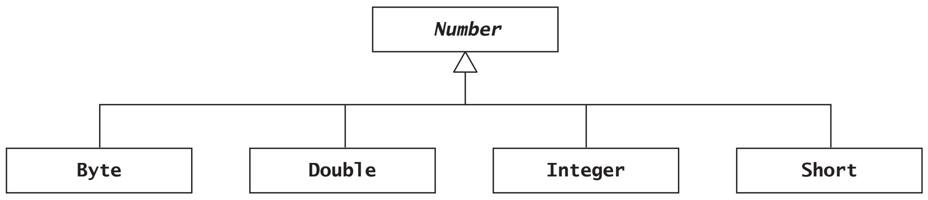


FIGURE 1.6
The Abstract Class `Number` and Selected Subclasses

In the actual implementation of `Number`, the body of `byteValue` would be provided, but we just indicate its presence in Listing 1.5.

LISTING 1.5
Part of Abstract Class java.lang.Number

```
public abstract class Number {
    // Abstract Methods
    /** Returns the value of the specified number as an int.
        @return The numeric value represented by this object after
                conversion to type int
     */
    public abstract int intValue();
    /** Returns the value of the specified number as a double.
        @return The numeric value represented by this object
                after conversion to type double
     */
    public abstract double doubleValue();

    ...

    // Actual Methods
    /** Returns the value of the specified number as a byte.
        @return The numeric value represented by this object
                after conversion to type byte
     */
    public byte byteValue() {
        // Implementation not shown.
        ...
    }
}
```

Summary of Features of Actual Classes, Abstract Classes, and Interfaces

It is easy to confuse abstract classes, interfaces, and actual classes (concrete classes). Table 1.1 summarizes some important points about these constructs.

A class (abstract or actual) can extend only one other class; however, there is no restriction on the number of interfaces a class can implement. An interface cannot extend a class.

TABLE 1.1
Comparison of Actual Classes, Abstract Classes, and Interfaces

Property	Actual Class	Abstract Class	Interface
Instances (objects) of this can be created	Yes	No	No
This can define instance variables	Yes	Yes	No
This can define methods	Yes	Yes	Yes
This can define constants	Yes	Yes	Yes
The number of these a class can extend	0 or 1	0 or 1	0
The number of these a class can implement	0	0	Any number
This can extend another class	Yes	Yes	No
This can declare abstract methods	No	Yes	Yes
Variables of this type can be declared	Yes	Yes	Yes

An abstract class may implement an interface just as an actual class does, but unlike an actual class, it doesn't have to define all of the methods declared in the interface. It can leave the implementation of some of the abstract methods to its subclasses.

Both abstract classes and interfaces declare abstract methods. However, unlike an interface, an abstract class can also have data fields and methods that are not abstract. You can think of an abstract class as combining the properties of an actual class, by providing inherited data fields and methods to its subclasses, and of an interface, by specifying requirements on its subclasses through its abstract method declarations.

Implementing Multiple Interfaces

A class can extend only one other class, but it may extend more than one interface. For example, assume interface `StudentInt` specifies methods required for student-like classes and interface `EmployeeInt` specifies methods required for employee-like classes. The following header for class `StudentWorker`

```
public class StudentWorker implements StudentInt, EmployeeInt
```

means that class `StudentWorker` must define (provide code for) all of the abstract methods declared in both interfaces. Therefore, class `StudentWorker` supports operations required for both interfaces.

Extending an Interface

Interfaces can also extend other interfaces. In Chapter 2 we will introduce the Java Collection Framework. This class hierarchy contains several interfaces and classes that manage the collection of objects. At the top of this hierarchy is the interface `Iterable`, which declares the method `iterator`. At the next lower level is interface `Collection`, which extends `Iterable`. This means that all classes that implement `Collection` must also implement `Iterable` and therefore must define the method `iterator`.

An interface can extend more than one other interface. In this case, the resulting interface includes the union of the methods defined in the superinterfaces. For example, we can define the interface `ComparableCollection`, which extends both `Comparable` and `Collection`, as follows:

```
public interface ComparableCollection extends Comparable, Collection { }
```

Note that this interface does not define any methods itself but does require any implementing class to implement all of the methods required by `Comparable` and by `Collection`.

EXERCISES FOR SECTION 1.4

SELF-CHECK

1. What are two important differences between an abstract class and an actual class? What are the similarities?

2. What do abstract methods and interfaces have in common? How do they differ?

3. Explain the effect of each statement in the following fragment and trace the loop execution for each value of `i`, indicating which `doubleValue` method executes, if any. What is the final value of x?

```
Number[] nums = new Number[5];
nums[0] = new Integer(35);
nums[1] = new Double(3.45);
nums[4] = new Double("2.45e6");
double x = 0;
```

```
for (int i = 0; i < nums.length; i++) {
    if (nums[i] != null)
        x += nums[i].doubleValue();
}
```

4. What is the purpose of the `if` statement in the loop in Question 3? What would happen if it were omitted?

PROGRAMMING

1. Write class `Vegetable`. Assume that a vegetable has three `double` constants: `VEG_FAT_CAL`, `VEG_PROTEIN_CAL`, and `VEG_CARBO_CAL`. Compute the fat percentage as `VEG_FAT_CAL` divided by the sum of all the constants.
2. Earlier we discussed a `Computer` class with a `Notebook` class as its only subclass. However, there are many different kinds of computers. An organization may have servers, mainframes, desktop PCs, and notebooks. There are also personal data assistants and game computers. So it may be more appropriate to declare class `Computer` as an abstract class that has an actual subclass for each category of computer. Write an abstract class `Computer` that defines all the methods shown earlier and declares an abstract method with the signature `costBenefit(double)` that returns the cost–benefit (type `double`) for each category of computer.

1.5 Class `Object` and Casting

The class `Object` is a special class in Java because it is the root of the class hierarchy, and every class has `Object` as a superclass. All classes inherit the methods defined in class `Object`; however, these methods may be overridden in the current class or in a superclass (if any). Table 1.2 shows a few of the methods of class `Object`. We discuss method `toString` next and the other `Object` methods shortly thereafter.

The Method `toString`

You should always override the `toString` method if you want to represent an object's state (information stored). If you don't override it, the `toString` method for class `Object` will execute and return a string, but not what you are expecting.

EXAMPLE 1.6 If we didn't have a `toString` method in class `Computer` or `Notebook`, the method call `aComputer.toString()` would call the `toString` method inherited from class `Object`. This method would return a string such as `Computer@ef08879`, which shows the object's class name and a special integer value that is its "hash code"—not its state. Method `hashCode` is discussed in Chapter 7.

TABLE 1.2
The Class `Object`

Method	Behavior
`boolean equals(Object obj)`	Compares this object to its argument
`int hashCode()`	Returns an integer hash code value for this object
`String toString()`	Returns a string that textually represents the object
`Class<?> getClass()`	Returns a unique object that identifies the class of this object

Operations Determined by Type of Reference Variable

You have seen that a variable can reference an object whose type is a subclass of the variable type. Because `Object` is a superclass of class `Integer`, the statement

```
Object aThing = new Integer(25);
```

will compile without error, creating the object reference shown in Figure 1.7. However, even though `aThing` references a type `Integer` object, we can't process this object like other `Integer` objects. For example, the method call `aThing.intValue()` would cause the syntax error method `intValue()` not found in class `java.lang.Object`. The reason for this is that the type of the reference, not the type of the object referenced, determines what operations can be performed, and class `Object` doesn't have an `intValue` method. During compilation, Java can't determine what kind of object will be referenced by a type `Object` variable, so the only operations permitted are those defined for class `Object`. The type `Integer` instance methods not defined in class `Object` (e.g., `intValue` and `doubleValue`) can't be invoked.

The method call `aThing.equals(new Integer("25"))` will compile because class `Object` has an `equals` method, and a subclass object has everything that is defined in its superclass. During execution, the `equals` method for class `Integer` is invoked, not class `Object`. (Why?)

Another surprising result is that the assignment statement

```
Integer aNum = aThing;  // incompatible types
```

won't compile even though `aThing` references a type `Integer` object. The syntax error: `incompatible types: found: Java.lang.Object, required: Java.lang.Integer` indicates that the expression type is incorrect (type `Object`, not type `Integer`). The reason Java won't compile this assignment is that Java is a strongly typed language, so the Java compiler always verifies that the type of the expression (`aThing` is type `Object`) being assigned is compatible with the variable type (`aNum` is type `Integer`). We show how to use casting to accomplish this in the next section.

Strong typing is also the reason that `aThing.intValue()` won't compile; the method invoked must be an instance method for class `Object` because `aThing` is type `Object`.

