CHAPTER 1

Important Features of Java

This is a book about programming: specifically, about understanding, using, and implementing data structures and algorithms. The Java Collections Framework—part of the package java.util—has implementations of a considerable number of data structures and algorithms. Subsequent chapters will focus on what the framework is and how to use the framework in your programs. For this information to make sense to you, you will need to be familiar with certain aspects of Java that we present in this chapter. Some of what follows may be a review for you; some may be brand new. All the material is needed, either for the framework itself or to enable you to use the framework in your programming projects. ■

CHAPTER OBJECTIVES

1. Review the fundamentals of classes, objects, and messages.
2. Compare a developer's view of a class with a user's view of that class.
3. Follow the sequence of events related to maintaining a graphical user interface (GUI) window.
4. Understand how polymorphic references can be created through inheritance.
5. Be able to create try blocks and catch blocks to handle exceptions.

1.1 | CLASSES

A class combines variables with methods that act on those variables.

A ***class*** consists of variables called ***fields*** together with methods that operate on those fields. A class encapsulates the passive components (fields) and active components (methods) into a single entity. This encapsulation increases program modularity: by isolating a class from the rest of the program, we make the program easier to understand and to modify.

Suppose that in trying to solve some problem, we decide that we need to work with calendar dates. We will create a class called CalendarDate. (This has nothing to do with either the Calendar or Date classes in the package java.util.) The class CalendarDate will consist of one or more fields to hold a date and of methods that act on those fields. Initially, we need not worry about choosing the fields that will represent a date. Since we, and maybe others after us, will be using CalendarDate, we need to determine the *responsibilities* of CalendarDate. That is, what is the class expected to provide to its users? Perhaps initially the only responsibilities are to

1. Construct a date, given a month, day, and year
2. Determine if a given date is valid
3. Return the next date after a given date
4. Return the date previous to the given date
5. Return the day of the week (such as Tuesday) on which a given date occurs
6. Determine if a given date is prior to some other date

1.1.1 Method Descriptions

A method description provides everything a user needs to know about a method.

A class's responsibilities are refined into ***method descriptions:*** the explicit information a user will need in order to invoke the method. Each method description will have three parts: a precondition, a postcondition, and a method heading followed by a semicolon. A ***precondition*** is an assumption about the state of the program just prior to the execution of the method. A ***postcondition*** is a claim about the state of the program just after the execution of the method, provided the precondition was true beforehand. The precondition and postcondition are stated in terms of the calling object and formal parameters.

For example, here is the method description for the isValid( ) method:

```
// Postcondition:  true has been returned if the date is legal: the year must be
//                 an integer between 1800 and 2200, inclusive; the month
//                 must be an integer between 1 and 12, inclusive; the day
//                 must be an integer between 1 and the maximum number of
//                 days for the given month and year, inclusive. Otherwise,
//                 false has been returned.
public boolean isValid( );
```

There is no precondition given because nothing special is assumed about the state of the program prior to a call to the method. Technically, the precondition is simply

true. But we will omit writing the precondition when that occurs. Every method should accomplish something, so the postcondition will always be given explicitly. In the postcondition, "the date" refers to the calling object, that is, the object that invoked this method. In the method heading, the value returned is of type boolean, the method identifier is isValid( ) and there are no formal parameters.

Given a class, an ***object***—sometimes called an ***instance*** of the class—is a variable that has the fields of that class and can call the methods of that class. In Java, objects are always accessed indirectly, through references. For example, if we define

```
CalendarDate thisDate;
```

then thisDate is a reference to an object of type CalendarDate. If we later write

```
if (thisDate.isValid( ))
    System.out.println ("The date is valid.");
else
    System.out.println ("The date is not valid.");
```

then the isValid( ) method is being invoked by the object referenced by thisDate. The output will be determined by the boolean value returned by that isValid( ) method. In general, the syntax for a method invocation consists of an object reference followed by a dot followed by the method identifier followed by a parenthesized argument list. In object-oriented parlance, a ***message*** is the invocation of a method by an object. For example, the following message returns the date that immediately follows the calling object's date:

```
thisDate.next( )
```

In this message, the next( ) method in the CalendarDate class is being invoked by the object referenced by thisDate. The term *message* is meant to suggest that a communication is being sent from one part of a program to another part. For example, the message thisDate.next( ) may be sent from a method in some class other than the CalendarDate class.

When we refine the responsibilities for CalendarDate's methods, we get the following method descriptions:

```
// Postcondition:  this CalendarDate has been constructed from year, month
//                 and day.
public CalendarDate (int year, int month, int day);

// Postcondition:  true has been returned if the date is legal: the year must be
//                 an integer between 1800 and 2200, inclusive; the month
//                 must be an integer between 1 and 12, inclusive; the day
//                 must be an integer between 1 and the maximum number of
//                 days for the given month and year, inclusive. Otherwise,
//                 false has been returned.
public boolean isValid( );
```

Example Suppose currentDate is a reference to an object in the class CalendarDate. If the values of the fields in that object represent February 29, 2000, then currentDate.isValid( ) would return true. But if the values of the fields in currentDate represent February 29, 2001, then currentDate.isValid( ) would return false.

```
// Precondition: The date is valid.
// Postcondition: The next date has been returned.
public CalendarDate next( );
```

Example Suppose date is a reference to an object in the class CalendarDate. If the values of that object's fields represent February 29, 2000, then the message date.next( ) would return a reference to a date representing March 1, 2000. What would happen if the object referenced by date had an invalid value and date.next( ) were called? Because an invalid date would not satisfy the precondition, the result would be undefined. That is, there may be no date returned, a nonsense date returned, a program crash, etc. The user of a class has the responsibility of making sure a method's precondition is satisfied before calling the method. For example, a user could proceed as follows:

```
if (date.isValid( ))
      . . . date.next( ) . . .
else
      System.out.println ("Invalid date");
```

Additionally, the developer of a class is responsible for making sure the method descriptions provide sufficient information for a user of the class.

```
// Precondition: The date is valid.
// Postcondition: The previous date has been returned.
public CalendarDate previous( );

// Precondition: The date is valid.
// Postcondition: The day of the week—"Sunday", "Monday", and so on—on
//                which the date falls has been returned.
public String dayOfWeek( );

// Precondition: The calling object and otherDate are valid dates.
// Postcondition: true has been returned if the calling object's date precedes
//                otherDate's date. Otherwise, false has been returned.
public boolean isPriorTo (Date otherDate);
```

Example Suppose that currentDate is a reference to a CalendarDate object whose field values represent March 27, 2003, and that startDate

is a reference to a CalendarDate object whose fields represent January 1, 2004. Then

```
currentDate.isPriorTo (startDate)
```

would return true.

A complete CalendarDate class, along with a CalendarTester class to validate the above methods, is available in the chapters/ch1 directory. You can run the corresponding project to find out, for example, on what day of the week you were born.

This view of a class is the user's perspective, focusing on what information about the class is needed by users of the class. In the next section, we will look at the developer's perspective and compare the two perspectives.

1.1.2 Data Abstraction

Data abstraction is the separation of what a class provides to users from how the class is defined.

So far, we have concentrated on method descriptions, that is, *what* the class provides to users, rather than on the class's fields and method definitions, that is, *how* the class is defined. This separation—called ***data abstraction***—of *what* from *how* is an essential feature of object-oriented programming. Users of the CalendarDate class will not care about how a date is represented or how the methods are defined. The fields—that is, the representation of the date—may take one of the following forms, or may be something entirely different:

int fields month, day, and year	with values such as 2 for month, 28 for day, and 2002 for year
An int field myDate	with a value such as 02282002
A String field myDate	with a value such as "02282002"

Similarly, there may be a choice of method definitions for some of the methods. For example, there are several choices for the definition of the isValid( ) method. The following definition assumes int fields month, day, and year. Some of the work is delegated to daysInMonth( ), a helper[1] method.

```
public boolean isValid( ) {

    final int MAX_MONTH = 12;
    final int MIN_YEAR = 1800;
    final int MAX_YEAR = 2200;

    if (month < 1 || month > MAX_MONTH || day < 1 || year < MIN_YEAR ||
            year > MAX_YEAR)
        return false;
    return day <= daysInMonth( );

} // method isValid
```

[1]A helper method is not accessible by users of a class but performs some task for one or more methods in the class.

