Licensed to: iChapters User

Data Structures and Algorithms in Java

SECOND EDITION

Adam Drozdek

Australia • Canada • Mexico • Singapore • Spain • United Kingdom • United States

Licensed to: iChapters User

THOMSON
COURSE TECHNOLOGY

Data Structures and Algorithms in Java, Second Edition
by Adam Drozdek

Senior Acquisitions Editor: *Amy Yarnevich*
Product Manager: *Alyssa Pratt*
Editorial Assistant: *Amanda Piantedosi*
Senior Marketing Manager: *Karen Sietz*
Production Editor: *Jennifer Harvey*
Associate Product Manager: *Mirella Misiaszek*
Cover Design: *Joel Sadagursky*
Compositor: *Pre-Press Company, Inc.*

COPYRIGHT © 2005 Course Technology, a division of Thomson Learning, Inc.
Thomson Learning™ is a trademark used herein under license.

Printed in the United States of America

1 2 3 4 5 6 7 8 9 BM 06 05 04 03 02

For more information, contact Course Technology,
25 Thomson Place, Boston, Massachusetts, 02210.
Or find us on the World Wide Web at: www.course.com

CENGAGEbrain.com

ALL RIGHTS RESERVED. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, Web distribution, or information storage and retrieval systems—without the written permission of the publisher.

For permission to use material from this text or product, contact us by
Tel (800) 730-2214
Fax (800) 730-2215
www.thomsonrights.com

Disclaimer
Course Technology reserves the right to revise this publication and make changes from time to time in its content without notice.

ISBN 0-534-49252-5

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

1 Object-Oriented Programming Using Java

This chapter introduces the reader to elementary Java. Java is an immense language and programming environment, and it is impossible to touch upon all Java-related issues within the confines of one chapter. This chapter introduces only those aspects of Java that are necessary for understanding the Java code offered in this book. The reader familiar with Java can skip this chapter.

CENGAGE brain

1.1 Rudimentary Java

A Java program is a sequence of statements that have to be formed in accordance with the predefined syntax. A statement is the smallest executable unit in Java. Each statement ends with a semicolon. Compound statements, or blocks, are marked by delimiting them with braces, { and }.

1.1.1 Variable Declarations

Each variable must be declared before it can be used in a program. It is declared by specifying its type and its name. Variable names are strings of any length of letters, digits, underscores, and dollar signs that begin with a letter, underscore, or dollar sign. However, a letter is any Unicode letter (a character above 192), not just 1 of the 26 letters in the English alphabet. Local variables must be initialized. Java is case sensitive, so variable `n` is different from variable `N`.

A type of variable is either one of the eight built-in basic types, a built-in or user-defined class type, or an array. Here are built-in types and their sizes:

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

Type	Size	Range
`boolean`	1 bit	`true`, `false`
`char`	16 bits	Unicode characters
`byte`	8 bits	[-128, 127]
`short`	16 bits	[-32768, 32767]
`int`	32 bits	[-2147483648, 2147483647]
`long`	64 bits	[-9223372036854775808, 9223372036854775807]
`float`	32 bits	[-3.4E38, 3.4E38]
`double`	64 bits	[-1.7E308, 1.7E308]

Note that the sizes of the types are fixed, which is extremely important for portability of programs. In C/C++, the size of integers and long integers is system dependent. Unlike C/C++, `boolean` is not a numeric type, and no arithmetic operations can be performed on `boolean` variables. But as in C/C++, characters are considered integers (in Java, they are unsigned integers) so that they can be operands of arithmetic operations.

Integer operations are performed with 32-bit precision (for long integers, it is 64-bit precision); therefore, operations on `byte` and `short` variables require a cast. For example, the statements

CENGAGE brain.com

```
byte a, b = 1, c = 2;
a = b + c;
```

give a compilation error, "incompatible type for =. Explicit cast is needed to convert `int` to `byte`." The addition `b + c` gives an integer value that must be cast to execute the assignment to the `byte` variable `a`. To avoid the problem, the assignment should be changed to

```
a = (byte) (b + c);
```

An overflow resulting from an arithmetic operation (unless it is division by zero) is not indicated, so the programmer must be aware that, for two integers,

```
int i = 2147483647, j = i + 1;
```

the value of `j` is –2147483648.

Java does not provide modifiers `signed` and `unsigned`, but it has other modifiers.

An important difference between C/C++ and Java is characters that are 8 bits long in C/C++ and 16 bits long in Java. With the usual 8-bit characters, only 256 different characters can be represented. To address the problem of representing characters of languages other than English, the set of available codes must be significantly extended. The problem is not only with representing letters with diacritical marks (e.g., Polish letter ń, Romanian letter ţ, or Danish letter ø), but also with non-Latin characters such as Cyrillic, Greek, Japanese, Chinese, and so on. By allowing a character variable to be of 2 bytes, the number of different characters represented now equals 65,536.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

To assign a specific Unicode character to a character variable, "u" followed by four hexadecimal digits can be used; for example,

```
char ch = '\u12ab';
```

However, high Unicode codes should be avoided, because as of now, few systems display them. Therefore, although the assignment to `ch` just given is legal, printing the value of `ch` results in displaying a question mark.

Other ways of assigning literal characters to character variables is by using a character surrounded with single quotes,

```
ch = 'q';
```

and using a character escape sequence, such as

```
ch = '\n';
```

to assign an end-of-line character; other possibilities are: `'\t'` (tab), `'\b'` (backspace), `'\r'` (carriage return), `'\f'` (formfeed), `'\''` (single quote), `'\"'` (double quote), and `'\\'` (backslash). Unlike C/C++, `'\b'` (bell) and `'\v'` (vertical tab) are not included. Moreover, an octal escape sequence '*ddd*' can be used, as in

```
ch = '\123'; // decimal 83, ASCII of 'S';
```

CENGAGE brain.com

where *ddd* represents an octal number [0, 377].

Integer literals can be expressed as decimal numbers by any sequence of digits 0 through 9,

```
int i = 123;
```

as octal numbers by 0 followed by any sequence of digits 0 through 7,

```
int j = 0123; // decimal 83;
```

or as hexadecimal numbers by "0x" followed by any sequence of hexadecimal numbers 0 through 9 and A through F (lower- or uppercase),

```
int k = 0x123a; // decimal 4666;
```

Literal integers are considered 32 bits long; therefore, to convert them to 64-bit numbers, they should be followed by an "L":

```
long p = 0x123aL;
```

Note that uppercase *L* should be used rather than lowercase *l* because the latter can be easily confused with the number 1.