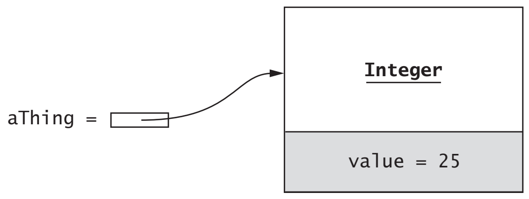


FIGURE 1.7
Type `Integer` Object Referenced by `aThing` (Type `Object`)

DESIGN CONCEPT

The Importance of Strong Typing

Suppose Java did not check the expression type and simply performed the assignment

```
Integer aNum = aThing;  // incompatible types
```

Farther down the line, we might attempt to apply an `Integer` method to the object referenced by `aNum`. Because `aNum` is type `Integer`, the compiler would permit this. If `aNum` were referencing a type `Integer` object, then performing this operation would do no harm. But if `aNum` was referencing an object that was not type `Integer`, performing this operation would cause either a run-time error or an undetected logic error. It is much better to have the compiler tell us that the assignment is invalid.

Casting in a Class Hierarchy

Java provides a mechanism, *casting*, that enables us to process the object referenced by `aThing` through a reference variable of its actual type, instead of through a type `Object` reference. The expression

```
(Integer) aThing
```

casts the type of the object referenced by `aThing` (type `Object`) to type `Integer`. The casting operation will succeed only if the object referenced by `aThing` is, in fact, type `Integer`; if not, a `ClassCastException` will be thrown.

What is the advantage of performing the cast? Casting gives us a type `Integer` reference to the object in Figure 1.7 that can be processed just like any other type `Integer` reference. The expression

```
((Integer) aThing).intValue()
```

will compile because now `intValue` is applied to a type `Integer` reference. Note that all parentheses are required so that method `intValue` will be invoked after the cast. Similarly, the assignment statement

```
Integer aNum = (Integer) aThing;
```

is valid because a type `Integer` reference is being assigned to `aNum` (type `Integer`).

Keep in mind that the casting operation does not change the object referenced by `aThing`; instead, it creates a type `Integer` reference to it. (This is called an *anonymous* or *unnamed reference*.) Using the type `Integer` reference, we can invoke any instance method of class `Integer` and process the object just like any other type `Integer` object.

The cast

```
(Integer) aThing
```

is called a downcast because we are casting from a higher type `(Object)` to a lower type `(Integer)`. It is analogous to a narrowing cast when dealing with primitive types:

```
double x = . . . ;
int count = (int) x;  // Narrowing cast, double is wider type than int
```

You can downcast from a more general type (a superclass type) to a more specific type (a subclass type) in a class hierarchy, provided that the more specific type is the same type as the object being cast (e.g., `(Integer) aThing`). You can also downcast from a more general type to a more specific type that is a superclass of the object being cast (e.g., `(Number) aThing`). *Upcasts* (casting from a more specific type to a more general type) are always valid; however, they are unnecessary and are rarely done.

PITFALL

Performing an Invalid Cast

Assume that `aThing` (type `Object`) references a type `Integer` object as before, and you want to get its string representation. The downcast

```
(String) aThing  // Invalid cast
```

is invalid and would cause a `ClassCastException` (a subclass of `RuntimeException`) because `aThing` references a type `Integer` object, and a type `Integer` object cannot be downcast to type `String` (`String` is not a superclass of `Integer`). However, the method call `aThing.toString()` is valid (and returns a string) because type `Object` has a `toString` method. (Which `toString` method would be called: `Object.toString` or `Integer.toString`?)

Using `instanceof` to Guard a Casting Operation

In the preceding Pitfall, we mentioned that a `ClassCastException` occurs if we attempt an invalid casting operation. Java provides the `instanceof` operator, which you can use to guard against this kind of error.

EXAMPLE 1.7 The following array `stuff` can store 10 objects of any data type because every object type is a subclass of `Object`.

```
Object[] stuff = new Object[10];
```

Assume that the array `stuff` has been loaded with data, and we want to find the sum of all numbers that are wrapped in objects. We can use the following loop to do so:

```
double sum = 0;
for (int i = 0; i < stuff.length; i++) {
    if (stuff[i] instanceof Number) {
        Number next = (Number) stuff[i];
        sum += next.doubleValue();
    }
}
```

The **if** condition (`stuff[i] instanceof Number`) is true if the object referenced by `stuff[i]` is a subclass of `Number`. It would be false if `stuff[i]` referenced a `String` or other nonnumeric object. The statement

```
Number next = (Number) stuff[i];
```

casts the object referenced by `stuff[i]` (type `Object`) to type `Number` and then references it through variable `next` (type `Number`). The variable `next` contains a reference to the same object as does `stuff[i]`, but the type of the reference is different (type `Number` instead of type `Object`). Then the statement

```
sum += next.doubleValue();
```

invokes the appropriate `doubleValue` method to extract the numeric value and add it to `sum`. Rather than declare variable `next`, you could write the **if** statement as

```
if (stuff[i] instanceof Number)
    sum += ((Number) stuff[i]).doubleValue();
```

PROGRAM STYLE

Polymorphism Eliminates Nested `if` Statements

If Java didn't support polymorphism, the `if` statement in Example 1.7 would be much more complicated. You would need to write something like the following:

```
// Inefficient code that does not take advantage of polymorphism
if (stuff[i] instanceof Integer)
    sum += ((Integer) stuff[i]).doubleValue();
else if (stuff[i] instanceof Double)
    sum += ((Double) stuff[i]).doubleValue();
else if (stuff[i] instanceof Float)
    sum += ((Float) stuff[i]).doubleValue();
...
```

Each condition here uses the `instanceof` operator to determine the data type of the actual object referenced by `stuff[i]`. Once the type is known, we cast to that type and call its `doubleValue` method. Obviously, this code is very cumbersome and is more likely to be flawed than the original **`if`** statement. More importantly, if a new wrapper class is defined for numbers, we would need to modify the **`if`** statement to process objects of this new class type. So be wary of selection statements like the one shown here; their presence often indicates that you are not taking advantage of polymorphism.

EXAMPLE 1.8 Suppose we have a class `Employee` with the following data fields:

```
public class Employee {
    // Data Fields
    private String name;
    private double hours;
    private double rate;
    private Address address;
...
```

To determine whether two `Employee` objects are equal, we could compare all four data fields. However, it makes more sense to determine whether two objects are the same employee by comparing their name and address data fields. Below, we show a method `equals` that overrides the `equals` method defined in class `Object`. By overriding this method, we ensure that the `equals` method for class `Employee` will always be called when method `equals` is applied to an `Employee` object. If we had declared the parameter type for `Employee.equals` as type `Employee` instead of `Object`, then the `Object.equals` method would be called if the argument was any data type except `Employee`.

```
/** Determines whether the current object matches its argument.
    @param obj The object to be compared to the current object
    @return true if the objects have the same name and address;
            otherwise, return false
 */
@Override
public boolean equals(Object obj) {
    if (obj == this) return true;
    if (obj == null) return false;
    if (this.getClass() == obj.getClass()) {
        Employee other = (Employee) obj;
        return name.equals(other.name) &&
               address.equals(other.address);
    } else {
        return false;
    }
}
```

If the object referenced by `obj` is not type `Employee`, we return `false`. If it is type `Employee`, we downcast that object to type `Employee`. After the downcast, the return statement calls method `String.equals` to compare the name field of the current object to the name field of object `other`, and method `Address.equals` to compare the two `address` data fields. Therefore, method `equals` must also be defined in class `Address`. The method result is `true` if both the `name` and `address` fields match, and it is false if one or both fields do not match. The method result is also `false` if the downcast can't be performed because the argument is an incorrect type or **`null`**.

The `Class` Class

Every class has a `Class` object that is automatically created when the class is loaded into an application. The `Class` class provides methods that are mostly beyond the scope of this text. The important point is that each `Class` object is unique for the class, and the `getClass` method (a member of `Object`) will return a reference to this unique object. Thus, if `this.getClass() == obj.getClass()` in Example 1.8 is true, then we know that `obj` and `this` are both of class `Employee`.