Here is the definition of the helper method daysInMonth:

```
// Precondition: 1 <= month <= 12; 1800 <= year <= 2200.
// Postcondition: the number of days in the given month and year has been
//                returned.

private int daysInMonth( ) {

    if (month == 4 || month == 6 || month == 9 || month == 11)
        return 30; // 30 days hath September, April, June and November

    else if (month == 1 || month == 3 || month == 5 || month == 7 ||
             month == 8 || month == 10 || month == 12)
        return 31; // January, March, May, July, August, October,
                   December

    else if(year % 4 != 0 || (year % 100 == 0 && year % 400 != 0))
        return 28; // one year is slightly less than 365.25 days
    return 29;

} // method daysInMonth
```

Such details would be of no help when you are trying to develop a class that *uses* CalendarDate. And it may be that someone else has already completed the definition of the CalendarDate class. Then you should use that CalendarDate class, rather than creating extra work for yourself. But even if you must define the CalendarDate class yourself, you can postpone that work until after you have completed the development of the classes that use the CalendarDate class. By working with CalendarDate's method descriptions, you increase the independence of those other classes: their effectiveness will not be affected by any changes to the CalendarDate class that do not affect the method descriptions.

When users focus on what a class provides rather than on the implementation details of that class, they are applying the principle of data abstraction:

Principle of Data Abstraction A user's code should not access the implementation details of the class used.

One important application of the principle of data abstraction is that if class A uses class B, then class A's methods should not access class B's fields. In fact, class B's fields should be accessed only in class B's methods. This turns out to be a benefit to users because their code will be unaffected if the developer of class B decides to replace the old fields with new ones. For example, suppose the following definition is made outside of the CalendarDate class:

```
CalendarDate currentDate;
```

Then an expression such as

```
currentDate.month
```

would be a violation of the principle of data abstraction because whether or not the CalendarDate class has a month field is an implementation detail. Even if the CalendarDate class currently has a month field, and even if that field has *public* visibility, the developer is free to make any changes to the CalendarDate class that do not affect the method descriptions. For example, the developer could have a single field:

```
String myDate;
```

We noted earlier that the principle of data abstraction is a benefit to users of a class because they are freed from reliance on implementation details of that class. This assumes, of course, that the class's method descriptions provide all the information that a user of that class needs. The developer of a class should create methods with sufficient functionality that users need not rely on any implementation details. That functionality should be clearly spelled out in the method descriptions.

The precondition and postcondition of a method are part of an implicit *contract* between the developer and the user. The terms of the contract are as follows:

> If the user of the method ensures that the precondition is true before the method is invoked, the developer guarantees that the postcondition will be true at the end of the execution of the method.

In the sections on visibility modifiers and exceptions, we will see how these two features of Java enable the developer of a class to force users of that class to invoke the methods properly.

We can summarize our discussion of classes so far by saying that from the *developer's* perspective, a class consists of fields and the definitions of methods that act on those fields. A *user's* view is an abstraction of this: A class consists of method descriptions.

The Java Collections Framework is, basically, a hierarchy of thoroughly tested classes that are useful in a variety of applications. The programs in this book will *use* the Java Collections Framework, so those programs will not rely on the definitions of the framework's methods. We will provide method descriptions and an overview of most of the classes. To give you experience in reading the code of professional programmers, you will also get to study the details of the method definitions.

The next section continues our discussion of method descriptions and method definitions with another example.

1.1.3 An Employee Class

For another example of a class, let's create a class called Employee for the employees in a company. The information available on each employee consists of the employee's name and gross pay. The responsibilities of the Employee class are to

1. Initialize an employee's name to a blank and gross pay to 0.00
2. Initialize an employee's name and gross pay from a String
3. Determine if the input sentinel has been reached (the sentinel has the name "***" and a gross pay of –1.00)

4. Determine if an employee's gross pay is greater than some other employee's gross pay
5. Convert an employee's name and gross pay to a String suitable for output.

The method descriptions are as follows:

```
// Postcondition: this employee's name has the value "" and gross pay has
//                the value 0.00.
public Employee( );
```

> ***Note*** In the above postcondition, and in the postconditions that follow, "employee" refers to the calling object.

```
// Precondition: s is not null.
// Postcondition: this employee has been initialized from s, which consists
//                of a name and gross pay, with at least one blank in between.
public Employee (String s);

// Postcondition: true has been returned if this employee is the input sentinel.
//                Otherwise, false has been returned.
public boolean isSentinel( );

// Postcondition: true has been returned if this employee's gross pay is
//                greater than that of otherEmployee. Otherwise, false has
//                been returned.
public boolean makesMoreThan (Employee otherEmployee);

// Postcondition: a String representation of this employee's name and gross
//                pay has been returned in the form <name $gross pay>.
public String toString( );
```

The Employee class's method descriptions are all that a user of the class will need. A developer of the class, on the other hand, must decide what fields to have and then define the methods. For example, a developer may well decide to have two fields: the employee's name (a String) and grossPay (a **double**). The complete method definitions might then be as given in the following:

```
import java.util.*;

class Employee {

    private static final String EMPTY_STRING = "";

    private static final String NAME_SENTINEL = "***";

    private static final double GROSS_PAY_SENTINEL = -1.00;

    private String name;
```

```
    private double grossPay;

    // Postcondition: this employee has been initialized: the name is an
    //                empty string and the gross pay is 0.00.
    public Employee(){
        name = EMPTY_STRING;
    } // default constructor

// Postcondition: this employee has been initialized from s, which consists of
//                a name and gross pay, with at least one blank in between.
public Employee (String s) {
    StringTokenizer tokens = new StringTokenizer (s);
    name = tokens.nextToken( );
    grossPay = Double.parseDouble (tokens.nextToken( ));
} // constructor with String parameter

// Postcondition: true has been returned if this employee is the sentinel.
//                Otherwise, false has been returned.
public boolean isSentinel( ) {
    if (name.equals (NAME_SENTINEL) && grossPay ==
            GROSS_PAY_SENTINEL)
        return true;
    return false;
} // method isSentinel

// Postcondition: if this employee's gross pay is larger than otherEmployee's
//                gross pay, true has been returned. Otherwise, false has
//                been returned.
public boolean makesMoreThan (Employee otherEmployee) {
    return grossPay > otherEmployee.grossPay;
} // method makesMoreThan

// Postcondition: a String representation of this employee's name and gross
//                pay has been returned in the form <name $gross pay>.
public String toString( ) {
    final String DOLLAR_SIGN = " $";
    return (name + DOLLAR_SIGN + grossPay);
} // method toString( )
} // class Employee
```

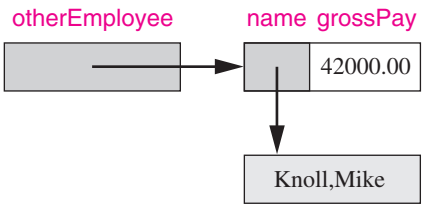
Now that we have a complete class to look at, we can consider some of the details of classes in general.

1.1.4 Local Variables and Fields

Variables declared within a method—including the method's formal parameters—are called ***local variables.*** A *field* is a variable that is declared inside a class but outside of the class's methods. Local variables can be accessed only within the method in which they are declared. For example, here is the definition of the makesMoreThan method of the Employee class:

```
public boolean makesMoreThan (Employee otherEmployee) {
    return grossPay > otherEmployee.grossPay;
} // method makesMoreThan
```

This method has one local variable: the formal parameter otherEmployee. The specified type is the class identifier Employee. In Java, when the specified type is a class, the variable is a *reference to an object in that class.* So otherEmployee is a reference to an object from the class Employee. That is, the variable otherEmployee will contain the address of an Employee object. For example, if that Employee object's name and gross pay were "Knoll,Mike" and 42000.00, we could draw the following:



In the above picture, we use an arrow from otherEmployee to the object referenced by otherEmployee. That referenced object has two fields: a reference to a String object, and a **double**.