Floating-point numbers are any sequences of digits 0 through 9 before and after a period; the sequences can be empty: 2., .2, 1.2. In addition, the number can be followed by a letter *e* and a sequence of digits possibly preceded by a sign: 4.5e+6 ($= 4.5 \cdot 10^6 = 4500000.0$), 102.055e–3 $= 102.055 \cdot 10^{-3} = .102055$). Floating-point literals are 64-bit numbers by default; therefore, the declaration and assignment

```
float x = 123.45;
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

result in a compilation error, "incompatible type for declaration. Explicit cast needed to convert double to float," which can be eliminated by appending the modifier f (or F) at the end of the number,

```
float x = 123.45f;
```

A modifier d or D can be appended to double numbers, but this is not necessary.

1.1.2 Operators

Value assignments are executed with the assignment operator =, which can be used one at a time or can be strung together with other assignment operators, as in

```
x = y = z = 1;
```

which means that all three variables are assigned the same value, number 1. Java uses shorthand for cases when the same value is updated; for example,

```
x = x + 1;
```

can be shortened to

```
x += 1;
```

Java also uses autoincrement and autodecrement prefix and postfix operators, as in `++n`, `n++`, `--n`, and `n--`, which are shorthands of assignments `n = n + 1` and `n = n - 1`, where `n` can be any number, including a floating-point number. The difference between prefix and postfix operators is that, for the prefix operator, a variable is incremented (or decremented) first and then an operation is performed in which the increment takes place. For a postfix operator, autoincrement (or autodecrement) is the last operation performed; for example, after executing assignments

```
x = 5;
y = 6 + ++x;
```

`y` equals 12, whereas after executing

```
x = 5;
y = 6 + x++;
```

`y` equals 11. In both cases, `x` equals 6 after the second statement is completely executed.

Java allows performing operations on individual bits with bitwise operators: & (bitwise and), | (bitwise or), ^ (bitwise xor), << (left shift), >> (right shift), >>> (zero filled shift right), and ~ (bitwise complement). Shorthands &=, |=, ^=, <<=, >>=, and >>>= are also possible. Except for the operator >>>, the other operators are also in C/C++. The operator >> shifts out a specified number of rightmost (least significant) bits and shifts in the same number of 0s for positive numbers and 1s for negative numbers. For example, the value of `m` after the assignments

```
int n = -4;
int m = n >> 1;
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

is –1 because –4 in `n` is a two-complement representation as the sequence of 32 bits 11 . . . 1100, which after shifting to the right by one bit gives in `m` the pattern 11 . . . 1110, which is a two-complement representation of –2. To have 0s shifted in also for negative numbers, the operator >>> should be used,

```
int n = –4;
int m = n >>> 1;
```

in which case, the pattern 11 . . . 1100 in `n` is transformed into the pattern 01 . . . 1110 in `m`, which is the number 2147483646 (one less than the maximum value for an integer).

1.1.3 Decision Statements

One decision statement is an `if` statement

```
if (condition)
        do something;
[else  do something else;]
```

in which the word `if` is followed by a condition surrounded by parentheses, by the body of the `if` clause, which is a block of statements, and by an optional `else` clause, which is the word `else` followed by a block of statements. A condition must return a Boolean value (in C/C++, it can return any value). A condition is formed with relational operators <, <=, ==, !=, >=, > that take two arguments and return a Boolean value, and with logical operators that take one (!) or two (&&, ||) Boolean arguments and return a Boolean value.

An alternative to an `if-else` statement is the conditional operator of the form

```
condition ? do-if-true : do-if-false;
```

The conditional operator returns a value, whereas an `if` statement does not, so the former can be used, for example, in assignments, as in

```
  n = i <= 0 ? 10 : 20;
```

Another decision statement is a `switch` statement, which is a shorthand for nested `if` statements. Its form is as follows:

```
switch (integer expression) {
     case value1: block1; break;
     . . . . . .
     case valueN: blockN; break;
     default: default block;
}
```

The test expression following `switch` must be an integer expression so that any expression of type `byte`, `char`, `short`, and `int` can be used. The value of the expression is compared to the values that follow the word `case`. If a match is found, the block of statements following this value is executed, and upon encountering `break`, the `switch` statement is exited. Note that if the word `break` is missing, then

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

execution is continued for the block of the next `case` clause. After executing the statement

```
switch (i) {
    case 5 : x + 10; break;
    case 6 : x = 20;
    case 7 : x *= 2; break;
    default : x = 30;
}
```

the value of `x` is 10 if `i` equals 5, 40 if `i` equals 6, it is doubled if `i` equals 7, and is 30 for any other value of `i`.

1.1.4 Loops

The first loop available in Java is the `while` loop:

```
while (condition)
    do something;
```

The condition must be a Boolean expression.

The second loop is a `do-while` loop:

```
do
    do something;
while (condition);
```

The loop continues until the Boolean condition is false.

The third loop is the `for` loop:

```
for (initialization; condition; increment)
    do something;
```

The initialization part may also declare variables, and these variables exist only during execution of the loop.

A loop can be exited before all the statements in its body are executed with an unlabeled `break` statement. We have already seen a `break` statement used in the `switch` statement. In the case of nested loops, when a `break` statement is encountered, the current loop is exited so that the outer loop can be continued. An unlabeled `continue` statement causes the loop to skip the remainder of its body and begin the next iteration.

1.1.5 Exception Handling

If an error is detected during execution of a Java program, Java throws an exception, after which the program is terminated and an error message is displayed informing the user which exception was raised (that is, what type of error occurred and where in the program it happened). However, the user may handle the error in the program should one occur, at least by making the program ignore it so that execution of the program can continue. But if an exception is raised, a special course of action can be

CENGAGEbrain.com

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

undertaken and then the program can continue. Catching an error is possible by using a `try-catch` mechanism.

A general format of the `try-catch` statement is

```
try {
    do something;
} catch (exception-type exception-name) {
    do something;
}
```

The number of `catch` clauses is not limited to one. There can be as many as needed, each one for a particular exception.

In this statement, execution of the body of the `try` clause is tried, and if an exception occurs, control is transferred to the `catch` clause to execute its body. Then execution of the program continues with a statement following the `try-catch` statement, unless it contains the `throw` clause, which causes an exit from the method.

Consider the following method:

```
public int f1(int[] a, int n) throws ArrayIndexOutOfBoundsException {
    return n * a[n+2];
}
```

The `throws` clause in the heading of the method is a warning to the user of the method that a particular exception can occur, and if not handled properly, the program crashes. To prevent that from happening, the user may include the `try-catch` statement in the caller of `f1()`:

```
public void f2() {
    int[] a = {1,2,3,4,5};
    try {
        for (int i = 0; i < a.length; i++)
            System.out.print(f1(a,i) + " ");
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Exception caught in f2()");
        throw e;
    }
}
```

The `catch` clause prints a message, but it does not perform any fixing operation on the array `a`, although it could. In this example, the `catch` clause also includes the `throw` statement, although this is not very common. In this way, not only is the exception caught and handled in `f2()`, but also a caller of `f2()` is forced to handle it, as in the method `f3()`:

```
public void f3() {
    try {
        f2();
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Exception caught in f3()");
    }
}
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

If the caller of `f2()` does not handle the exception, the program crashes, although the exception was caught in `f2()`. The same fate meets a program if a caller of `f1()` does not catch the exception:

```
public void f4() {
    int[] a = {1,2,3,4,5};
    for (int i = 0; i < a.length; i++)
        System.out.print(f1(a,i) + " ");
}
```

Note that the behavior of the program in all these cases is the same (that is, handling an exception, passing it to another method, or crashing the program) if the `throws` clause is not included in the heading of `f1()` so that `f1()` could simply be:

```
public int f1(int[] a, int n) {
    return n * a[n+2];
}
```

The `throws` clause is thus a very important signal for the user to a possible problem that may occur when calling a particular method.

Not all types of exceptions can be ignored as the exception raised by `f1()` is ignored by `f4()`. Most of the time, exceptions have to be handled in the program; otherwise, the program does not compile. This, for instance, is the case with `IOException` thrown by I/O methods; therefore, these methods are usually called inside `try-catch` clauses.

CENGAGE brain

1.2 Object-Oriented Programming in Java

1.2.1 Encapsulation

Object-oriented programming (OOP) revolves around the concept of an object. Objects, however, are created using a class definition. A *class* is a template in accordance to which objects are created. A class is a piece of software that includes a data specification and functions operating on these data and possibly on the data belonging to other class instances. Functions defined in a class are called *methods,* and variables used in a class are called *class scope variables* (to distinguish them from variables local to method or blocks), *data fields,* or simply *fields*. This combining of the data and related operations is called *data encapsulation.* An object is an instance of a class, an entity created using a class definition.

In contradistinction to functions in languages that are non–object-oriented languages (OOL), objects make the connection between data and methods much tighter and more meaningful. In non-OOLs, declarations of data and definitions of functions could be interspersed throughout the entire program, and only the program documentation indicates that there is a connection between them. In OOLs, a connection is established right at the outset; in fact, the program is based on this connection. An object encompasses related data and operations, and because there may be many objects used in the same program, the objects communicate by exchanging messages, thereby re-

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

vealing to each other only as much, or rather as little, detail about their internal structure as necessary for adequate communication. Structuring programs in terms of objects allows us to accomplish several goals.

First, this strong coupling of data and operations can be used much better in modeling a fragment of the world, which is emphasized especially by software engineering. Not surprisingly, OOP has its roots in simulation; that is, in modeling real-world events. The first OOL was Simula; it was developed in the 1960s in Norway.

Second, objects allow for easier error finding because operations are localized to the confines of their objects. Even if side effects can occur, they are easier to trace.

Third, objects allow us to conceal certain details of their operations from other objects so that these operations may not be adversely affected by other objects. This is known as the *information-hiding principle.* In languages that are not object-oriented, this principle can be found to some extent in the case of local variables, or as in Pascal, in local functions and procedures, which can be used and accessed only by the function defining them. This is, however, a very tight hiding or no hiding at all. Sometimes we may need to use (again, as in Pascal) a function *f2* defined in *f1* outside of *f1,* but we cannot. Sometimes we may need to access some data local to *f1* without exactly knowing the structure of these data, but in non-OOLs, we cannot. Hence, some modification is needed, and it is accomplished in OOLs.

An object in OOL is like a watch. As users, we are interested in what the hands show, but not in the inner workings of the watch. We are aware that there are gears and springs inside the watch, but we usually know very little about why all these parts are in a particular configuration. Because of that, we should not have access to this mechanism so that we do not damage it, inadvertently or on purpose. Therefore, this mechanism is hidden from us, we have no immediate access to it, and thereby the watch is protected and works better than when its mechanism is open for everyone to see.

Hence, an object is like a black box whose behavior is very well defined, and we use the object because we know what it does, not because we have an insight into how it does it. This opacity of objects is extremely useful for maintaining them independently of each other. If communication channels between the objects are well defined, then changes made inside an object can affect other objects only as much as these changes affect the communication channels. Knowing the kind of information sent out and received by an object, the object can be replaced more easily by another object more suitable in a particular situation: A new object can perform the same task differently but more quickly in a certain hardware environment. Hence, an object discloses only as much as is needed for the user to utilize it. It has a public part that can be accessed by any user when the user sends a message matching any of the method names revealed by the object. In this public part, the object displays to the user buttons that can be pushed to invoke the object's operations. The user knows only the names of these operations and the expected behavior.

Information hiding tends to blur the dividing line between data and operations. In Pascal-like languages, the distinction between data and functions/procedures is clear and rigid. They are defined differently and their roles are very distinct. OOLs put data and methods together, and to the user of the object, this distinction is much less noticeable. To some extent, this is an incorporation of features of functional languages. LISP, one of the earliest programming languages, allows the user to treat data and functions on a par, because the structure of both is the same.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

We have already made a distinction between particular objects and object types or classes. We write methods to be used with different variables, and by analogy, we do not want to be forced to write as many object declarations as the number of objects required by the program. Certain objects are of the same type and we would like only to use a reference to a general object specification. For single variables, we make a distinction between type declaration and variable declaration. In the case of objects, a distinction is made between a class declaration and its instantiation, which is an object. Consider the following program:

```
class C {
    public C() {
        this("",1,0);
    }
    public C(String s) {
        this(s,1,0);
    }
    public C(String s, int i) {
        this(s,i,0);
    }
    public C(String s, int i, double d) {
        dataField1 = new String(s);
        dataField2 = i;
        dataField3 = d;
    }
    public void method1() {
        System.out.println(dataField1 + " " + dataField2 + " " + dataField3);
    }
    public void method2(int i) {
        method2(i,"unknown");
    }
    public void method2(int i, String s) {
        dataField2 = i;
        System.out.println(i + " received from " + s);
    }
    private String dataField1;
    private int dataField2;
    private double dataField3;
    public static void main (String args[]) {
        C object1 = new C("object1",100,2000),
          object2 = new C("object2"), object3 = new C();
        object1.method2(123);
        object1.method1();
        object2.method2(123,"object2");
    }
}
```

CENGAGEbrain.com

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

The program contains a declaration of class `C`. Inside method `main()`, objects of class type `C` are generated by declaring and instantiating them:

```
C object1 = new C("object1",100,2000),
  object2 = new C("object2"), object3 = new C();
```

It is very important to see that object declarations do not create objects, so that the two lines

```
C object1;
object1.method1();
```

result in a compilation error. The object variable `object1` has to be assigned an object, which can be done directly in declaration by initializing the variable with the operator `new`,

```
C object1 = new C(. . .);
```

Message passing is equivalent to a function call in traditional languages. However, to stress the fact that in OOLs the methods are relative to objects, this new term is used. For example, the call to `method1()` with respect to `object1`,

```
object1.method1();
```

is to be seen as the message `method1()` sent to `object1`. Upon receiving the message, the object invokes its method. Messages can acquire parameters so that

CENGAGE brain .com

```
object1.method2(123);
```

is the message `method2()` with parameter 123 received by `object1`.

The lines containing these messages are in a method of the current object or another object. Therefore, the receiver of the message is identifiable, but not necessarily the sender. If `object1` receives the message `method1()`, it does not know where the message originated. It only responds to it by displaying the information `method1()` encompasses. The same goes for `method2()`. Therefore, the sender may prefer sending a message that also includes its identification, as in

```
object1.method2(123,"object1");
```

The declaration of class `C` contains the method `main()`, which, as in C/C++, is the starting point for execution of programs. Unlike in C/C++, `main()` must be included inside a class; it cannot be a stand-alone method. In this way, after class `C` is stored in a file `C.java`, the file can be compiled with the instruction

```
javac C.java
```

and then the program can be run with

```
java C
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

The `javac` instruction can be applied to any file containing Java code, but the `java` instruction requires that the class with which it is invoked includes method `main()`. The example of class `C` shows only one class, but as we shall see throughout the book, the number of classes is usually not limited to one; however, one of the classes must include the method `main()` to make the program executable.

The signature of the method `main()` is always the same:

```
public static void main(String[] args)
```

It returns no value (`void`) and allows for taking command line arguments from the interpreter by storing them in an array of strings. The `public` modifier belongs to the category of access modifiers. Java uses four access modifiers (three plus one unnamed), which are related to the concept of a package. A package is a collection of classes that are located in one subdirectory. It is a counterpart of a C/C++ library.