EXERCISES FOR SECTION 1.5

SELF-CHECK

1. Indicate the effect of each of the following statements:

```
    Object o = new String("Hello");
    String s = o;
    Object p = 25;
    int k = p;
    Number n = k;
```

2. Rewrite the invalid statements in Question 1 to remove the errors.

PROGRAMMING

1. Write an `equals` method for class `Computer` (Listing 1.2).
2. Write an `equals` method for class `Notebook` (Listing 1.4).
3. Write an `equals` method for the following class. What other `equals` methods should be defined?

```
public class Airplane {
    // Data Fields
    Engine eng;
    Rudder rud;
    Wing[] wings = new Wing[2];
    ...
}
```

1.6 A Java Inheritance Example—The `Exception` Class Hierarchy

Next we show how Java uses inheritance to build a class hierarchy that is fundamental to detecting and correcting errors during program execution (run-time errors). A run-time error occurs during program execution when the Java Virtual Machine (JVM) detects an operation that it knows to be incorrect. A run-time error will cause the JVM to *throw an exception*—that is, to create an object of an exception type that identifies the kind of incorrect operation and also interrupts normal processing. Table 1.3 shows some examples of exceptions that are run-time errors. All are subclasses of class `RuntimeException`. Following are some examples of the exceptions listed in the table.

Division by Zero

If `count` represents the number of items being processed and it is possible for `count` to be zero, then the assignment statement

```
    average = sum / count;
```

TABLE 1.3
Subclasses of java.lang.RuntimeException

Class	Cause/Consequence
`ArithmeticException`	An attempt to perform an integer division by zero
`ArrayIndexOutOfBoundsException`	An attempt to access an array element using an index (subscript) less than zero or greater than or equal to the array's length
`NumberFormatException`	An attempt to convert a string that is not numeric to a number
`NullPointerException`	An attempt to use a `null` reference value to access an object
`NoSuchElementException`	An attempt to get a next element after all elements were accessed
`InputMismatchException`	The token returned by a `Scanner next...` method does not match the pattern for the expected data type

can cause a division-by-zero error. If `sum` and `count` are **`int`** variables, this error is indicated by the JVM throwing an `ArithmeticException`. You can easily guard against such a division with an **`if`** statement so that the division operation will not be performed when `count` is zero:

```
if (count == 0)
    average = 0;
else
    average = sum / count;
```

Normally, you would compute an average as a `double` value, so you could cast an `int` value in `sum` to type `double` before doing the division. In this case, an exception is not thrown if `count` is zero. Instead, `average` will have one of the special values `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY`, or `Double.NaN` depending on whether `sum` was positive, negative, or zero.

Array Index Out of Bounds

An `ArrayIndexOutOfBoundsException` is thrown by the JVM when an index value (subscript) used to access an element in an array is less than zero or greater than or equal to the array's length. For example, suppose we define the array `scores` as follows:

```
int[] scores = new int[500];
```

The subscripted variable `scores[i]` uses `i` (type **`int`**) as the array index. An `ArrayIndexOutOfBoundsException` will be thrown if `i` is less than zero or greater than 499.

Array index out of bounds errors can be prevented by carefully checking the boundary values for an index that is also a loop control variable. A common error is using the array size as the upper limit rather than the array size minus 1.

EXAMPLE 1.9 The following loop would cause an `ArrayIndexOutOfBoundsException` on the last pass, when `i` is equal to `x.length`.

```
for (int i = 0; i <= x.length; i++)
    x[i] = i * i;
```

The loop repetition test should be `i < x.length`.

`NumberFormatException` and `InputMismatchException`

The `NumberFormatException` is thrown when a program attempts to convert a nonnumeric string (usually a data value) to a numeric value. For example, if the user types in the string `"2.6e"`, method `parseDouble`, in the following code:

```
String speedStr = JOptionPane.showInputDialog("Enter speed");
double speed = Double.parseDouble(speedStr);
```

would throw a `NumberFormatException` because `"2.6e"` is not a valid numeric string (it has no exponent after the `e`). There is no general way to avoid this exception because it is impossible to guard against all possible data entry errors the user can make.

A similar error can occur if you are using a `Scanner` object for data entry. If `scan` is a `Scanner`, the statement

```
double speed = scan.nextDouble();
```

will throw an `InputMismatchException` if the next token scanned is `"2.6e"`.

Null Pointer

The `NullPointerException` is thrown when there is an attempt to access an object that does not exist; that is, the reference variable being accessed contains a special value, known as `null`. You can guard against this by testing for `null` before invoking a method.

The `Exception` Class Hierarchy

The exceptions in Table 1.3 are all subclasses of `Runtime`. All Exception classes are defined within a class hierarchy that has the class `Throwable` as its superclass (see the UML diagram in Figure 1.8). The UML diagram shows that classes `Error` and `Exception` are subclasses of `Throwable`. Each of these classes has subclasses that are shown in the figure. We will focus on class `Exception` and its subclasses in this chapter. Because `RuntimeException` is a subclass of `Exception`, it is also a subclass of `Throwable` (the subclass relationship is transitive).

The Class `Throwable`

The class `Throwable` is the superclass of all exceptions. The methods that you will use from class `Throwable` are summarized in Table 1.4. Because all exception classes are subclasses of class `Throwable`, they can call any of its methods including `getMessage`, `printStackTrace`, and `toString`. If `ex` is an `Exception` object, the call

```
ex.printStackTrace();
```

FIGURE 1.8
Summary of `Exception` Class Hierarchy

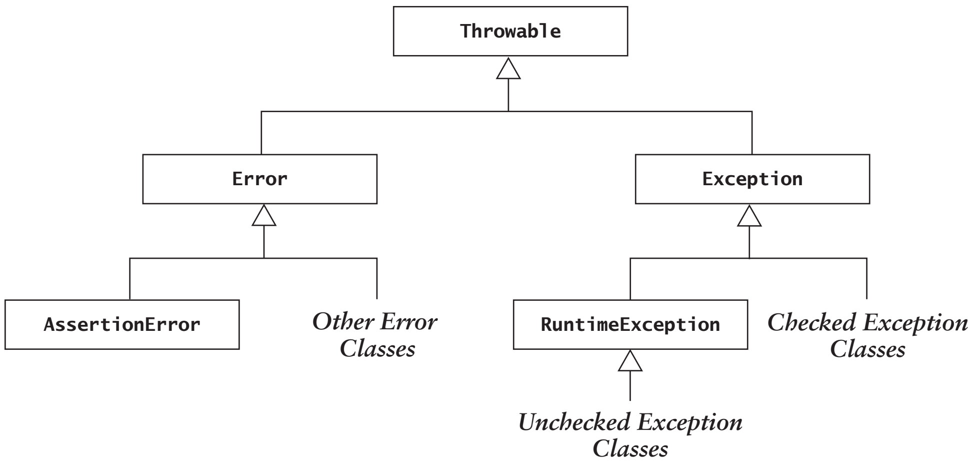


TABLE 1.4
Summary of Commonly Used Methods from the `java.lang.Throwable` Class

Method	Behavior
`String getMessage()`	Returns the detail message
`void printStackTrace()`	Prints the stack trace to System.err
`String toString()`	Returns the name of the exception followed by the detail message

displays a stack trace, discussed in Appendix A (Section A.11). The statement

```
System.err.println(ex.getMessage());
```

displays a detail message (or error message) describing the exception. The statement

```
System.err.println(ex.toString);
```

displays the name of the exception followed by the detail message.

Checked and Unchecked Exceptions

There are two categories of exceptions: *checked* and *unchecked*. A checked exception is an error that is normally not due to programmer error and is beyond the control of the programmer. All exceptions caused by input/output errors are considered checked exceptions. For example, if the programmer attempts to access a data file that is not available because of a user or system error, a `FileNotFoundException` is thrown. The class `IOException` and its subclasses (see Table 1.5) are checked exceptions. Even though checked exceptions are beyond the control of the programmer, the programmer must be aware of them and must handle them in some way (discussed later). All checked exceptions are subclasses of `Exception`, but they are not subclasses of `RuntimeException`. Figure 1.9 is a more complete diagram of the Exception hierarchy.