1.1.5 Constructors

A ***constructor*** is a method that is invoked when an instance of a class is created. The purpose of a constructor is to initialize the object instantiated. A constructor has the same identifier as the class, and a constructor has no return type. For example, the Employee class above has two constructors. Their definitions are

```
// Postcondition: this employee has been initialized: the name is an
//                empty string and the gross pay is 0.00.
public Employee( ) {
    name = EMPTY_STRING;
} // default constructor
```

```
// Postcondition: this employee has been initialized from s, which
//                consists of a name and gross pay.
public Employee (String s) {

    StringTokenizer tokens = new StringTokenizer (s);
    name = tokens.nextToken( );
    grossPay = Double.parseDouble (tokens.nextToken( ));

} // constructor with String parameter
```

A default constructor is a constructor with no parameters.

The first of these two constructors has no parameters and is called the ***default constructor.*** Whenever a constructor is called, for example,

```
new Employee( );
```

all the class's fields are automatically initialized according to their types. In the case of Employee, the name field is initialized to a **null** reference and grossPay is initialized to 0.0. Similarly, fields of type **int** are initialized to 0, and fields of type **boolean** are initialized to **false**. The body of the constructor can then override those initializations. For example, the default constructor above assigns to name a reference to an empty String object—its value is "".

The constructor with a parameter has two local variables: s and tokens. That constructor overrides the default initializations for both name and grossPay.

If you neglect to include any constructors in a class, the Java compiler will automatically provide a default constructor, which performs the automatic field initializations just discussed. But if you include any constructor, the compiler will not provide a default constructor, and this omission can have an impact on inheritance, as discussed in the section on inheritance.

1.1.6 Instance Variables and Static Variables

In a class, the term ***member*** refers to either a field or method. Let's look at some of the different kinds of members. There are two kinds of fields. An ***instance variable*** is a field associated with an object—that is, with an instance of a class. For example, in the Employee class, name and grossPay are instance variables. Each Employee object will have its own pair of instance variables. If we declare

```
Employee oldEmployee,
         currentEmployee,
         newEmployee;
```

then the object referenced by oldEmployee will have its own copy of the instance variables name and grossPay, and the objects referenced by currentEmployee and newEmployee will have their own copies also.

In addition to instance variables, which are associated with a particular object in a class, we can declare ***static variables,*** which are associated with the class itself. The space for a static variable—also called a ***class variable***—is shared by all instances of the class. A variable is designated as a static variable by the modifier **static**, which is a reserved word. For example, if a count field is to maintain information about all

objects in a class Student, we could declare the field count to be a static variable in the Student class:

```
static int count;
```

Java also allows static methods. A static method is associated with the class itself. For example, every project must have a class with a static main method.

A static member, that is, a static variable or static method, is declared with the modifier **static**. To access a static member inside its class, the member identifier alone is sufficient. For example, the above static field count could be accessed in a method in the Student class as follows:

```
count++;
```

In order to access a static member *outside* of its class, the class identifier itself is used as the qualifier. For example, outside of the class Student, we could write:

```
if (Student.count == 0)
```

We will use the **static** modifier for any constant that is defined outside of a class's methods. The reason is that the constant cannot change for the individual objects in the class, so we might as well have just one copy of the constant for all objects, instead of one copy for each object. Here, from the above declaration of the Employee class, is an example of a static constant:

```
private static final double GROSS_PAY_SENTINEL = -1.00;
```

The **static** modifier is not available within a method. For example, in the above declaration of the toString( ) method in the Employee class, we had:

```
final String DOLLAR_SIGN = " $";
return (name + DOLLAR_SIGN + grossPay);
```

It would have been illegal to use the static modifier for this constant.

1.1.7 Visibility Modifiers

A ***visibility modifier*** is a reserved word that determines where a class or class member can be accessed. The two most important visibility modifiers are

private	Accessible only within the given class
public	Accessible anywhere

In Java, there can be at most one public class per file, that is, at most one class with the visibility modifier **public**. Also, the name of that public class must be the same as the name of the file—without the .java extension. At the beginning of the file, there must be import statements for any package (or file) needed by the file but not part of the project. An exception is made for java.lang.*, which is automatically imported for any file.

Typically, a class's methods will be **public** so they can be accessed outside of the class. Of course, that access will rely only on the methods' descriptions, not on

the methods' definitions. Occasionally, a "helper" method is used to simplify the work of another method. That helper method should not be accessed outside of the class, so its visibility modifier will be private.

In order to promote data abstraction, the developer of a class should not have public fields. For now, all fields will be declared as **private**. Later in this chapter, we will encounter another visibility modifier, **protected**, which represents a sort of compromise between **public** and **private**.

A member with no visibility modifier is said to have *default visibility*. A member with default visibility can be accessed by any object (or class, in the case of a static member) in the same package as the class in which the member is declared. That is why default visibility is sometimes referred to as "package-friendly visibility." Java has a "default" package consisting of all classes that are not part of any package but are either in the current directory or in a directory listed in the CLASSPATH variable. So, for example, classes in the same directory can access each other's default-visibility members.

1.1.8 Graphical User Interfaces

The Employee class does not include any input or output statements. Input and output of employee information are the responsibility of classes that use the Employee class. This separation—of the Employee class from input-output details—increases the flexibility of the Employee class because it can then be used in a variety of input-output environments, such as console input-output or file input-output.

In the next section, we will begin the development of a Company class that uses the Employee class. The Company class will handle the input and output of employee information. For input and output, we will utilize a graphical user interface (GUI). If you have not used a GUI before, you might want to invest some time in studying the sample GUI in Appendix 2. That is the same GUI we will be using for this application and, in fact, the same GUI we will be using for every other application! Here are the essential features of programs that use this GUI:

1. The program does not use the readLine method to read keyboard input or the System.out.println method for screen output.
2. Instead, the GUI class creates a window with one input line and any number of output lines.
3. The associated GUIListener class has an actionPerformed method that is automatically invoked when the user of the program presses the Enter key after entering a string in the input line. That actionPerformed method invokes the processInput method in the Company class.
4. The processInput method—with a String parameter—produces a name and gross pay from the String.
5. The GUI class has a print and a println method for output to the GUI-created window.
6. Neither the GUI class nor the GUIListener class is in java.util.

1.1.9 The Company Class

Suppose we want to determine who is the best-paid employee in a company. We will create a Company class that uses the Employee class. The responsibilities of the Company class are to initialize a Company and to process the input—determine if the current employee is the best-paid so far; if the current employee is the sentinel, the name and gross pay of the best-paid employee are printed. We refine these responsibilities into method descriptions for a constructor and processInput methods:

The processInput *method is executed every time a new input line is entered.*

```
// Postcondition: this company has been initialized.
public Company( );

// Postcondition: the input string s, consisting of the name and gross pay of
//                one employee in this company, has been processed against
//                what had been the best-paid employee in this company. If s
//                was the sentinel, the name and gross pay of the best-paid
//                employee have been printed.
public void processInput (String s);
```

The details of these Company methods are covered in Lab 1, which combines the Employee, Company, and GUI classes into a complete project.

LAB Lab 1: The CompanyMain Project. *All Labs Are Optional* LAB

As noted earlier, we should use existing classes whenever possible. What if a class has most, but not all, of what is needed for an application? We could simply scrap the existing class and develop our own, but that would be time-consuming and inefficient. Another option is to copy the needed parts of the existing class and incorporate those parts into a new class that we develop. The danger with that option is that those parts may be incorrect or inefficient. If the developer of the original class replaces the incorrect or inefficient code, our class would still be erroneous or inefficient. A better alternative is to use inheritance, which is explained in the next section.