Methods and fields declared `public` can be used by any other object.

The `protected` modifier means that a method or a field is accessible to derived classes and in the package that includes the class that declares the method or the data field.

The `private` modifier indicates methods and fields that can be used only by this class.

A default modifier is no modifier at all, which indicates access to methods and fields in the package that includes the class that declares the method or the data field.

Two more modifiers need to be listed. A `final` method of field cannot be changed by derived classes (see Section 1.2.3, Inheritance). A method and field declared `static` are the same for all instances (objects) of the class.

CENGAGE brain .com

1.2.1.1 Class Methods and Class Variables

Static methods and variables are associated with the class itself; there is exactly one instance of a static element even if there is no object of the particular class type. They are called *class methods* and *class variables* (nonstatic variables and methods are called *instance variables* and *instance methods*). The method `main()` must be declared as `static`. Then, all methods and variables it uses directly must also be static (method `f()` called by `main()` must be static, but not when it is called as `obj.f()`). Consider this class:

```
class C2 {
    public void f1() {
        System.out.println("f1()");
    }
    static public void f2() {
        System.out.println("f2()");
    }
}
```

Because `f1()` is not a class method, the command

```
C2.f1();  // error;
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

causes a compilation error; thus, to execute the method, an object of class type C2 has to be created, and then the method invoked, as in

```
C2 c = new C2();
c.f1();
```

or simply

```
(new C2()).f1();
```

On the other hand, because f2() is a class method, the command

```
C2.f2();
```

is sufficient because the method f2() does exist in class C2, even without creating an object of class type C2; that is, a static method can be accessed through a class name, although it can also be accessed through an object name. This is how methods in class Math can be accessed, for example, Math.sqrt(123).

To see how class variables work, consider this class,

```
class C3 {
    public static int n = 0;
    public int m = 0;
    public C3() {
        n++; m++;
    }
    public void display() {
        System.out.println(n + " " + m);
    }
}
```

CENGAGEbrain.com

Execution of

```
C3 c1 = new C3();
c1.display();
C3 c2 = new C3();
c1.display();
c2.display();
```

prints numbers

```
1 1
2 1
2 1
```

Also, n and m can be accessed and modified from the outside, as in

```
C3.n = 10;
c1.m = 11;
c2.m = 12;
```

Note that the class name is used to access n and the object name to access m.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

1.2.1.2 Generic Classes

A very important aspect of OOP in Java is the possibility of declaring generic classes. For example, if we need to declare a class that uses an array for storing some items, then we may declare this class as

```
class IntClass {
    int[] storage = new int[50];
    ..................
}
```

However, in this way, we limit the usability of this class to integers only; hence, if a class is needed that performs the same operations as `IntClass`, but it operates on double numbers, then a new declaration is needed, such as

```
class DoubleClass {
    double[] storage = new double[50];
    ..................
}
```

If `storage` is to hold objects of a particular class, then another class must be declared. It is much better to declare a generic class and decide during the run of the program to which type of items it is referring. Java allows us to declare a class in this way, and the declaration for the example is

CENGAGEbrain.com

```
class GenClass {
    Object[] storage = new Object[50];
    Object find(int n) {
        return storage[n];
    }
    ..................
}
```

Then the decision is made as to how to create two specific objects:

```
GenClass intObject = new GenClass();
GenClass doubleObject = new GenClass();
```

This generic class manifests itself in different forms depending on the way information is stored in it or retrieved from it. To treat it as an object holding an array of integers, the following way of accessing data can be used:

```
int k = ((Integer) intObject.find(n)).intValue();
```

To retrieve data from `doubleObject`, the return value has to be cast as `Double`. The same cast can also be used for `intObject` so that objects respond differently in different situations. One generic declaration suffices for enabling such different forms.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

1.2.1.3 Arrays

Arrays are Java objects, but to the user, they are objects to a very limited extent. There is no keyword with which all arrays are declared. They may be considered instances of an understood array class. The lack of a keyword for all arrays also means that no subclasses can be created (see Section 1.2.3).

Arrays are declared with empty brackets after the name of the type or the name of the array itself. These two declarations are equivalent:

```
int[] a;
```

and

```
int a[];
```

A declaration of a basic data type also creates an item of the specified type. As for all objects, an array declaration does not create an array. An array can be created with the operator `new`, and very often declaration and initialization are combined, as in

```
int[] a = new int[10];
```

This creates an array of 10 cells that are indexed with numbers 0 through 9; that is, the first cell is `a[0]` and the last cell `a[9]`. An array can also be created by specifying the value of its cells,

```
int[] b = {5, 4, 2, 1};
```

CENGAGEbrain.com

which creates a four-cell array of integers.

Unlike C/C++, it is impossible to access a cell that is out of bounds. An attempt to do so, as with the assignment

```
a[10] = 5;
```

results in a run-time error `ArrayIndexOutOfBoundsException`. To avoid this, the length of the array can be checked before performing an assignment with the variable `length` associated with each array, `a.length`.

Note that for arrays, `length` is a variable, not a method, as is the case for strings; therefore, `a.length()` would be an error.

Because arrays are objects, two array variables can refer to the same array:

```
int[] a1 = new int[3], a2 = a1;
a1[0] = 12;
System.out.println(a2[0]); // print 12
```

Arrays are passed to methods by reference; that is, changes performed in a method that takes an array parameter affects the array permanently. For example, if a method for doubling values in integer arrays is defined:

```
public void doubleArray(int[] a) {
    for (int i = 0; i < a.length; i++)
        a[i] += a[i];
}
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

then the following code

```
a1[1] = a1[2] = 13;
doubleArray(a1);
for (int i = 0; i < a1.length; i++)
    System.out.print(a1[i] + " ");
```

results in printing numbers 24, 26, and 26.

In Java 1.2, the class `Arrays` is added with several useful methods to be applied to arrays, in particular `binarySearch()`, `equals()`, `fill()`, and `sort()`. For example, to sort an array, it is enough to import class `java.util.Arrays` and execute one of the versions of the method sort, for example,

```
Arrays.sort(a);
```

1.2.1.4 Wrapper Classes

Except for basic data types, everything in Java is an object. For this reason, many classes in the `java.util` package operate on items of type `Object`. To include basic data types in this category so that the utility classes can also operate on them, the so-called wrapper classes are introduced to provide object versions of basic data types. For example, the `Integer` class is an object wrapper for the type `int`. The class provides several methods. The `Integer` class includes the following methods: `getInteger()` to convert a string into an `Integer`, `parseInt()` to convert a string into an `int`, `convert()` to convert a string into a number when the radix is not known, `toString()` to convert an integer into a string, and a sequence of methods to convert an integer into other basic types: `intValue()`, `longValue()`, and so on.

CENGAGE brain .com

1.2.2 Abstract Data Types

Before a program is written, the programmer should have a fairly good idea of how to accomplish the task being implemented by the program. Hence, an outline of the program containing its requirements should precede the coding process. The larger and more complex the project, the more detailed the outline phase should be. The implementation details should be delayed to the later stages of the project. In particular, the details of the particular data structures to be used in the implementation should not be specified at the beginning.