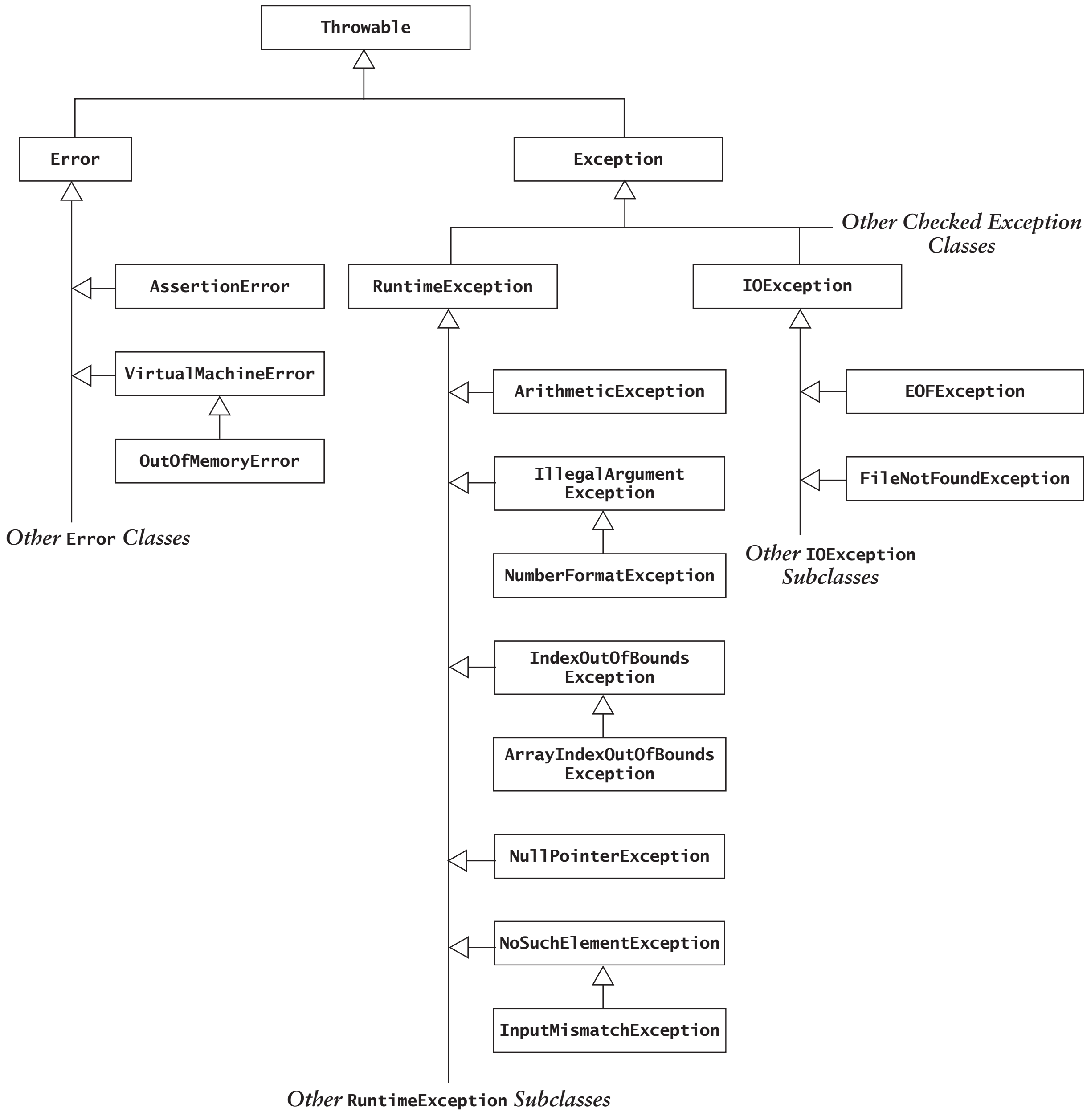
The *unchecked* exceptions represent error conditions that may occur as a result of programmer error or of serious external conditions that are considered unrecoverable. For example, exceptions such as `NullPointerException` or `ArrayIndexOutOfBounds-Exception` are unchecked exceptions that are generally due to programmer error. These exceptions are all subclasses of `RuntimeException`. While you can sometimes prevent these exceptions via defensive programming, it is impractical to try to prevent them all or to provide exception handling for all of them. Therefore, you can handle these exceptions, but Java does not require you to do so.

The class `Error` and its subclasses represent errors that are due to serious external conditions. An example of such an error is `OutOfMemoryError`, which is thrown when there is no memory available. You can't foresee or guard against these kinds of errors. You can attempt to handle these exceptions, but you are strongly discouraged from trying to do so because you probably will be unsuccessful. For example, if an `OutOfMemoryError` is thrown, there is no memory available to process the exception-handling code, so the exception would be thrown again.

TABLE 1.5
Class `java.io.IOException` and Some Subclasses

Exception Class	Cause
`IOException`	Some sort of input/output error
`EOFException`	Attempt to read beyond the end of data with a `DataInputStream`
`FileNotFoundException`	Inability to find a file

FIGURE 1.9
Exception Hierarchy Showing Selected Checked and Unchecked Exceptions



How do we know which exceptions are checked and which are unchecked? Exception classes that are subclasses of `RuntimeException` and `Error` are unchecked. All other exception classes are checked exceptions.

We discuss Java exceptions further in Appendix A Section A.11 describes how to use the `try-catch` statement to handle different kinds of exceptions; Section A.12 shows how to write statements that throw exceptions when your code detects an error during run time.

Handling Exceptions to Recover from Errors

Exceptions enable Java programmers to write code that can report errors and sometimes recover from them. The key to this process is the **`try-catch`** sequence. We will cover the essentials of catching and throwing exceptions in this section. A complete discussion is provided in Sections A.11 and A.12.

The `try-catch` Sequence

The **`try-catch`** sequence is used to catch and handle exceptions. It resembles an **`if-then-else`** statement. It consists of one **`try`** block followed by one or more **`catch`** clauses. The statements in the **`try`** block are executed first. If they execute without error, the **`catch`** clauses are skipped. If a statement in the **`try`** block throws an exception, the rest of the statements in the **`try`** block are skipped and execution continues with the statements in the **`catch`** clause for that particular type of exception. If there is no **`catch`** clause for that exception type, the exception is rethrown to the calling method. If the `main` method is reached and no appropriate **`catch`** clause is located, the program terminates with an unhandled exception error. A **`try`** block with two **`catch`** clauses follows.

```
try {
    // Execute the following statements until an exception is thrown
    ...
    // Skip the catch blocks if no exceptions were thrown
} catch (ExceptionTypeA ex) {
    // Execute this catch block if an exception of type ExceptionTypeA
    // was thrown in the try block
    ...
} catch (ExceptionTypeB ex) {
    // Execute this catch block if an exception of type ExceptionTypeB
    // was thrown in the try block
    ...
}
```

A **`catch`** clause header resembles a method header. The expression in parentheses in the **`catch`** clause header is like a method parameter declaration (the parameter is `ex`). The statements in curly braces, the **`catch`** block, execute if the exception that was thrown is the specified exception type or is a subclass of that exception type.

PITFALL

Unreachable `catch` block

In the above, `ExceptionTypeA` cannot be a superclass of `ExceptionTyeB`. If it is, `ExceptionTypeB` is considered unreachable because its exceptions would be caught by the first `catch` clause.

Using `try-catch` to Recover from an Error

One common source of exceptions is user input. For example, the `Scanner nextInt` method is supposed to read a type **`int`** value. If an **`int`** is not the next item read, Java throws an `InputMismatchException`. Rather than have this problem terminate the program, you can read the data value in a **`try`** block and catch an `InputMismatchException` in the **`catch`** clause. If one is thrown, you can give the user another chance to enter an integer as shown in method `getIntValue`, as follows.

```
/** Reads an integer using a scanner.
    @return the first integer read.
*/
public static int getIntValue(Scanner scan) {
    int nextInt = 0;          // next int value
    boolean validInt = false; // flag for valid input
    while (!validInt) {
        try {
            System.out.println("Enter number of kids:");
            nextInt = scan.nextInt();
            validInt = true;
        }  catch (InputMismatchException ex) {
            scan.nextLine();   // clear buffer
            System.out.println("Bad data -- enter an integer:");
        }
    }
    return nextInt;
}
```

The **while** loop repeats while validInt is **false** (its initial value). The **try** block attempts to read a type **int** value using Scanner scan. If the user enters an integer, validInt is set to **true** and the **try-catch** statement and **while** loop are exited. The integer data value will be returned as the method result.

If the user enters a data item that is not an integer, however, Java throws an InputMismatchException. This is caught by the **catch** clause

```
catch (InputMismatchException ex)
```

The first statement in the **catch** block clears the Scanner buffer, and the user is prompted to enter an integer. Because validInt is still false, the while loop repeats until the user successfully enters an integer.

Throwing an Exception When Recovery Is Not Obvious

In the last example, method getIntValue was able to recover from a bad data item by giving the user another chance to enter data. In some cases, you may be able to write code that detects certain kinds of errors, but there may not be an obvious way to recover from them. In these cases, the best approach is just to throw an exception reporting the error to the method that called it. The caller can then catch the exception and handle it.