1.1.10 Inheritance

We should write program components that are reusable. For example, instead of defining a method that calculates the average gross pay of 10 employees, it would have wider applicability to define a method that calculates the average gross pay of any number of employees. By writing reusable code, we not only save time, but we also avoid the risk of incorrectly modifying the existing code.

One way that reusability can be applied to classes is through a special and powerful property of classes: inheritance. ***Inheritance*** is the ability to define a new class that includes all the fields and some or all of the methods of an existing class. The

previously existing class is called the ***superclass*** or ***base class.*** The new class, which may declare new fields and methods, is called the ***subclass*** or ***derived class.*** A subclass may also *override* existing methods by giving them method definitions that differ from those in the superclass.

As an example of how inheritance works, let's start with the class Employee. Suppose that several applications use Employee. A new application involves finding the best-paid hourly employee. For this application, the input consists of the employee's name, hours worked (an **int**), and pay rate (a **double**). The gross pay is the hours worked times the pay rate.

We could alter Employee by adding hoursWorked and payRate fields and modifying the methods. But it is risky to modify, for the sake of a new application, a class that is being used successfully in existing applications. The underlying concept is known as the open-closed principle:

> **The Open-Closed Principle** Every class should be open (extendible through inheritance) and closed (stable for existing applications).

Instead of rewriting Employee, we will create HourlyEmployee, a subclass of Employee. To indicate that a class is a subclass of another class, the subclass identifier is immediately followed by the reserved word **extends**. For example, we can declare the HourlyEmployee class to be a subclass of Employee as follows:

```
class HourlyEmployee extends Employee {
. . .
```

Each HourlyEmployee object will have the information from Employee—name and gross pay—as well as hours worked and pay rate.

Some of Employee's methods—makesMoreThan and toString—are inherited as is by HourlyEmployee. The isSentinel method and the constructor that has a String parameter are overridden because the input line will contain a name, hours worked, and pay rate. For example, here is a definition of the constructor-with-parameter:

```
// Postcondition: this HourlyEmployee has been initialized from s, which
//                consists of a name, hours worked and pay rate, with at least
//                one blank in between each of those components.
public HourlyEmployee (String s) {

    StringTokenizer tokens = new StringTokenizer (s);

    name = tokens.nextToken( );
    hoursWorked = Integer.parseInt (tokens.nextToken( ));
    payRate = Double.parseDouble (tokens.nextToken( ));

    grossPay = hoursWorked * payRate;

} // constructor with parameter
```

1.1.11 The protected Visibility Modifier

Notice that in the above constructor for HourlyEmployee, the name and grossPay fields from the Employee class are treated as if they were declared as fields in the HourlyEmployee class. In order for this phenomenon to work, the visibility modifier for those fields in Employee is changed from private to protected:

```
protected String name;

protected double grossPay;
```

These declarations enable any subclass of Employee to access the name and grossPay fields as if they were declared within the subclass itself. This makes sense because an HourlyEmployee object *is an* Employee object as well. So the HourlyEmployee class actually has four fields: two inherited and two explicitly declared in HourlyEmployee.

The subclass HourlyEmployee automatically inherits all the fields, from Employee, that have the **protected** modifier. Later on, if a subclass of HourlyEmployee is created, we would want that subclass's methods to be able to inherit the HourlyEmployee fields—as well as the Employee fields. So the declarations of the HourlyEmployee fields hoursWorked and payRate also have the **protected** modifier:

```
protected int hoursWorked;

protected double payRate;
```

In general, if an identifier (for a field, constant or method) in a class has protected visibility, that identifier can be accessed in any class that is in the same package as the given class. So any class—whether or not a subclass—that is in the same package as Employee can access the name and grossPay fields of an Employee object. And all classes that are not in a specific package are considered to be part of the same default package.

Does protectedness transfer across packages? Yes, but only within a subclass, and only for objects whose type is that subclass. For a bare-bones illustration, suppose we have class A declared in package APackage:

```
package APackage;

public class A {

    protected int t;

} // class A
```

Also, suppose that classes C and D are subclasses of A and that C and D are in a different package from A. Then within class D, the t field is treated as if it were declared in D *instead of* in A. Here are possible declarations for classes C and D:

```
import APackage.*;

public class C extends A {}
```

Class D is declared in another file:

```
import APackage.*;
public class D extends A {

    public void meth( ) {

        D d = new D( );
        d.t = 1; // ok
        t = 2; // ok
        A a = new A( );
        a.t = 3; // illegal
        C c  new C( );
        c.t = 4; // illegal

    } method meth
} // class D
```

In D's method meth( ), the assignment to d.t is legal because t is treated as if it were declared in D instead of in A. For the very same reason, the assignment to the calling object's t field is legal in the statement:

```
t = 2;
```

And, also because t is treated as if it were declared in D, the assignments to a.t and c.t are illegal.

When you get to studying the Java Collections Framework, you will notice that many fields have default visibility—they do not have any visibility modifier. As noted above, fields with default visibility are "package friendly"; that is, they can be accessed in any class in the same package and cannot be accessed in any class in a different package. As a consequence, your projects will probably not be able to create subclasses of the classes in the Java Collections Framework.

With few exceptions, subclassing across package boundaries is discouraged in the Java Collections Framework. Why? The main reason is philosophical: a belief that the efficiency to users of the subclass is not worth the risk to the integrity of the subclass if the superclass is subsequently modified. This danger is not merely hypothetical. In Java 1.1, a class in java.security subclassed the Hashtable class. In Java 2, the Hashtable class was modified, and this opened a security hole in the subclass. The bottom line is that subclassing represents more of a commitment than mere use. So even if a class permits subclassing, it is not necessarily the wisest choice.

For the sake of completeness, here is the HourlyEmployee class:

```
public class HourlyEmployee extends Employee {

    protected static final int HOURS_WORKED_SENTINEL = −1;

    protected static final double PAY_RATE_SENTINEL = −1.00;

    protected int hoursWorked;
```

```
        protected double payRate;

        // Postcondition: this HourlyEmployee has been initialized.
        public HourlyEmployee( ) {}

        // Postcondition: this HourlyEmployee has been initialized from s,
        //                which contains a name, hours worked and pay rate,
        //                with at least one blank in between those components.
        public HourlyEmployee (String s) {
            StringTokenizer tokens = new StringTokenizer (s);

            name = tokens.nextToken( );
            hoursWorked = Integer.parseInt (tokens.nextToken( ));
            payRate = Double.parseDouble (tokens.nextToken( ));

            grossPay = hoursWorked * payRate;
    } // constructor

    // Postcondition:   true has been returned if the employee is the sentinel.
    //                  Otherwise, false has been returned.
    public boolean isSentinel( ) {

        if (name.equals (NAME_SENTINEL)
                && hoursWorked == HOURS_WORKED_SENTINEL
                && payRate == PAY_RATE_SENTINEL)
            return true;
        return false;
        } // method isSentinel

    } // class HourlyEmployee
```

The next section continues our discussion of inheritance by examining the interplay between inheritance and constructors.

1.1.12 Inheritance and Constructors

Constructors provide initialization for instances of a given class. For that reason, constructors are never inherited. But whenever a subclass constructor is called, the execution of the subclass constructor starts with an automatic call to the superclass's default constructor. This ensures that at least the default initialization of fields from the superclass will occur. For example, the Employee class's default constructor is automatically invoked at the beginning of any call to an HourlyEmployee constructor.