From the start, it is important to specify each task in terms of input and output. At the beginning stages, we should be more concerned with what the program should do, not how it should or could be done. Behavior of the program is more important than the gears of the mechanism accomplishing it. For example, if an item is needed to accomplish some tasks, the item is specified in terms of operations performed on it rather than in terms of its inner structure. These operations may act upon this item by modifying it, searching for some details in it, or storing something in it. After these operations are precisely specified, the implementation of the program may start. The implementation decides which data structure should be used to make execution most efficient in terms of time and space. An item specified in terms of operations is called an *abstract data type.* In Java, an abstract data type can be part of a program in the form of an interface.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

Interfaces, successors of protocols in Objective-C, are similar to classes, but they can contain only constants (*final* variables) and method prototypes or signatures; that is, specifications of method names, types of parameters, and types of return values. Declarations in an interface are public, and data are final even if they are not labeled. Methods are thus not defined, and the task of defining methods is passed to a class that *implements* an interface (that is, implements as public all the methods listed in the interface). One class can implement more than one interface, the same interface can be implemented by more than one class, and the classes implementing one interface do not have to be related in any way to each other. Therefore, at the first stage of program design, interfaces can be specified, and the specifics of implementation of their methods are left until later for the implementation classes. And because an interface can extend another interface, a top-down design can become part of the program in a very natural way. This allows the program developer to concentrate first on big issues when designing a program, but also allows a user of a particular implementation of an interface to be certain that all the methods listed in the interface are implemented. In this way, the user is assured that no method listed in the interface is left out in any of the implementation classes, and all instances of implementation classes respond to the same method calls.

The rigidity of interfaces is somewhat relaxed in abstract classes. A class declared `abstract` can include defined methods; that is, not only method signatures, but also method bodies. A method that is specified only by its signature must also be declared as `abstract`. A class can make an abstract class specific by extending it. Here is an example:

CENGAGEbrain.com

```
interface I {
    void If1(int n);
    final int m = 10;
}
class A implements I {
    public void If1(int n) {
        System.out.println("AIf1 " + n*m);
    }
}
abstract class AC {
    abstract void ACf1(int n);
    void ACf2(int n) {
        System.out.println("ACf2 " + n);
    }
}
class B extends AC {
    public void ACf1(int n) {
         System.out.println("BACf1 " + n);
    }
}
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

1.2.3 Inheritance

OOLs allow for creating a hierarchy of classes so that objects do not have to be instantiations of a single class. Before discussing the problem of inheritance, consider the following class definitions:

```
package basePackage;

class BaseClass {
    public BaseClass() {
    }
    public void f(String s) {
        System.out.println("Method f() in BaseClass called from " + s);
        h("BaseClass");
    }
    protected void g(String s) {
        System.out.println("Method g() in BaseClass called from " + s);
    }
    private void h(String s) {
        System.out.println("Method h() in BaseClass called from " + s);
    }
    void k(String s) {
        System.out.println("Method k() in BaseClass called from " + s);
    }
}
```

CENGAGEbrain.com

A file `BaseClass.java` is in a subdirectory `basePackage`. The directory in which this subdirectory is located contains the file `testInheritance.java`, which contains the following classes:

```
class Derived1Level1 extends BaseClass {
    public void f(String s) {
        System.out.println("Method f() in Derived1Level1 called from " + s);
        g("Derived1Level1");
    }
    public void h(String s) {
        System.out.println("Method h() in Derived1Level1 called from " + s);
    }
    void k(String s) {
        System.out.println("Method k() in Derived1Level1 called from " + s);
    }
}
class Derived2Level1 extends BaseClass {
    public void f(String s) {
        System.out.println("Method f() in Derived2Level1 called from " + s);
//      h("Derived2Level1"); // h() has private access in basePackage.BaseClass.
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

```
//      k("Derived2Level1"); // k() is not public in basePackage.BaseClass;
                             // cannot be accessed from outside package.
    }
    protected void g(String s) {
        System.out.println("Method g() in Derived2Level1 called from " + s);
    }
}
class DerivedLevel2 extends Derived1Level1 {
    public void f(String s) {
        System.out.println("Method f() in DerivedLevel2 called from " + s);
        g("DerivedLevel2");
        h("DerivedLevel2");
        k("DerivedLevel2");
        super.f("DerivedLevel2");
     }
}
class TestInheritance {
    void run() {
        BaseClass bc = new BaseClass();
        Derived1Level1 d1l1 = new Derived1Level1();
        Derived2Level1 d2l1 = new Derived2Level1();
        DerivedLevel2 dl2 = new DerivedLevel2();
        bc.f("main(1)");
//      bc.g("main(2)");    // g() has protected access in basePackage.BaseClass.
//      bc.h("main(3)");    // h() has private access in basePackage.BaseClass.
//      bc.k("main(4)");    // k() is not public in basePackage.BaseClass;
                            // cannot be accessed from outside package.
        d1l1.f("main(5)");
//      d1l1.g("main(6)");  // g() has protected access in basePackage.BaseClass.
        d1l1.h("main(7)");
        d1l1.k("main(8)");
        d2l1.f("main(9)");
        d2l1.g("main(10)");
//      d2l1.h("main(11)"); // h() has private access in basePackage.BaseClass.
        dl2.f("main(12)");
//      dl2.g("main(13)");  // g() has protected access in basePackage.BaseClass.
        dl2.h("main(14)");
    }
    public static void main(String args[]) {
        (new TestInheritance()).run();
}
```

CENGAGEbrain.com

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

The execution of this code generates the following output:

```
Method f() in BaseClass called from main(1)
Method h() in BaseClass called from BaseClass
Method f() in Derived1Level1 called from main(5)
Method g() in BaseClass called from Derived1Level1
Method h() in Derived1Level1 called from main(7)
Method k() in Derived1Level1 called from main(8)
Method f() in Derived2Level1 called from main(9)
Method g() in Derived2Level1 called from main(10)
Method f() in DerivedLevel2 called from main(12)
Method g() in BaseClass called from DerivedLevel2
Method h() in Derived1Level1 called from DerivedLevel2
Method k() in Derived1Level1 called from DerivedLevel2
Method f() in Derived1Level1 called from DerivedLevel2
Method g() in BaseClass called from Derived1Level1
Method h() in Derived1Level1 called from main(14)
```

The class `BaseClass` is called a *base class* or a *superclass,* and other classes are called *subclasses* or *derived classes* because they are derived from the superclass in that they can use the data fields and methods specified in `BaseClass` as `protected`, `public`, or—when subclasses are in the same package as the base class—have no access modifier. They inherit all these fields and methods from their base class so that they do not have to repeat the same definitions. However, a derived class can override the definition of a non-`final` method by introducing its own definition. In this way, both the base class and the derived class have some measure of control over their methods.

The base class can decide which methods and data fields can be revealed to derived classes so that the principle of information hiding holds not only with respect to the user of the base class, but also to the derived classes. Moreover, the derived class can decide which public and protected methods and data fields to retain and use and which to modify. For example, both `Derived1Level1` and `Derived2Level1` redefine method `f()` by giving their own versions of `f()`. However, the access to the method with the same name in the parent class is still possible by preceding the method name with the keyword `super`, as shown in the call of `super.f()` from `f()` in `DerivedLevel2`.

A derived class can add new methods and fields of its own. Such a class can become a base class for other classes that can be derived from it so that the inheritance hierarchy can be deliberately extended. For example, the class `Derived1Level1` is derived from `BaseClass`, but at the same time, it is the base class for `DerivedLevel2`.