Method processPositiveInteger requires a positive integer as its argument. If the argument is not positive, there is no reason to continue executing the method because the result may be meaningless, or the method execution may cause a different exception to be thrown, which could confuse the method caller. There is also no obvious way to correct this error because the method has no way of knowing what n should be, so a **try-catch** sequence would not fix the problem.

```
public static void processPositiveInteger(int n) {
    if (n < 0)
        throw new IllegalArgumentException(
                "Invalid negative argument");
    else {
         // Process n as required
         //...
         System.out.println("Finished processing " + n);
    }
}
```

If the argument n is not positive, the statement

```
throw new IllegalArgumentException("Invalid negative argument");
```

executes and throws an `IllegalArgumentException` object. The string in the last line is stored in the exception object's `message` data field, and the method is exited, returning control to the caller. The caller is then responsible for handling this exception. If possible, the caller may be able to recover from this error and would attempt to do so.

The `main` method, which follows, calls both `getIntValue` and `processPositiveInteger` in the **`try`** block. If an `IllegalArgumentException` is thrown, the message `invalid negative argument` is displayed, and the program terminates with an error indication. If no exception is thrown, the program exits normally.

```
public static void main(String[] args) {
    Scanner scan = new scanner (system.in);
    try {
        int num = getIntValue(scan);
        processPositiveInteger(num);
    } catch (IllegalArgumentException ex) {
        System.err.println(ex.getMessage());
        System.exit(1); // error indication
    }
    System.exit(0);    //normal exit
}
```

EXERCISES FOR SECTION 1.6

SELF-CHECK

1. Explain the key difference between checked and unchecked exceptions. Give an example of each kind of exception. What criterion does Java use to decide whether an exception is checked or unchecked?
2. What is the difference between the kind of unchecked exceptions in class `Error` and the kind in class `Exception`?
3. List four subclasses of `RuntimeException`.
4. List two subclasses of `IOException`.
5. What happens in the `main` method preceding the exercises if an exception of a different type occurs in method `processPositiveInteger`?
6. Trace the execution of method `getIntValue` if the following data items are entered by a careless user. What would be displayed?
```
ace
7.5
-5
```
7. Trace the execution of method `main` preceding the exercises if the data items in Question 6 were entered. What would be displayed?

1.7 Packages and Visibility

Packages

You have already seen packages. The Java API is organized into packages such as `java.lang`, `java.util`, `java.io`, and `javax.swing`. The package to which a class belongs is declared by the first statement in the file in which the class is defined using the keyword **`package`**, followed by the package name. For example, we could begin each class in the computer hierarchy (class `Notebook` and class `Computer`) with the line:

```
package computers;
```

All classes in the same package are stored in the same directory or folder. The directory must have the same name as the package. All the classes in the folder must declare themselves to be in the package.

Classes that are not part of a package may access only public members (data fields or methods) of classes in the package. If the application class is not in the package, it must reference the classes by their complete names. The complete name of a class is `packageName.className`. However, if the package is imported by the application class, then the prefix `packageName.` is not required. For example, we can reference the constant `GREEN` in class `java.awt.Color` as `Color.GREEN` if we import package `java.awt`. Otherwise, we would need to use the complete name `java.awt.Color.GREEN`.

The No-Package-Declared Environment

So far we have not specified packages, yet objects of one class could communicate with objects of another class. How does this work? Just as there is a default visibility, there is a default package. Files that do not specify a package are considered part of the default package. Therefore, if you don't declare packages, all your classes belong to the same package (the default package).

SYNTAX Package Declaration

FORM:

`package` *packageName*;

EXAMPLE:

```
package computers;
```

INTERPRETATION:

This declaration appears as the first line of the file in which a class is defined. The class is now considered part of the package. This file must be contained in a folder with the same name as the package.

PROGRAM STYLE

When to Package Classes

The default package facility is intended for use during the early stages of implementing classes or for small prototype programs. If you are developing an application that has several classes that are part of a hierarchy of classes, you should declare them all to be in the same package. The package declaration will keep you from accidentally referring to classes by their short names in other classes that are outside the package. It will also restrict the visibility of protected members of a class to only its subclasses outside the package (and to other classes inside the package) as intended.

Package Visibility

So far, we have discussed three layers of visibility for classes and class members (data fields and methods): private, protected, and public. There is a fourth layer, called *package visibility*, that sits between private and protected. Classes, data fields, and methods with package visibility are accessible to all other methods of the same package but are not accessible to methods outside of the package. By contrast, classes, data fields, and methods that are declared protected are visible within subclasses that are declared outside the package, in addition to being visible to all members of the package.

We have used the visibility modifiers `private`, `public`, and `protected` to specify the visibility of a class member. If we do not use one of these visibility modifiers, then the class member has package visibility and it is visible in all classes of the same package, but not outside the package. Note that there is no visibility modifier `package`; package visibility is the default if no visibility modifier is specified.

Visibility Supports Encapsulation

The rules for visibility control how encapsulation occurs in a Java program. Table 1.6 summarizes the rules in order of decreasing protection. Note that private visibility is for members of a class that should not be accessible to anyone but the class, not even classes that extend it. Except for inner classes, it does not make sense for a class to be private. It would mean that no other class can use it.

Also, note that package visibility (the default if a visibility modifier is not given) allows the developer of a library to shield classes and class members from classes outside the package. Typically, such classes perform tasks required by the public classes within the package.

Use of protected visibility allows the package developer to give control to other programmers who want to extend classes in the package. Protected data fields are typically essential to an object. Similarly, protected methods are those that are essential to an extending class.

Table 1.6 shows that public classes and members are universally visible. Within a package, the public classes are those that are essential to communicating with objects outside the package.

TABLE 1.6
Summary of Kinds of Visibility

Visibility	Applied to Classes	Applied to Class Members
`private`	Applicable to inner classes. Accessible only to members of the class in which it is declared	Visible only within this class
Default or package	Visible to classes in this package	Visible to classes in this package
`protected`	Applicable to inner classes. Visible to classes in this package and to classes outside the package that extend the class in which it is declared	Visible to classes in this package and to classes outside the package that extend this class
`public`	Visible to all classes	Visible to all classes. The class defining the member must also be public

PITFALL

Protected Visibility Can Be Equivalent to Public Visibility

The intention of protected visibility is to enable a subclass to access a member (data field or method) of a superclass directly. However, protected members can also be accessed within any class that is in the same package. This is not a problem if the class with the protected members is declared to be in a package; however, if it is not, then it is in the default package. Protected members of a class in the default package are visible in all other classes you have defined that are not part of an actual package. This is generally not a desirable situation. You can avoid this dilemma by using protected visibility only with members of classes that are in explicitly declared packages. In all other classes, use either public or private visibility because protected visibility is virtually equivalent to public visibility.

EXERCISES FOR SECTION 1.7

SELF-CHECK

1. Consider the following declarations:

```
package pack1;
public class Class1 {
    private int v1;
    protected int v2;
    int v3;
    public int v4;
}

package pack1;
public class Class2 {...}

package pack2;
public class Class3 extends pack1.Class1 {...}

package pack2;
public class Class4 {...}
```

   a. What visibility must variables declared in `pack1.Class1` have in order to be visible in `pack1.Class2`?
   b. What visibility must variables declared in `pack1.Class1` have in order to be visible in `pack2.Class3`?
   c. What visibility must variables declared in `pack1.Class1` have in order to be visible in `pack2.Class4`?

1.8 A Shape Class Hierarchy

In this section, we provide a case study that illustrates some of the principles in this chapter. For each case study, we will begin with a statement of the problem (Problem). Then we analyze the problem to determine exactly what is expected and to develop an initial strategy for solution (Analysis). Next, we design a solution to the problem, developing and refining an algorithm (Design). We write one or more Java classes that contain methods for the algorithm steps (Implementation). Finally, we provide a strategy for testing the completed classes and discuss special cases that should be investigated (Testing). We often provide a separate class that does the testing.

CASE STUDY Processing Geometric Figures

Problem We would like to process some standard geometric shapes. Each `figure` object will be one of three standard shapes (rectangle, circle, and right triangle). We would like to be able to do standard computations, such as finding the area and perimeter, for any of these shapes.

Analysis For each of the geometric shapes we can process, we need a class that represents the shape and knows how to perform the standard computations on it (i.e., find its area and perimeter). These classes will be `Rectangle`, `Circle`, and `RtTriangle`. To ensure that these shape classes all define the required computational methods (finding area and perimeter), we will make them abstract methods in the base class for the shape hierarchy. If a shape class does not have the required methods, we will get a syntax error when we attempt to compile it.

Figure 1.10 shows the class hierarchy. We used abstract class `Shape` as the base class of the hierarchy. We didn't consider using an actual class because there are no actual objects of the base class type. The single data field `shapeName` stores the kind of shape object as a `String`.