What if the superclass has a constructor but no default constructor? Then the first statement in any subclass constructor must explicitly call the superclass constructor. A call to a superclass constructor consists of the reserved word super followed by the argument list, in parentheses. For example, suppose some class B's

only constructor has an int parameter. If C is a subclass of B and has a constructor with a String parameter, that constructor must start out by invoking B's constructor:

```
public C (String s) {

    super (Integer.parseInt (s)); // explicitly calls B's int-parameter constructor
    . . .
```

So if a superclass explicitly defines a default (that is, zero-parameter) constructor, there are no restrictions on its subclasses. Similarly, if the superclass does not define any constructors, the compiler will automatically provide a default constructor, and there are no restrictions on the subclasses. But if a superclass defines at least one constructor and does not define a default constructor, the first statement in any subclass's constructor must explicitly invoke a superclass constructor.

Just as the Company class used Employee, the problem of finding the best-paid hourly employee requires that we create Company2, which uses the HourlyEmployee class. Company2, a subclass of the Company class described earlier, differs only slightly from the Company class. Most notably, the processInput method is overridden because the object defined in that class is

```
HourlyEmployee employee = new HourlyEmployee(s);
```

instead of

```
Employee employee = new Employee (s);
```

Here is the Company2 class:

```
public class Company2 extends Company {

    protected static final String SENTINEL_MESSAGE =
        "The sentinels are ***, -1 and -1.00.\n\n";

    protected static final String INPUT_PROMPT =
        "\nIn the Input line, please enter a name (with no blanks), " +
        "\nhours worked and pay rate, followed by the Enter key.";

    // Postcondition: this company has been initialized.
    public Company2( ) {
        gui.clear( );//make the output area blank.
        gui.println (SENTINEL_MESSAGE);
        gui.println (INPUT_PROMPT);

    } // default constructor

    // Precondition: s is not null.
    // Postcondition: the hourly-employee's information has been extracted
    //                from s and, if the sentinel, the best-paid hourly
    //                employee has been printed. Otherwise, that information
    //                has been used to, possibly, update the best-paid-so-far
```

```
//                  hourly-employee.
public void processInput (String s) {

        gui.println (s);
        HourlyEmployee employee = new HourlyEmployee (s);
        if (!employee.isSentinel( )) {

                atLeastOneEmployee = true;
                if (employee.makesMoreThan (bestPaid))
                        bestPaid = employee;
                gui.println (INPUT_PROMPT);

        } // not the sentinel
        else
                printBestPaid( );

} // processInput

} // class Company2
```

In the Company2 class, both the *constructor* and the processInput method have an interesting feature. Let's look at the constructor first. As noted earlier, whenever a subclass's constructor is called, the first statement executed is always a call to the superclass's default constructor. The Company class's default constructor initializes bestPaid, constructs a GUI window, and outputs a couple of lines to that window's output area. So the Company2 class's constructor needs to clear that output area and print the new prompt and sentinel messages. The clearing is accomplished in the clear( ) method, which sets the output area's text to blanks.

In the processInput method, the curious statement is the assignment statement:

```
bestPaid = employee;
```

It seems that the types do not agree: the type of bestPaid is reference-to-Employee, but the type of employee is reference-to-HourlyEmployee. Such an assignment is legal because an HourlyEmployee *is an* Employee. In general, we have the subclass substitution rule:

> **Subclass Substitution Rule** Whenever a reference-to-superclass-object is called for in an expression, a reference-to-subclass-object may be substituted.

For another example, suppose that class Y is a subclass of class X. The following is legal:

```
X x = new X( );
Y y = new Y( );
x = y;
```

In this last assignment, a reference-to-X is called for in the expression on the right-hand side, so a reference-to-Y may be substituted: a Y *is an* X.

But the reverse assignment is illegal:

```
X x = new X();
Y y = new Y();
y = x; // illegal
```

On the right-hand side of this last assignment, the compiler expects a reference-to-Y, so a reference-to-X is unacceptable: an X is not a Y. Note that the left-hand side of an assignment statement must consist of a variable, not an expression, so the subclass substitution rule does not apply to the left-hand side.

Now suppose we had the following:

```
X x = new X();
Y y = new Y();
x = y;
y = x;
```

After the assignment of y to x, x contains a reference to a Y object. But the assignment:

```
y = x;
```

still generates a compile-time error because the declared type of x is still reference-to-X. We can avoid a compile-time error in this situation with a *cast*: the temporary conversion of an expression's type to another type. The syntax for a cast is:

```
(the new type)expression
```

For example, we could cast x's type to Y as follows:

```
X x = new X();
Y y = new Y();
x = y;
y = (Y)x;
```

This last assignment passes muster with the compiler because the right-hand side now has type reference-to-Y. And there is no problem at run-time either because—from the previous assignment of y to x—the value on the right-hand side really is a reference-to-Y. But the following—acceptable to the compiler—generates a ClassCastException at run-time:

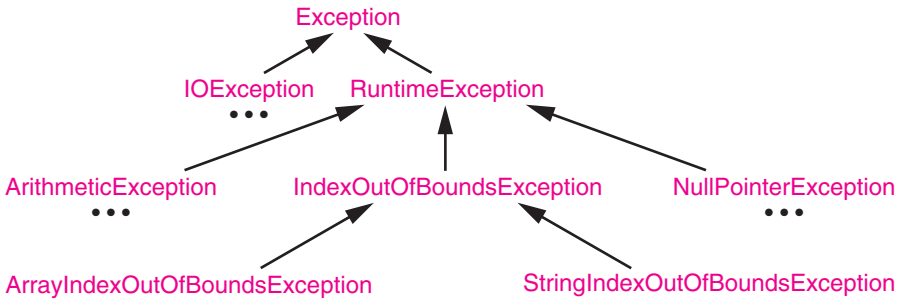
```
X x = new X();
Y y = new Y();
y = (Y)x;
```

The run-time problem is that x is actually pointing to an X object, not to a Y object.

The complete project, CompanyMain2, is in the chapters/ch1 directory. Lab 2's project illustrates another subclass of Employee.

LAB **Lab 2: The SalariedEmployee Class.** *All Labs Are Optional* LAB

In the above example, the superclass Employee has only one subclass, HourlyEmployee, and Company's only subclass is Company2. In some situations there may be an entire hierarchy of classes. For example, in Java, an ***exception*** is an object created by an unusual condition, typically, an attempt at invalid processing. Here is part of Java's exception hierarchy:



For example, if a program attempts to divide by zero, an ArithmeticException object will be constructed, and if you attempt to use an uninitialized object, a NullPointerException object will be constructed. Exception handling is discussed later in this chapter.

You will often encounter the following situation. You are developing a class B, and you realize that the methods of some other class, A, will be helpful. One possibility is for B to inherit all of A; that is, B will be a subclass of A. Then all of A's methods are available to B. An alternative is to define, in class B, a field whose class is A. Then the methods of A can be invoked by that field. It is important to grasp the distinction between these two ways to access the class A.

Inheritance describes an *is-a* relationship. An object of the subclass HourlyEmployee will be an object of the superclass Employee, so we can say that an HourlyEmployee is an Employee. An ArithmeticException is a RunTimeException. A NullPointerException is a RunTimeException. It is also true that a RunTimeException is an Exception, so an ArithmeticException is an Exception. To put it mathematically, the relation *is a* is transitive.

On the other hand, the fields in a class constitute a *has-a* relationship to the class. For example, the name field in the Employee class is of type (reference to) String, so we can say an Employee object has a String. Also, as you learned in Lab 2, the bestPaid field in the Company class is of type (reference to) Employee, so we can say a Company object has an Employee.

Typically, if class B shares the overall functionality of A, then inheritance of A by B is preferable. But if there is only one component of B that will benefit from A's methods, the better alternative will be to define an A object as a field in class B. That

object can invoke the relevant methods from class A. Often, the choice is not clear-cut, so experience is your best guide. We will encounter this problem in Chapters 5 and 7.

With an object-oriented approach, the emphasis is not so much on developing the program as a whole but on developing modular program-parts, namely, classes. These classes not only make the program easier to understand and to maintain, but they are reusable for other programs as well. A further advantage to this approach is that decisions about a class can easily be modified. We first decide what classes will be needed. And because each class interacts with other classes through its method descriptions, we can change the class's fields and method definitions as desired as long as the method descriptions remain intact.