Protected methods or fields of the base class are accessible to derived classes. They are also accessible to nonderived classes if these classes are in the same package as the class that defines the protected methods and fields. For this reason, `Derived1Level1` can call `BaseClass`'s protected method `g()`, but a call to this method from `run()` in `TestInheritance` is rendered illegal.

However, `run()` can call method `g()`, declared protected in `Derived2Level1`, because both `Derived2Level1` and `TestInheritance` in which `run()` is defined are in the same package.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

Methods and data fields with no access modifier can be accessed by any class in the same package. For example, the method `k()` in `BaseClass` cannot be accessed even in a derived class, `Derived2Level1`, because the derived class is in a different package. But the method `k()` in `Derived1Level1` can be accessed even in a non-derived class, such as `TestInheritance` in `run()`, because both `Derived1Level1` and `TestInheritance` are in the same package.

Unlike C++, which supports multiple inheritance, inheritance in Java has to be limited to one class only so that it is not possible to declare a new class with the declaration

```
class Derived2Level2 extends Derived1Level1, Derived2Level1 { ... }
```

In addition, a class declared `final` cannot be extended (the wrapper classes are examples of `final` classes).

1.2.4 Polymorphism

Polymorphism refers to the ability of acquiring many forms. In the context of OOP, this means that the same method name denotes many methods that are members of different objects. This is accomplished by so-called *dynamic binding,* when the type of a method to be executed can be delayed until run time. This is distinguished from *static binding,* when the type of response is determined at compilation time, as in the case of the `IntObject` and `DoubleObject` presented in Section 1.2.1. Both of these objects are declared as objects whose storage fields hold data of type `Object` and not integer or double. The conversion is performed dynamically, but outside the object itself. For dynamic binding, consider the following declarations:

```
class A {
    public void process() {
        System.out.println("Inside A");
    }
}
class ExtA extends A {
    public void process() {
         System.out.println("Inside ExtA");
    }
}
```

then the code

```
    A object = new A();
    object.process();
    object = new ExtA();
    object.process();
```

results in the output

```
    Inside A
    Inside ExtA
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

This is due to dynamic binding: The system checks dynamically the type of object to which a variable is currently referring and chooses the method appropriate for this type. Thus, although the variable `object` is declared to be of type `A`, it is assigned in the second assignment an object of type `ExtA` and executes the method `process()`, which is defined in class `ExtA`, rather than the method by the same name defined in class `A`.

This is also true for interfaces. For example, if the declarations

```
interface B {
    void process();
}
class ImplB1 implements B {
    public void process() {
        System.out.println("Inside ImplB1");
    }
}
class ImplB2 implements B {
    public void process() {
        System.out.println("Inside ImplB2");
    }
}
```

CENGAGEbrain.com

are followed by the statements

```
    B object = new ImplB1();
    object.process();
    object = new ImplB2();
    object.process();
```

then the output is

```
    Inside ImplB1
    Inside ImplB2
```

notwithstanding the fact that `object` is of type `B`. The system recognizes that, for the first call of `process()`, `object` refers to an object of type `ImplB1`, and in the second call, it refers to an object of type `ImplB2`.

Polymorphism is thus a powerful tool in OOP. It is enough to send a standard message to many different objects without specifying how the message will be followed. There is no need to know of what type the objects are. The receiver is responsible for interpreting the message and following it. The sender does not have to modify the message depending on the type of receiver. There is no need for `switch` or `if-else` statements. Also, new units can be added to a complex program without the need of recompiling the entire program.

Dynamic binding allows for empowering the definition of `GenClass`. Assume that the definition of this class also includes a method for finding a position of a particular piece of information. If the information is not found, –1 is returned. The definition is now

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

```
class GenClass {
    Object[] storage = new Object[50];
    int find(Object el) {
        for (int i = 0; i < 50; i++)
            if (storage[i] != null && storage[i].equals(el))
                return i;
        return -1;
    }
    void store(Object el) {
        . . . . . . . . .
    }
    . . . . . . . . .
}
```

The method `find()` returns the correct result if wrappers of basic types are used—`Character()`, `Integer()`, and so on—but what happens if we want to store nonstandard objects in an instance of `GenClass`? Consider the following declaration

```
class SomeInfo {
    SomeInfo (int n) {
        this.n = n;
    }
    private int n;
}
```

CENGAGEbrain.com

Now the problem is, what happens if for a declaration

```
GenClass object = new GenClass();
```

we execute:

```
    object.store(new SomeInfo(17));
    System.out.println(object.find(new SomeInfo(17)));
```

As it turns out, –1 is printed to indicate an unsuccessful search. The result is caused by the method `equals()`. The system is using a built-in method for type `Object` that returns true if *references* of the compared variables are the same, not the *contents* of objects to which they refer. To overcome this limitation, the method `equals()` must be redefined by overriding the standard definition by a new definition. Therefore, the definition of `SomeInfo` is incomplete, and should be extended to

```
class SomeInfo {
    SomeInfo (int n) {
        this.n = n;
    }
    public boolean equals(Object si) {
        return n == ((SomeInfo) si).n;
    }
    private int n;
}
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

With this new definition, `find()` returns the position of the object holding number 17. The reason this works properly is that all classes are extensions of the class `Object`. This extension is done implicitly by the system so that the declaration of `SomeInfo` is really

```
class SomeInfo extends Object {
    . . . . . . . .
}
```

The qualifier `extends Object` is understood in the original definition and does not have to be made explicit. In this way, the standard method `equals()` is overridden by redefinition of this method in the class `SomeInfo` and by the power of dynamic binding. When executing the call `object.find(new SomeInfo(17))`, the system uses the method `equals()` defined in the class `SomeInfo` because an instance of this class is an argument in the method call. Thus, inside `find()`, the local variable becomes a reference to an object of type `SomeInfo`, although it is defined as a parameter of type `Object`. The problem is, however, that such a quick adjustment can be done only for the built-in methods for `Object`, in particular, `equals()` and `toString()`. Only slightly more complicated is the case when we want to define a generic class that makes comparisons possible. A more realistic example of polymorphism is given in the case study.

1.3 Input and Output

The `java.io` package provides several classes for reading and writing data. To use the classes, the package has to be explicitly included with the statement

```
import java.io.*;
```

In this section, we briefly introduce classes for reading from a standard device (keyboard), writing to a standard device (monitor), and processing I/O on files. There are also a number of other classes that are particularly important for interacting with the network, such as buffered, filtered, and piped streams.