Design We will discuss the design of the `Rectangle` class here. The design of the other classes is similar and is left as an exercise. Table 1.7 shows class `Rectangle`. Class `Rectangle` has data fields `width` and `height`. It has methods to compute area and perimeter, a method to read in the attributes of a rectangular object (`readShapeData`), and a `toString` method.

FIGURE 1.10
Abstract Class Shape and Its Three Actual Subclasses

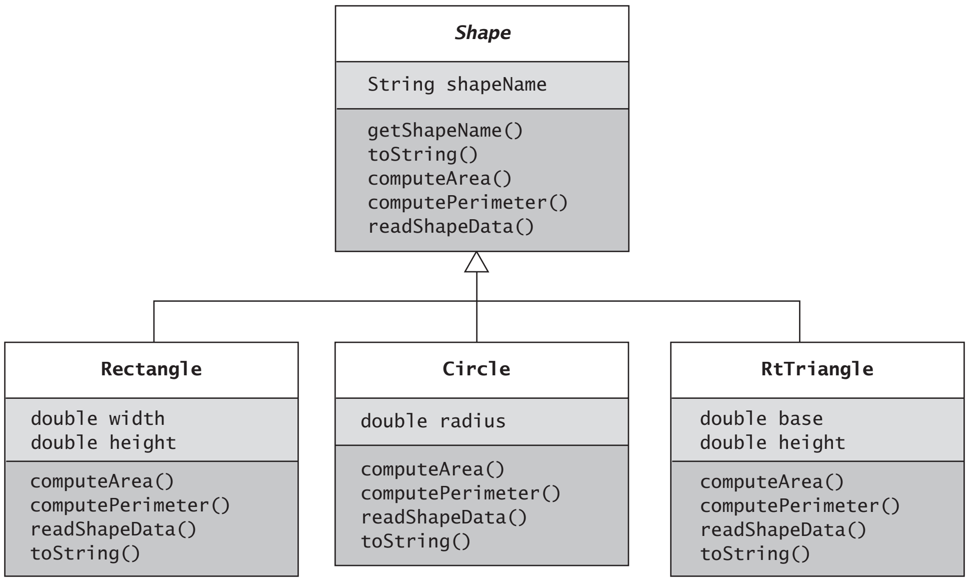


TABLE 1.7
Class Rectangle

Data Field	Attribute
double width	Width of a rectangle
double height	Height of a rectangle
Method	**Behavior**
double computeArea()	Computes the rectangle area (width × height)
double computePerimeter()	Computes the rectangle perimeter (2 × width + 2 × height)
void readShapeData()	Reads the width and height
String toString()	Returns a string representing the state

Implementation Listing 1.6 shows abstract class Shape.

LISTING 1.6
Abstract Class Shape (Shape.java)

```java
/** Abstract class for a geometric shape. */
public abstract class Shape {

    /** The name of the shape */
    private String shapeName = "";
    /** Initializes the shapeName.
        @param shapeName the kind of shape
     */
    public Shape(String shapeName) {
        this.shapeName = shapeName;
    }

    /** Get the kind of shape.
        @return the shapeName
     */
    public String getShapeName() { return shapeName; }

    @Override
    public String toString() { return "Shape is a " + shapeName; }

    // abstract methods
    public abstract double computeArea();
    public abstract double computePerimeter();
    public abstract void readShapeData();
}
```

Listing 1.7 shows class Rectangle.

LISTING 1.7
Class Rectangle (Rectangle.java)

```java
import java.util.Scanner;

/** Represents a rectangle.
    Extends Shape.
 */
public class Rectangle extends Shape {

    // Data Fields
    /** The width of the rectangle. */
    private double width = 0;
    /** The height of the rectangle. */
    private double height = 0;

    // Constructors
    public Rectangle() {
        super("Rectangle");
    }

    /** Constructs a rectangle of the specified size.
        @param width the width
        @param height the height
     */
    public Rectangle(double width, double height) {
        super("Rectangle");
        this.width = width;
        this.height = height;
    }

    // Methods
    /** Get the width.
        @return The width
     */
    public double getWidth() {
        return width;
    }

    /** Get the height.
        @return The height
     */
    public double getHeight() {
        return height;
    }

    /** Compute the area.
        @return The area of the rectangle
     */
    @Override
    public double computeArea() {
        return height * width;
    }

    /** Compute the perimeter.
        @return The perimeter of the rectangle
     */
    @Override
    public double computePerimeter() {
        return 2 * (height + width);
    }

    /** Read the attributes of the rectangle. */
    @Override
    public void readShapeData() {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter the width of the Rectangle");
        width = in.nextDouble();
        System.out.println("Enter the height of the Rectangle");
        height = in.nextDouble();
    }

    /** Create a string representation of the rectangle.
        @return A string representation of the rectangle
     */
    @Override
    public String toString() {
        return super.toString() + ": width is " + width + ", height is " +
               height;
    }
}
```

Testing To test the shape hierarchy, we will write a program that will prompt for the kind of figure, read the parameters for that figure, and display the results. The code for `ComputeAreaAndPerimeter` is shown in Listing 1.8. The `main` method is very straightforward, and so is `displayResult`. The `main` method first calls `getShape`, which displays a list of available shapes and prompts the user for the choice. The reply is expected to be a single character. The nested `if` statement determines which shape instance to return. For example, if the user's choice is C (for Circle), the statement

```
return new Circle();
```

returns a reference to a new `Circle` object.

After the new shape instance is returned to `myShape in main`, the statement

```
myShape.readShapeData();
```

uses polymorphism to invoke the correct member function `readShapeData` to read the shape object's parameter(s). The methods `computeArea` and `computePerimeter` are then called to obtain the values of the area and perimeter. Finally, `displayResult` is called to display the result.

A sample of the output from `ComputeAreaAndPerimeter` follows.

```
Enter C for circle
Enter R for Rectangle
Enter T for Right Triangle
R
Enter the width of the Rectangle
120
Enter the height of the Rectangle
200
Shape is a Rectangle: width is 120.0, height is 200.0
The area is 24000.00
The perimeter is 640.00
```

LISTING 1.8

ComputeAreaAndPerimeter.java

```
import java.util.Scanner;

/**
   Computes the area and perimeter of selected figures.
   @author Koffman and Wolfgang
 */
public class ComputeAreaAndPerimeter {

    /** The main program performs the following steps.
        1. It asks the user for the type of figure.
        2. It asks the user for the characteristics of that figure.
        3. It computes the perimeter.
        4. It computes the area.
        5. It displays the result.
        @param args The command line arguments -- not used
     */
    public static void main(String args[]) {
        Shape myShape;
        double perimeter;
        double area;
        myShape = getShape();    // Ask for figure type
        myShape.readShapeData(); // Read the shape data
        perimeter = myShape.computePerimeter();  // Compute perimeter
        area = myShape.computeArea();            // Compute the area
        displayResult(myShape, area, perimeter); // Display the result
        System.exit(0);                          // Exit the program
  }

  /** Ask the user for the type of figure.
       @return An instance of the selected shape
   */
    public static Shape getShape() {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter C for circle");
        System.out.println("Enter R for Rectangle");
        System.out.println("Enter T for Right Triangle");
        String figType = in.next();
        if (figType.equalsIgnoreCase("c")) {
          return new Circle();
        }
        else if (figType.equalsIgnoreCase("r")) {
             return new Rectangle();
        }
        else if (figType.equalsIgnoreCase("t")) {
             return new RtTriangle();
        }
        else {
             return null;
        }
    }
    /** Display the result of the computation.
             @param area The area of the figure
             @param perim The perimeter of the figure
      */
      private static void displayResult(Shape myShape, double area, double perim) {
          System.out.println(myShape);
          System.out.printf("The area is %.2f%nThe perimeter is %.2f%n",
                            area, perim);
    }
}
```

PROGRAM STYLE

Using Factory Methods to Return Objects

The method `getShape` is an example of a *factory method* because it creates a new object and returns a reference to it. The author of the `main` method does not need to know what kinds of shapes are available. Knowledge of the available shapes is confined to the `getShape` method. This function must present a list of available shapes to the user and decode the user's response to return an instance of the desired shape. If you add a new geometric shape class to the class hierarchy, you only need to modify the `if` statement in the factory method so that it can create and return an object of that type.