There are three essential features of object-oriented programming: the *encapsulation* of fields and methods into a single entity, the *inheritance* of a class's fields and methods by subclasses, and *polymorphism*, discussed in the next section.

1.1.13 Polymorphism

One of the major aids to code reuse in object-oriented languages is polymorphism. ***Polymorphism***—from the Greek words for "many" and "shapes"—is the ability of a reference to refer to different objects. For a simple example of this surprisingly useful concept, suppose *D* is a subclass of A and that D overrides A's scan method. If found is a **boolean** variable whose value is determined at run-time, we could have the following:

```
A a;   // a is a reference to an object of type A, the superclass

if (found)
      a = new A();
else
      a = new D();      // a is a reference to an object of type D, the subclass
a.scan();
```

With polymorphism, the method invoked in a message depends on the run-time type of the object that invoked the method.

In the message a.scan(), which version of scan is being invoked? That determination cannot be made at compile-time because until run-time, it cannot be determined whether a is a reference to an A object or to a D object. To put it another way, a is a polymorphic reference.

For a slightly longer but more revealing example, let's go back to the Company class. That class had a processInput method. Later, we created a subclass, Company2, with a slightly modified version of processInput. Namely, the new version had a different input prompt and constructed an HourlyEmployee object instead of an Employee object. Now suppose we wanted to combine the two applications. For employee input, the name and gross pay are entered; for hourly-employee input, the name, hours worked, and pay rate are entered.

Here is a CompanyAll class to handle the combined application:

```
import java.util.*;

public class CompanyAll implements Process {

    protected static final String SENTINEL_MESSAGE =
        "The sentinels for employee input are *** and -1.00.\n" +
        "The sentinels for hourly-employee input are ***, −1 and −1.00.";

    protected static final String EMPLOYEE_PROMPT =
        "\n\nIn the Input line, please enter a name (with no blanks), " +
        "\nand gross pay, followed by the Enter key.";

    protected static final String HOURLY_PROMPT =
        "\n\nIn the Input line, please enter a name (with no blanks), " +
        "\nhours worked and pay rate, followed by the Enter key.";

    protected static final String GENERIC_PROMPT =
        "\nFor employee input:" + EMPLOYEE_PROMPT +
        "\n\nFor hourly-employee input:" + HOURLY_PROMPT;

    protected Employee bestPaid;

    protected GUI gui;

    protected boolean atLeastOneEmployee;

    // Postcondition: this company has been initialized.
    public CompanyAll( ) {

        bestPaid = new Employee( );
        atLeastOneEmployee = false;
        gui = new GUI (this);
        gui.println (SENTINEL_MESSAGE);
        gui.println (GENERIC_PROMPT);

} // default constructor

// Postcondition: the input string s—consisting of either an employee or an
//                hourly employee—has been processed against what had
//                been this company's best-paid employee.
public void processInput (String s) {

    final String ERROR =
        "\nError: the input is incorrect.\n";

    Employee employee;

    String prompt;

    gui.println (s);
    int count = new StringTokenizer (s).countTokens( );
```

```
        if (count !=2 && count != 3)
            gui.println (ERROR + GENERIC_PROMPT);
        else {
            if (count == 2) { // 2 input fields for an employee
                employee = new Employee (s);
                prompt = EMPLOYEE_PROMPT;
            } // employee input
            else { // 3 input fields for an hourly employee
                employee = new HourlyEmployee (s);
                prompt = HOURLY_PROMPT;
            } // hourly-employee input
            if (!employee.isSentinel( )) {
                atLeastOneEmployee = true;
                if (employee.makesMoreThan (bestPaid))
                    bestPaid = employee;
                gui.println (prompt);
            } // not the sentinel
            else
                printBestPaid( );
            } // 2 or 3 tokens
        } // processInput
        // Postcondition: this Company's best-paid employee has been printed out.
        protected void printBestPaid( ) {
            final String NO_INPUT_MESSAGE =
                "\n\n\nERROR: there were no employees in the input.";
            final String BEST_PAID_MESSAGE =
                "\n\n\nThe best-paid employee (and gross pay) is ";
            final String CLOSE_WINDOW_PROMPT =
                "\n\nPlease close this window when you are ready.";
            if (atLeastOneEmployee)
                gui.println (BEST_PAID_MESSAGE + bestPaid);
            else
                gui.println (NO_INPUT_MESSAGE);
            gui.println (CLOSE_WINDOW_PROMPT);
            gui.freeze( );
        } // method printBestPaid
    } // class CompanyAll
```

In this example, it is legal to write

```
employee = new Employee (s);
```

So, by the subclass substitution rule, it is also legal to write

```
employee = new HourlyEmployee (s);
```

Now consider the meaning of a subsequent message such as

```
!employee.isSentinel( )
```

The version of the isSentinel( ) method executed depends on the type of the object that employee is currently referencing. For example, if the input string s has only two tokens, then employee is assigned a reference to an Employee object, so the Employee class's version of isSentinel( ) is invoked. On the other hand, if the input string s has three tokens, then employee is assigned a reference to an HourlyEmployee object, so the HourlyEmployee class's version of isSentinel( ) is invoked.

In this example, employee is a polymorphic reference: the object referred to can be an Employee or an HourlyEmployee, and this information is not available until run-time. This illustrates an essential aspect of polymorphism:

> When a message is sent, the version of the method invoked depends on the type of the object, not on the type of the reference.

What is important here is that polymorphism allows code reuse for methods related by inheritance. We did not need to explicitly call the two versions of the isSentinel( ) method.

The code for the CompanyAll class raises a question: How can the Java compiler generate the appropriate bytecode for a message such as employee.isSentinel( )? Another way to phrase the same question is this: How can the method identifier isSentinel be *bound* to the correct version—in Employee or HourlyEmployee—at compile-time when the necessary information is not available until run-time? The answer is simple: the binding cannot be done at compile-time but must be delayed until run-time! A method that is bound to its method identifier at run-time is called a ***virtual method.***

In Java, almost all methods are virtual. The only exceptions are for **static** methods (because they are associated with a class itself rather than an object) and for **final** methods (the **final** modifier signifies that the method cannot be overridden in subclasses). This delayed binding—also called ***dynamic binding*** or ***late binding***—of method identifiers to methods is one of the reasons that Java programs execute more slowly than programs in some other languages.

It turns out that polymorphism is a key feature of the Java language and makes the Java Collections Framework possible. We will have more to say about this in Chapter 2.

The CompanyAll class utilized the Company and Employee classes. Would it be correct to say that users of the Company and Employee classes *had* to obey the principle of data abstraction? The next section considers the extent to which a language can allow developers of a class to force users of that class to obey the principle of data abstraction.

1.1.14 Information Hiding

The principle of data abstraction states that a user's code should not access the implementation details of the class used. By following that principle, the user's code is protected from changes to those implementation details, such as a change in fields or method definitions.

Protection is further enhanced if a user's code is *prohibited* from accessing the implementation details of the class used. ***Information hiding*** means making the implementation details of a class inaccessible to code that uses that class. The burden of obeying the principle of data abstraction falls on users, whereas information hiding is a language feature that allows class developers to keep users from violating the principle of data abstraction.

As you saw in Labs 1 and 2, Java supports information hiding through the use of the private, protected, and default-visibility modifiers for methods, fields, and constants. Through its visibility modifiers, Java forces users to access class members only to the extent permitted by the developers.

Java also has a feature that enables developers to make sure that a method's precondition is satisfied before a user can invoke that method: the exception mechanism. But that is just one aspect of exceptions. In the next section, you will learn more about exceptions, especially, how to handle them. Java's exception-handling mechanism provides programmers with significant control over what happens when errors occur.