To print anything on the screen, two statements are sufficient:

```
System.out.print(message);
System.out.println(message);
```

The two statements differ in that the second version outputs the end-of-line character after printing a message. The message printed by the statement is a string. The string can be composed of literal strings, string variables, and strings generated by the method `toString()` for a particular object; all components of the print statement are concatenated with the operator +. For example, having declared the class `C`:

```
class C {
    int i = 10;
    char a = 'A';
    public String toString() {
        return "(" + i + " " + a + ")";
    }
}
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

and an object

```
C obj = new C();
```

a printing statement

```
System.out.println("The object: " + obj);
```

outputs

```
The object: (10 A)
```

Note that if `toString()` were not defined in `C`, the output would be

```
The object: C@1cc789
```

because class `C` is by default an extension of the class `Object` whose method `toString()` prints the address of a particular object. Therefore, it is almost always critical that `toString()` is redefined in a user class to have a more meaningful output than an address.

Reading input is markedly more cumbersome. Data are read from standard input with the input stream `System.in`. To that end, the method `read()` can be used, which returns an integer. To read a line, the user must define a new method, for example,

CENGAGEbrain.com

```
public String readLine() {
    int ch;
    String s = "";
    while (true) {
        try {
            ch = System.in.read();
            if (ch == -1 || (char)ch == '\n') //end of file or end of line;
                break;
            else if ((char)ch != '\r')        // ignore carriage return;
                 s = s + (char)ch;
        } catch(IOException e) {
        }
    }
    return s;
}
```

Because `read()` is defined as a method that throws an `IOException`, the exception has to be handled with the `try-catch` clause.

Note that `ch` must be an integer to detect the end of file (which is Ctrl-z entered from the PC keyboard). The end-of-field marker is the number –1, and characters are really unsigned integers. If `ch` were declared as a character, then the assignment statement would have to be

```
ch = (char) System.in.read();
```

with which the end-of-line marker –1 would be stored as number 65535 in `ch`, whereby the subcondition `ch == –1` would be evaluated to false and the system would wait for further input.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

Fortunately, the task can be accomplished differently by first declaring an input stream with the declarations:

```
InputStreamReader cin = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(cin);
```

or with one declaration:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

and then a built-in method `readLine()` can be called to assign a value to a string `s`:

```
try {
       s = br.readLine();
} catch(IOException e) {
}
```

To read a number, an input must be read as a string and then converted into a number. For example, if input is in string `s`, then the conversion can be performed with

```
try {
     i = Integer.parseInt(s.trim());
} catch (NumberFormatException e) {
     System.out.println("Not a number");
}
```

CENGAGEbrain.com

Input can be simplified after `javax.swing.JOptionPane` is imported:

```
String s = JOptionPane.showInputDialog("Enter a number");
i = Integer.parseInt(s.trim());
```

To perform input and output on a file, a decision has to be made as to what type of data are to be processed so that the proper type of file can be elected.

A binary file is processed as a sequence of bytes, a text file as a sequence of characters. Generally, text files are portable; binary files are not because the data in the file have the same representation as data stored in main memory, and this representation varies from one system to another. However, Java binary files are platform independent.

The `java.io` package includes many streams for performing I/O that are organized in a hierarchy. The `InputStream` and `OutputStream` classes are used for processing binary files, and the `Reader` and `Writer` classes are for text files.

1.3.1 Reading and Writing Bytes

Consider a method that reads one byte at a time from an input file and writes it into an output file and to the screen:

```
void readBytes1(String fInName, String fOutName) throws IOException {
    FileInputStream fIn = new FileInputStream(fInName);
    FileOutputStream fOut = new FileOutputStream(fOutName);
    int i;
    while ((i = fIn.read()) != -1) {
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

```
        System.out.print((char)i + " "); // display characters;
    //  System.out.print(i + " ");       // display ASCII values;
        fOut.write(i);
    }
    fIn.close();
    fOut.close();
}
```

An integer variable is used to read the input, and the end of file is indicated by –1. To make reading and writing more efficient, input and output are buffered:

```
void readBytes2(String fInName, String fOutName) throws IOException {
    BufferedInputStream fIn = new BufferedInputStream(
                              new FileInputStream(fInName));
    BufferedOutputStream fOut = new BufferedOutputStream(
                                new FileOutputStream(fOutName));
    int i;
    while ((i = fIn.read()) != -1) {
        System.out.print(i + " ");
        fOut.write(i);
    }
    fIn.close();
    fOut.close();
}
```

CENGAGEbrain.com

1.3.2 Reading Lines

To read one line at a time, the method `readLine()` from `BufferedReader` is used, as in this example:

```
void readLines(String fInName, String fOutName) throws IOException {
    BufferedReader fIn = new BufferedReader(
                         new FileReader(fInName));
    PrintWriter fOut = new PrintWriter(new FileWriter(fOutName));
    String s;
    while ((s = fIn.readLine()) != null) {
        System.out.println(s);
        fOut.println(s);
    }
    fIn.close();
    fOut.close();
}
```

The end of file is detected upon reading a null string (after reading an empty line, the string `s` is not null, but is of length 0).

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

1.3.3 Reading Tokens: Words and Numbers

A `StreamTokenizer` extracts from a text file various types of tokens including identifiers (sequences of letters and digits that begin with a letter or a byte from the range '\u00A0' through '\u00FF') and numbers. It can also extract quoted strings and various comment styles when the tokenizer is properly set up. The `nextToken()` method skips space characters separating tokens and updates the tokenizer's instance variables: `sval` of type `String`, which contains the current token when it is a word; `nval` of type `double`, which contains the current token when it is a number; and `ttype` of type `int`, which contains the type of the current token. There are four types of tokens: `TT_EOF` (end of file), `TT_EOL` (end of line), `TT_WORD`, and `TT_NUMBER`. Here is an example:

```
void readTokens(String fInName) throws IOException {
    StreamTokenizer fIn = new StreamTokenizer(
                          new BufferedReader(
                          new FileReader(fInName)));
    fIn.nextToken();
    String s;
    while (fIn.ttype != StreamTokenizer.TT_EOF) {
        if (fIn.ttype == StreamTokenizer.TT_WORD)
             s = "word";
        else if (fIn.ttype == StreamTokenizer.TT_NUMBER)
             s = "number";
        else s = "other";
        System.out.println(s + ":\t" + fIn);
        fIn.nextToken();
    }
}
```

When a text file consists of words or numbers separated by blanks, then it may be easier to extract each word or number by reading one line at a time and then applying a string tokenizer, as in

```
void readTokens2(String fInName) throws IOException {
    BufferedReader fIn = new BufferedReader(
                         new FileReader(fInName));
    String s;
    while ((s = fIn.readLine()) != null) {
        java.util.StringTokenizer line = new java.util.StringTokenizer(s);
        while (line.hasMoreTokens())
            System.out.println(line.nextToken());
    }
    fIn.close();
}
```

Another example of applying a stream tokenizer can be found in Section 5.11.

CENGAGEbrain.com

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

1.3.4 Reading and Writing Primitive Data Types

The `DataInputStream` class provides methods for reading primitive data types in binary format. The methods include `readBoolean()`, `readByte()`, `readShort()`, `readChar()`, `readInt()`, `readLong()`, and `readUTF()` (to read strings in Unicode Text Format).

```
void writePrimitives(String fOutName) throws IOException {
    DataOutputStream fOut = new DataOutputStream(
                            new FileOutputStream(fOutName));
    fOut.writeBoolean(5<6);
    fOut.writeChar('A');
    fOut.writeDouble(1.2);
    fOut.writeFloat(3.4f);
    fOut.writeShort(56);
    fOut.writeInt(78);
    fOut.writeLong(90);
    fOut.writeByte('*');
    fOut.writeUTF("abc");
    fOut.close();
}
void readPrimitives(String fInName) throws IOException {
    DataInputStream fIn = new DataInputStream(
                          new FileInputStream(fInName));
    System.out.println(fIn.readBoolean() + " " +
                       fIn.readChar() + " " +
                       fIn.readDouble() + " " +
                       fIn.readFloat() + " " +
                       fIn.readShort() + " " +
                       fIn.readInt() + " " +
                       fIn.readLong() + " " +
                       fIn.readByte() + " " +
                       fIn.readUTF());
    fIn.close();
}
```

The case study at the end of the chapter relies on the I/O for primitive data types.