EXERCISES FOR SECTION 1.8

SELF-CHECK

1. Explain why `Shape` cannot be an actual class.
2. Explain why `Shape` cannot be an interface.

PROGRAMMING

1. Write class `Circle`.
2. Write class `RtTriangle`.

Chapter Review

- Inheritance and class hierarchies enable you to capture the idea that one thing may be a refinement or an extension of another. For example, a plant is a living thing. Such *is-a* relationships create the right balance between too much and too little structure. Think of inheritance as a means of creating a refinement of an abstraction. The entities farther down the hierarchy are more complex and less general than those higher up. The entities farther down the hierarchy may inherit data members (attributes) and methods from those farther up, but not vice versa. A class that inherits from another class **`extends`** that class.
- Encapsulation and inheritance impose structure on object abstractions. Polymorphism provides a degree of flexibility in defining methods. It loosens the structure a bit in order to make methods more accessible and useful. *Polymorphism* means "many forms." It captures the idea that methods may take on a variety of forms to suit different purposes.
- All exceptions in the `Exception` class hierarchy are derived from a common superclass called `Throwable`. This class provides methods for collecting and reporting the state of the program when an exception is thrown. The commonly used methods are `getMessage` and `toString`, which return a detail message describing what caused the exception to be thrown, and `printStackTrace`, which prints the exception and then shows the line where the exception occurred and the sequence of method calls leading to the exception.
- There are two categories of exceptions: checked and unchecked. Checked exceptions are generally due to an error condition external to the program. Unchecked exceptions are generally due to a programmer error or a dire event.
- The keyword **`interface`** defines an interface. A Java interface can be used to specify an abstract data type (ADT), and a Java class can be used to implement an ADT. A class that implements an interface must define the methods that the interface declares.
- The keyword **`abstract`** defines an abstract class or method. An abstract class is like an interface in that it leaves method implementations up to subclasses, but it can also have data fields and actual methods. You use an abstract class as the superclass for a group of classes in a hierarchy.
- Visibility is influenced by the package in which a class is declared. You assign classes to a package by including the statement **`package`** *packageName*; at the top of the file. You can refer to classes within a package by their direct names when the package is imported through an import declaration.

Java Constructs Introduced in This Chapter

```
abstract              private               super(...)
extends               protected             this.
instanceof            public                this(...)
interface             super.
package
```

Java API Classes Introduced in This Chapter

```
java.lang.Byte               java.lang.Number
java.lang.Float              java.lang.Object
java.lang.Integer            java.lang.Short
```

User-Defined Interfaces and Classes in This Chapter

`ComputeAreaAndPerimeter`	`Food`	`Student`
`Computer`	`Notebook`	`StudentInt`
`Employee`	`Rectangle`	`StudentWorker`
`EmployeeInt`	`Shape`	

Quick-Check Exercises

1. What does *polymorphism* mean, and how is it used in Java? What is method overriding? Method overloading?
2. What is a method signature? Describe how it is used in method overloading.
3. Describe the use of the keywords **`super`** and **`this`**.
4. Indicate whether each error or exception in the following list is checked or unchecked: `IOException`, `EOFException`, `VirtualMachineError`, `IndexOutOfBoundsException`, `OutOfMemoryError`.
5. When would you use an abstract class, and what should it contain?
6. An ______ specifies the requirements of an ADT as a contract between the ______ and ______; a ______ implements the ADT.
7. An interface can be implemented by multiple classes. (True/False)
8. Describe the difference between *is-a* and *has-a* relationships.
9. Which can have more data fields and methods: the superclass or the subclass?
10. You can reference an object of a ______ type through a variable of a ______ type.
11. You cast an object referenced by a ______ type to an object of a ______ type in order to apply methods of the ______ type to the object. This is called a ______.
12. The four kinds of visibility in order of decreasing visibility are ______, ______, ______, and ______.

Review Questions

1. Which method is invoked in a particular class when a method definition is overridden in several classes that are part of an inheritance hierarchy? Answer the question for the case in which the class has a definition for the method and also for the case where it doesn't.
2. Explain how assignments can be made within a class hierarchy and the role of casting in a class hierarchy. What is strong typing? Why is it an important language feature?
3. If Java encounters a method call of the following form:

   ```
   superclassVar.methodName()
   ```

   where `superclassVar` is a variable of a superclass that references an object whose type is a subclass, what is necessary for this statement to compile? During run time, will method `methodName` from the class that is the type of `superclassVar` always be invoked, or is it possible that a different method `methodName` will be invoked? Explain your answer.
4. Assume the situation in Question 3, but method `methodName` is not defined in the class that is the type of `superclassVar`, although it is defined in the subclass type. Rewrite the method call so that it will compile.
5. Explain the process of initializing an object that is a subclass type in the subclass constructor. What part of the object must be initialized first? How is this done?
6. What is default or package visibility?
7. Indicate what kind of exception each of the following errors would cause. Indicate whether each error is a checked or an unchecked exception.
   a. Attempting to create a `Scanner` for a file that does not exist
   b. Attempting to call a method on a variable that has not been initialized
   c. Using −1 as an array index

8. Discuss when abstract classes are used. How do they differ from actual classes and from interfaces?
9. What is the advantage of specifying an ADT as an interface instead of just going ahead and implementing it as a class?
10. Define an interface to specify an ADT `Money` that has methods for arithmetic operations (addition, subtraction, multiplication, and division) on real numbers having exactly two digits to the right of the decimal point, as well as methods for representing a `Money` object as a string and as a real number. Also, include methods `equals` and `compareTo` for this ADT.
11. Answer Review Question 10 for an ADT `Complex` that has methods for arithmetic operations on a complex number (a number with a real and an imaginary part). Assume that the same operations (+, –, *, /) are supported. Also, provide methods `toString`, `equals`, and `compareTo` for the ADT `Complex`.
12. Like a rectangle, a parallelogram has opposite sides that are parallel, but it has a corner angle, theta, that is less than 90 degrees. Discuss how you would add parallelograms to the class hierarchy for geometric shapes (see Figure 1.10). Write a definition for class `Parallelogram`.

Programming Projects

1. A veterinary office wants to store information regarding the kinds of animals it treats. Data includes diet, whether the animal is nocturnal, whether its bite is poisonous (as for some snakes), whether it flies, and so on. Use a superclass `Pet` with abstract methods and create appropriate subclasses to support about 10 animals of your choice.
2. A student is a person, and so is an employee. Create a class `Person` that has the data attributes common to both students and employees (name, social security number, age, gender, address, and telephone number) and appropriate method definitions. A student has a grade-point average (GPA), major, and year of graduation. An employee has a department, job title, and year of hire. In addition, there are hourly employees (hourly rate, hours worked, and union dues) and salaried employees (annual salary). Define a class hierarchy and write an application class that you can use to first store the data for an array of people and then display that information in a meaningful way.
3. Create a pricing system for a company that makes individualized computers, such as you might see on a Web site. There are two kinds of computers: notebooks and desktop computers. The customer can select the processor speed, the amount of memory, and the size of the disk drive. The customer can also choose a CD drive (CD ROM, CD-RW), a DVD drive, or both. For notebooks, there is a choice of screen size. Other options are a modem, a network card, or a wireless network. You should have an abstract class `Computer` and subclasses `DeskTop` and `Notebook`. Each subclass should have methods for calculating the price of a computer, given the base price plus the cost of the different options. You should have methods for calculating memory price, hard drive price, and so on. There should be a method to calculate shipping cost.
4. Write a banking program that simulates the operation of your local bank. You should declare the following collection of classes.

   Class `Account`

   Data fields: `customer` (type `Customer`), `balance`, `accountNumber`, `transactions` array (type `Transaction[]`). Allocate an initial `Transaction` array of a reasonable size (e.g., 20) and provide a `reallocate` method that doubles the size of the `Transaction` array when it becomes full.

   Methods: `getBalance`, `getCustomer`, `toString`, `setCustomer`

   Class `SavingsAccount` extends `Account`

   Methods: `deposit`, `withdraw`, `addInterest`

   Class `CheckingAccount` extends `Account`

   Methods: `deposit`, `withdraw`, `addInterest`

Class `Customer`

Data fields: `name`, `address`, `age`, `telephoneNumber`, `customerNumber`

Methods: Accessors and modifiers for data fields plus the additional abstract methods `getSavingsInterest`, `getCheckInterest`, and `getCheckCharge`.

Classes `Senior`, `Adult`, `Student`, all these classes extend `Customer`

Each has constant data fields `SAVINGS_INTEREST`, `CHECK_INTEREST`, `CHECK_CHARGE`, and `OVERDRAFT_PENALTY` that define these values for customers of that type, and each class implements the corresponding accessors.

Class `Bank`

Data field: `accounts` array (type `Account[]`). Allocate an array of a reasonable size (e.g., 100) and provide a reallocate method.

Methods: `addAccount`, `makeDeposit`, `makeWithdrawal`, `getAccount`

Class `Transaction`

Data fields: `customerNumber`, `transactionType`, `amount`, `date`, and `fees` (a string describing unusual fees)

Methods: `processTran`

You need to write all these classes and an application class that interacts with the user. In the application, you should first open several accounts and then enter several transactions.