1.1.15 Exception Handling

Recall that an ***exception*** is an object created by an unusual condition, typically, an attempt at invalid processing. When an exception object is constructed, the normal flow of control is halted; the exception is said to be *thrown.* Control is immediately transferred to code—either in the current method or in some other method—that "handles" the exception. The exception handling usually depends on the particular exception and may involve printing an error message, terminating the program, taking other action, or maybe doing nothing.

A ***robust*** program is one that does not terminate unexpectedly from invalid user input. We almost always prefer programs that—instead of "crashing"—allow recovery from an error such as the input of 7.o instead of 7.0 for a **double**. Java's exception-handling feature allows the programmer to avoid almost all unexpected terminations.

For an example of how exceptions can be handled by the programmer, suppose we have a block of code with one or more array references. Using the reserved words **try** and **catch**, we "try" to execute the block of code and "catch" an ArrayIndex OutOfBounds exception if one occurs. Here is the code—the parenthetical expression after the word **catch** gives the specific exception class and an object in that class:

```
final int ARRAY_SIZE = 5;
int[ ] a = new int [ARRAY_SIZE];
```

```
int i = 0;

try {

        i = 3;
        a [i] = 2;
        i = 5;
        a [i] = 12;
        i = −1; // This statement will never be executed.
        a [i] = 1;

} // try
catch (ArrayIndexOutOfBoundsException exception) {

        gui.println (exception + "i = " + i);
} // catch

gui.println ("continuing");
```

Because the array indices range from 0 to 4, an exception is thrown when i, with a value of 5, is used as an array index. When that exception is thrown during the execution of the **try** block, the rest of the **try** block is ignored and control is transferred to the **catch** block—the **catch** block must immediately follow the **try** block. In this example, the output would be

```
java.lang.ArrayIndexOutOfBoundsException: i = 5
continuing
```

The fact that "continuing" is printed indicates that there is no abnormal termination: the remainder of the program is executed.

You could avoid the throwing of an exception by testing each array index before making the array reference:

```
if (i >= 0 && i < ARRAY_SIZE)

        . . . do something with/to a [i]
else
        gui.println (i + " is an illegal array reference.");
```

But to do this for every array reference would unnecessarily clutter up your code. The "**try** . . . **catch**" mechanism provides a single place to handle all illegal indices in a block of code.

Immediately after a **try** block there must be at least one **catch** block. What if you had several possible exceptions in a block of code? No problem! You simply use several **catch** blocks. For example:

```
try {
        . . .
} // try
catch (ArrayOutOfBoundsException except1) {
        . . .
} // catch array indexes
```

```
catch (NullPointerException except2) {
        . . .
} // catch null pointers
```

Exceptions can be explicitly thrown by the programmer. For example, suppose we have a method that searches part of an array scores for givenScore, and two of the formal parameters are start and finish, the bounds of the indices to be searched. If either start is less than zero or finish is greater than or equal to scores.length, we want to throw an exception and print an appropriate message. The mechanism for throwing the exception is the **throw** statement, which can be placed anywhere a statement is allowed. For example, the code may be

```
if (start < 0 || finish >= scores.length)
        throw new IllegalArgumentException ("start too small or finish too big");
```

If the given condition is true, the **throw** statement is executed, which creates a new instance of the exception class IllegalArgumentException. If the **throw** statement is in a **try** block, you could set up a **catch** block to handle this exception:

```
catch (IllegalArgumentException e) {

        // whatever

} // catch
```

If there are no applicable catch blocks to handle this exception, the program will terminate with the output message:

```
java.lang.IllegalArgumentException: start too small or finish too big
```

In this example, having the exception explicitly raised may be better than having the ArrayIndexOutOfBoundsException thrown in subsequent code:

```
try {

        for (int i = start; i <= finish; i++)
                if (scores [i] == givenScore)
                        return i;
        return -1;

} // try
catch (ArrayIndexOutOfBoundsException e) {

        // whatever

} // catch
```

The problem with the ArrayIndexOutOfBoundsException is that, even if finish is too large, the exception will not be thrown unless i is too large—that is, unless the givenScore was not found at an earlier index. The user might not want the search to be conducted if finish is too large. That is why it is better, in this situation, to have the IllegalArgumentException explicitly thrown, as previously done.

A user can even create new exception classes, for example,

```
import java.io.*;
public class ProcrastiPhobiaException extends RuntimeException {

        public ProcrastiPhobiaException ( ) {
                System.out.println ("Start projects early!");
        } // default constructor

} // class ProcrastiPhobiaException
```

To throw this exception, we write:

```
throw new ProcrastiPhobiaException( );
```

This creates a new instance of the class ProcrastiPhobiaException. Assume there is no catch block to handle this exception. Then the program would terminate with the output:

```
Start projects early!
java.lang.reflect.InvocationTargetException: ProcrastiPhobiaException
```

1.1.16 Propagating Exceptions

What happens if an exception, such as NumberFormatException, is thrown in a method that does not catch that exception? Then control is transferred back to the calling method: the method that called the method that threw the exception. This transferring of control is known as ***propagating the exception.*** For example, we can have the following:

```
public void printAverage( ) {
        try {
                gui.println (getAverage( ));
        } // try
        catch (ArithmeticException e) {
                gui.println ("Attempt to divide by 0. " + e)
        } // catch
} // method printAverage
public double getAverage( ) {
        double sum = 35.3;
        int n = 0;
        return sum / n;
} // method getAverage
```

The exception is thrown in the method getAverage( ) but is caught in the method printAverage( ). The output will be

```
Attempt to divide by 0. java.lang.ArithmeticException: / by zero
```

If the calling method does not handle the exception, then the exception is propagated back to the calling method of the calling method itself. Ultimately, if the exception has not been caught even in the *main* method, the program will terminate abnormally and a message describing the exception will be printed. The advantage to propagating an exception is that the exception can be handled at a higher level in the program. Decisions to change how the exception is handled can be made in one place, rather than scattered throughout the program. Also, the higher level might have facilities not available at lower levels, such as a GUI class.

An important category of exception is the *checked exception,* such as when a file is not found or the end-of-file marker is encountered while input is being read. If a method contains a statement whose execution may throw a checked exception, the exception can be caught within the method itself. But if not, the compiler requires—to allow propagation of the exception—that a throws clause be appended to the method heading. We might have

```
public void sample( ) throws IOException {
```

This indicates that the sample method might throw an IOException. If so, the exception will be propagated back to the method that called sample. That calling method *must* either catch IOException or append the same throws clause to its method heading. And so on.

For example, here is a simple program that illustrates how checked exceptions can be handled (the two classes are in different files):

```
import java.io.*;

public class ExceptionMain {
    public static void main (String[ ] args) {
        ExceptionHandling.try1( );
    } // method main
// class ExceptionMain

public class ExceptionHandling {
    public static void try1( ) {
        System.out.println ("start of try1");
        try {
            try2( );
        } // try
        catch (IOException e) {
            System.out.println (e + "something went wrong with IO");
```

```
            } // catch
            System.out.println ("end of try1");
        } // method try1

        public static void try2( ) throws IOException {
            System.out.println ("start of try2");
            try3( );
            System.out.println ("end of try2");
        } // method try2

        public static void try3( ) throws IOException {
            System.out.println ("start of try3");
            throw new IOException ("IO error in method try3: ");
        } // method try3

    } // class ExceptionHandling
```

The IOException is explicitly raised in try3 but not caught there. So the throws clause after try3's heading ensures that the exception is propagated back to try3's calling method: try2. Then try2 propagates the exception back to try1, which catches the exception. So the main method need not have either a throws clause or a catch block.

The output of this program will be

```
start of try1
start of try2
start of try3
java.io.IOException: IO error in method try3: something went wrong with IO
end of try1
```

A checked exception must be caught or specified in a throws clause, and the compiler "checks" to make sure this has been done. Which exceptions are checked, and which are unchecked? Subclasses of RuntimeException—NumberFormatException, NullPointerException, IndexOutOfBoundsException, and so on—are unchecked; all other exceptions are checked. The motivation behind this is that a RuntimeException, such as NullPointerException or NumberFormatException, can occur in almost any method. So appending a throws clause to the heading of such a method would burden the developer of the method without providing any helpful information to the reader of that method.