5. You have a sizable collection of music and videos and want to develop a database for storing and processing information about this collection. You need to develop a class hierarchy for your media collection that will be helpful in designing the database. Try the class hierarchy shown in Figure 1.11, where Audio and Video are media categories. Then CDs and cassette tapes would be subclasses of `Audio`, and DVDs and VHS tapes would be subclasses of `Video`.

   If you go to the video store to get a movie, you can rent or purchase only movies that are recorded on VHS tapes or DVDs. For this reason, class `Video` (and also classes `Media` and `Audio`) should be abstract classes because there are no actual objects of these types. However, they are useful classes to help define the hierarchy.

   Class `Media` should have data fields and methods common to all classes in the hierarchy. Every media object has a title, major artist, distributor, playing time, price, and so on. Class `Video` should have additional data fields for information describing movies recorded on DVDs and videotapes. This would include information about the supporting actors, the producer, the director, and the movie's rating. Class DVD would have specific information about DVD movies only, such as the format of the picture and special features on the disk. Figure 1.12 shows a possible class diagram for `Media`, `Video`, and subclasses of `Video`.

FIGURE 1.11
Media Class Hierarchy

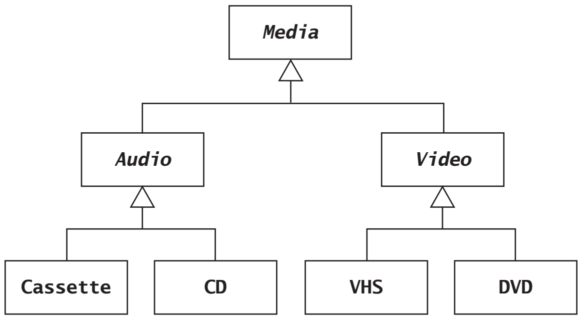
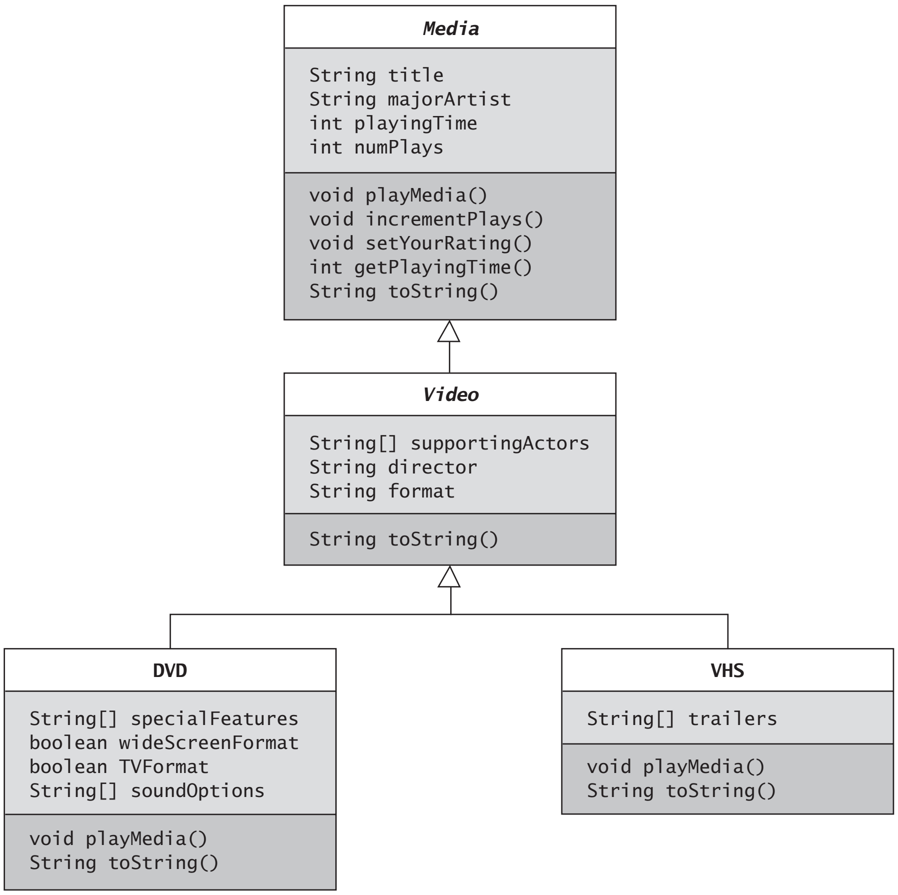


FIGURE 1.12
Class `Video` and Its Supers and Subclasses



Provide methods to load the media collection from a file and write it back out to a file. Also, provide a method to retrieve the information for a particular item identified by its title and a method to retrieve all your items for a particular artist.

6. Add shape classes `Square` and `EquilateralTriangle` to the figures hierarchy in Section 1.7. Modify class `ComputeAreaAndPerim` (Listing 1.8) to accept the new figures.
7. Complete the `Food` class hierarchy in Section 1.4. Read and store a list of your favorite foods. Show the total calories for these foods and the overall percentages of fat, protein, and carbohydrates for this list. To find the overall percentage, if an item has 200 calories and 10 percent is fat calories, then that item contributes 20 fat calories. You need to find the totals for fat calories, protein calories, and carbohydrate calories and then calculate the percentages.
8. A hospital has different kinds of patients who require different procedures for billing and approval of procedures. Some patients have insurance and some do not. Of the insured patients, some are on Medicare, some are in HMOs, and some have other health insurance plans. Develop a collection of classes to model these different kinds of patients.
9. A company has two different kinds of employees: professional and nonprofessional. Generally, professional employees have a monthly salary, whereas nonprofessional employees are paid an hourly rate. Similarly, professional employees have a certain number of days of vacation, whereas nonprofessional employees receive vacation hours based on the number of hours they have worked. The amount contributed for health insurance is also different for each kind of employee. Use an abstract class `Employee` to store information common to all employees and to declare methods for calculating weekly salary and computing health care contributions and vacation days earned that week. Define subclasses `Professional` and `Nonprofessional`. Test your class hierarchy.

10. Implement class `AMTbandkAmerica` in Section 1.1.
11. For the shape class hierarchy discussed in Section 1.8, consider adding classes `DrawableRectangle`, `DrawableCircle`, and so on that would have additional data fields and methods that would enable a shape to be drawn on a monitor. Provide an interface `DrawableInt` that specifies the methods required for drawing a shape. Class `DrawableRectangle`, for example, should extend `Rectangle` and implement this interface. Draw the new class hierarchy and write the new interface and classes. Using the Java Abstract Window Toolkit (AWT) to draw objects is described in Appendix Section C.7.

Answers to Quick-Check Exercises

1. *Polymorphism* means "many forms." Method overriding means that the same method appears in a subclass and a superclass. Method overloading means that the same method appears with different signatures in the same class.
2. A signature is the form of a method determined by its name and arguments. For example, doIt(int, double) is the signature for a method doIt that has one type int parameter and one type double parameter. If several methods in a class have the same name (method overloading), Java applies the one with the same signature as the method call.
3. The keyword **`this`** followed by a dot and a name means use the named member (data field or method) of the object to which the current method is applied rather than the member with the same name declared locally in the method. The keyword **`super.`** means use the method (or data field) with this name defined in the superclass of the object, not the one belonging to the object. Using **`super(...)`** as a method call in a constructor tells Java to call a constructor for the superclass of the object being created. Similarly, using **`this(...)`** as a method call in a constructor tells Java to call another constructor for the same class but with a different parameter list. The **`super(...)`** or **`this(...)`** call must be the first statement in a subclass constructor.
4. `VirtualMachineError`, `OutOfMemoryError`, and `IndexOutOfBoundsException` are unchecked; the rest are checked.
5. An abstract class is used as a parent class for a collection of related subclasses. An abstract class cannot be instantiated. The abstract methods (identified by modifier `abstract`) defined in the abstract class act as placeholders for the actual methods. Also, you should define data fields that are common to all the subclasses in the abstract class. An abstract class can have actual methods as well as abstract methods.
6. An interface specifies the requirements of an ADT as a contract between the developer and the user; a class implements the ADT.
7. True.
8. An *is-a* relationship between classes means that one class is a subclass of a parent class. A *has-a* relationship means that one class has data members of the other class type.
9. Subclass.
10. You can reference an object of a subclass type through a variable of a superclass type.
11. You cast an object referenced by a superclass type to an object of a subclass type in order to apply methods of the subclass type to the object. This is called a *downcast*.
12. The four kinds of visibility in order of decreasing visibility are *public*, *protected*, *package*, and *private*.