Exceptions also play a role in the user-developer relationship. The developer of a method can throw an exception if the precondition is violated, and this forces the user to ensure the precondition is true prior to method invocation. For example, we might have the following:

```
// Precondition: n >= 0. Otherwise, IllegalArgumentException will be thrown.
// Postcondition: the factorial of n has been returned.
```

```
public void factorial (int n) {
    if (n < 0)
        throw new IllegalArgumentException( );
    . . .
} // method factorial
```

In Lab 3, you will modify the SalariedEmployee class from Lab 2 to handle exceptions.

LAB Lab 3: An Example of Exception Handling. *All Labs Are Optional* LAB

SUMMARY

This chapter presents the Java material you will need in subsequent chapters. We started with a brief review of classes and objects. The three essential features of an object-oriented language are

1. *Encapsulation* of fields and methods into a single entity, the class
2. *Inheritance* of a class's fields and methods by subclasses
3. *Polymorphism,* the ability of a reference to refer to different objects

Data abstraction—the separation of method descriptions from field and method definitions—is a way for users of a class to protect their code from being affected by changes in the implementation details of that class. Java, with its private, protected, and default visibility, supports *information hiding,* whereby the developer of a class can prohibit users' code from accessing implementation details of the class. Java's exception-throwing mechanism provides another way for developers to force users to invoke methods properly. The exception-handling facilities promote *robust* programs: they do not terminate unexpectedly from invalid user input.

EXERCISES

1.1 In the CalendarDate class, develop a method description for a daysLeft InMonth method. If, for example, myDate is a reference to a CalendarDate object whose date is February 13, 2003, then

```
myDate.daysLeftInMonth( )
```

will return 15.

1.2 Define the daysLeftInMonth( ) method from the method description developed in Exercise 1.1. Assume the CalendarDate class has **int** fields day, month, and year.

1.3 Here is a simple class—but with method descriptions instead of method definitions—to find the highest age in the input and to print out that age:

```
import java.util.*;

public class Age implements Process {

    protected static final String INPUT_PROMPT =
        "\nIn the Input line, please enter an age; the sentinel is ";

    protected static final int SENTINEL = -1;

    protected static final String HIGHEST_MESSAGE =
        "\n\n\nThe highest age is ";

    GUI gui;

    int age,
        highestAge;

    // Postcondition: this Age has been initialized.
    public Age();

    // Postcondition: the input string s, consisting of an age, has been
    //                processed against the highest age so far.
    public void processInput (String s);

    // Postcondition: the highest age has been printed.
    private void printHighest();

} // class Age
```

a. Fill in the method definitions for the Age class.
b. Test your Age class by developing a project and running the project.
c. Modify your Age class to handle NumberFormatException—if the String read in is not an integer.

1.4 With the Age class in Exercise 1.3a as a guide, develop a Salary class to read in salaries from the input until the sentinel (−1.00) is reached and to print out how many of those salaries are above average. The average salary is the total of the salaries divided by the number of salaries. Assume there will be at most 100 salaries in the input. Use an array to hold the salaries.

1.5 Suppose Y and Z are subclasses of X, and we have

```
X x = new X();
Y y = new Y();
Z z = new Z();

x = z;
```

Which one of the following assignments would be legal both at compile-time and at run-time?

a. `z = (Z)x;`
b. `z = x;`
c. `z = (X)x;`
d. `y = (Y)x;`

Create a small project to validate your claim.

1.6 Given the classes below (and the usual GUI,GUIListener and Process classes), determine the output when the project is run. Assume each class is in a separate file.

```
public class PolyMain {

    public static void main (String args[ ]) {

        Poly poly = new Poly( );

    } // method main

} // class PolyMain

public class Poly implements Process {

    protected static final String INPUT_PROMPT =
        "In the Input Line, please enter 0 or a non-zero integer.";

    protected GUI gui;

    public Poly( ) {

        gui = new GUI (this);
        gui.println (INPUT_PROMPT);

    } // constructor

    public void processInput (String s) {

        A a;

        int code = Integer.parseInt (s);

        if (code == 0)
            a = new A( );
        else  // non-zero int entered
            a = new D( );
        gui.println (a);
    } // method processInput

} // class Poly

public class A
{
    public String toString( ) {

        return "A";

    } // method toString

} // class A

public class D extends A {
```

```
    public String toString( ) {
        return "D";
    } // method toString
} // class D
```

1.7 The following class does not do much. It prompts the end-user to enter a file name; when the file name is entered, that file—if it exists—is opened. Modify the class so that if there is no file by the given name, the end-user is reprompted. The program should not terminate until the end-user enters the name of an existing file.

```
import java.io.*;
public class FileOpener implements Process {

    protected final String PROMPT =
        "\nIn the Input line, please enter a file name.";
    protected GUI gui;

    // Postcondition: this FileOpener has been initialized.
    public FileOpener( ) {
        gui = new GUI (this);
        gui.println (PROMPT);
    } // default constructor

    // Postcondition: The input string s has been processed.
    public void processInput (String s) {
        final String CLOSE_WINDOW_PROMPT = "The execution of
                this project is complete." + Please close this window
                when you are ready.";

        BufferedReader fileReader = new BufferedReader (new
                FileReader (s));
        gui.println (CLOSE_WINDOW_PROMPT);
        gui.freeze( );
    } // method processInput

} // class FileOpener
```

Hint Try to open the file, and catch the FileNotFoundException.

1.8 If a class, such as Company, constructs a GUI window, the class's only public methods should be its own constructor and processInput. Explain.

PROGRAMMING PROJECT 1.1

Developing and Using a Sequence Class

In this project, you will get to be a developer of a class, and then become a user of that class. To start with, here are method descriptions for a Sequence class that holds a sequence of Integers:

```
// Precondition: n >= 1. Otherwise an IllegalAgrumentException object will be
                 thrown.
// Postcondition: this is an empty Sequence that will hold at most n Integers.
public Sequence (int n);

// Postcondition: the number of Integers currently in this Sequence has been
//                returned.
public int size( );

// Precondition:  this Sequence will hold at least one more Integer. Otherwise,
//                an IllegalStateException object will be thrown.
// Postcondition: anInt has been appended (inserted at the end of) this Sequence.
public void append (Integer anInt);

// Precondition:  this Sequence has at least k Integers. Otherwise, an
//                IndexOutOfBoundsException object will be thrown.
// Postcondition: the element at position k in this Sequence has been returned.
public Integer get (int k);

// Precondition:  this Sequence has at least k Integers. Otherwise, an
//                IndexOutOfBoundsException object will be thrown.
// Postcondition: the element at position k in this Sequence has been changed to
//                newInt.
public void set (int k, Integer newInt);
```

Part 1

Define the methods in the Sequence class.

Hint Use the following fields:

```
protected Integer[ ] data;
protected int size; // the number of elements in the Sequence object, not the
                    // capacity of the data array.
```

(continued on next page)

(continued from previous page)

Part 2

Create a SequenceTester class that implements the Process interface. The SequenceTester class will resemble the Company class, with a gui field and a process Input (String s) method. There will also be a field:

```
protected Sequence sequence;
```

The processInput method should consist of a **try** block and **three** catch blocks (to print a message for IllegalArgumentException, IllegalStateException, and IndexOutOf Bounds Exception). In the processInput method, if s is not the sentinel ("***"), s is converted to an Integer and appended to sequence. When the sentinel is reached, the minimum and maximum Integers in Sequence are printed, all occurrences of 13 are changed to 31, and then the number of Integers in Sequence between 18 and 65, inclusive, is printed. Also, for each of the four Sequence methods that can throw an exception, make a call that will throw the exception.