1.3.5 Reading and Writing Objects

Objects can also be saved in a file if they are made persistent. An object becomes persistent if its class type is stated to implement the `Serializable` interface, as in

```
class C implements Serializable {
    int i;
    char ch;
    C(int j, char c) {
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

```
                i = j; ch = c;
            }
            public String toString() {
                return "("+ i + " " + ch +")";
            }
        }
```

Such a declaration is possible if all instance variables are also `Serializable`, which is the case for class `C`, because in Java all basic data types, arrays, and many classes are `Serializable`. Here is an example of writing and reading an object of class type `C`:

```
void writeObjects(String fOutName) throws IOException {
    C c1 = new C(10,'A'), c2 = new C(20,'B');
    ObjectOutputStream fOut = new ObjectOutputStream(
                              new FileOutputStream(fOutName));
    fOut.writeObject(c1);
    fOut.writeObject(c2);
    fOut.close();
}
void readObjects(String fInName) throws IOException {
    C c1 = new C(30,'C'), c2 = c1;
    ObjectInputStream fIn = new ObjectInputStream(
                            new FileInputStream(fInName));
    try {
        c1 = (C)fIn.readObject();
        c2 = (C)fIn.readObject();
    } catch(ClassNotFoundException e) {
    }
    System.out.println(c1 + " " + c2);
}
```

CENGAGEbrain.com

1.3.6 Random Access Files

The files discussed thus far are processed sequentially: We read (write) one item at a time and proceed toward the end of the file. To be able to both read and write in the same file at any position in the file, a random access file should be used. A file is created with the constructor

`RandomAccessFile(`*name, mode*`);`

The constructor opens a file with the specified name either for reading only or for reading and writing. The mode is specified by either the letter *w* or the letters *rw;* for instance,

```
RandomAccessFile = raf new RandomAccessFile("myFile", "rw");
```

We can move anywhere in the file, but to know that we are within the file, we can use the method `length()` that returns the size of the file measured in bytes. The method `getFilePointer()` returns the current position in the file. The method `seek(pos)` moves the file pointer to the position specified by an integer `pos`.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

Reading is done by the method `read()`, which returns a byte as an integer; by `read(b)`, which fills entirely a byte array `b`; by `read(b,off,len)`, which fills `len` cells of the byte array `b` starting from cell `off`; and by `readLine()`, which reads one line of input. Other reading methods return a value specified by their names: `readBoolean()`, `readByte()`, `readShort()`, `readChar()`, `readInt()`, `readLong()`, and `readUTF()`. All of these reading methods have corresponding writing methods, for example, `write(c)`, where `c` is an `int`, `write(b)`, and `write(b,off,len)`, plus the method `writeBytes(s)` to write a string `s` as a sequence of bytes.

After file processing is finished, the file should be closed with the `close()` method:

```
raf.close();
```

Examples of the application of random access files can be found in the Case Study in Section 1.7.

1.4 Java and Pointers

In this section, a problem of implementing Java objects is analyzed.

Although Java does not use explicit pointers and does not allow the programmer to use them, object access is implemented in terms of pointers. An object occupies some memory space starting from a certain memory location. A pointer to this object is a variable that holds the address of the object, and this address is the starting position of the object in memory. In many languages, *pointer* is a technical term for a type of variable; therefore, the term is avoided in discussing Java programs and usually the term *reference* is used instead.

Consider the following declarations:

```
    class Node {
        String name;
        int age;
    }
```

With declarations

```
Node p = null, q = new Node("Bill",20);
```

two reference variables are created, `p` and `q`. The variable `p` is initialized to `null`. The pointer `null` does not point anywhere. It is not able to point to any object of any type; therefore, `null` is compatible with and can be assigned to a reference variable of any type. After execution of the assignment

```
p = null;
```

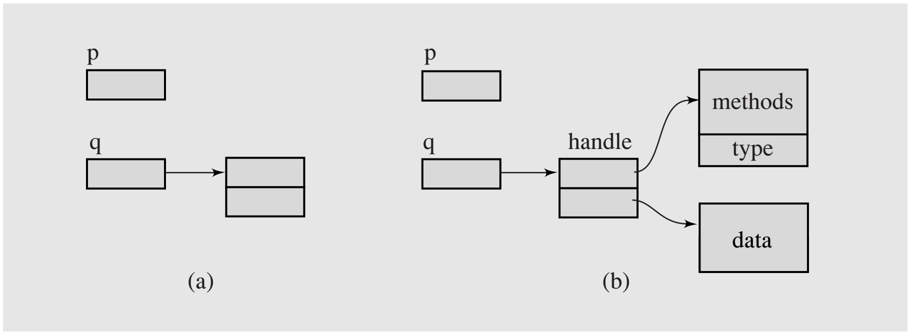
we may not say that `p` refers to `null` or points to `null`, but that `p` becomes `null` or `p` is `null`. The variable `p` is created to be used in the future as a reference to an object, but currently, it does not refer to any. The variable `q` is a reference to an object that is an instance of class `Node`. Forced by the built-in method `new`, the operating system through its memory manager allocates enough space for one unnamed object that can be

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

accessed, for now, only through the reference variable `q`. This reference is a pointer to the address of the memory chunk allocated for the object just created, as shown in Figure 1.1a. Figure 1.1a represents the logic of object reference, whose implementation can vary from one system to another and is usually much more intricate than the simple logic presented in this figure. For example, in Sun's implementation of Java, `q` refers to a handle that is a pair of pointers: one to the method table of the object and its type (which is a pointer to the class whose instance the object is) and the other to the object's data (Figure 1.1b). In Microsoft's implementation, `q` refers to the object's data, and the type and method table are pointed to by a hidden field of the object `q`. For simplicity, the subsequent illustrations use the form reflecting the logic of object reference, as in Figure 1.1a.

FIGURE 1.1 Object reference variables `p` and `q`: (a) logic of reference of `q` to an object; (b) implementation of this reference.



Keeping in mind how object access is implemented in Java helps explain the results of reference comparisons in Java. Consider the following code:

```
p = new Node("Bill",20);
System.out.print(p == q);
```

The printing statement outputs `false` because we compare references to two different objects; that is, we compare two different references (addresses), not the objects. To compare the objects' contents, their data fields have to be compared one by one using a method defined just for this reason. If `Node` includes the method

```
public boolean equals(Node n) {
    return name.equals(n.name) && age == n.age;
}
```

then the printing statement

```
System.out.print(p.equals(q));
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

outputs `true`. (This can be accomplished much more elegantly in C++ by overloading the equality operator ==; that is, by defining a method that allows for application of this operator to instances of class `Node`.)

The realization that object variables are really references to objects helps explain the need for caution with the use of the assignment operator. The intention of the declarations

```
Node node1 = new Node("Roger",20), node2 = node1;
```

is to create object `node1`, assign values to the two fields in `node1`, and then create object `node2` and initialize its fields to the same values as in `node1`. These objects are to be independent entities so that assigning values to one of them should not affect values in the other. However, after the assignments

```
node2.name = "Wendy";
node2.age  = 30;
```

the printing statement

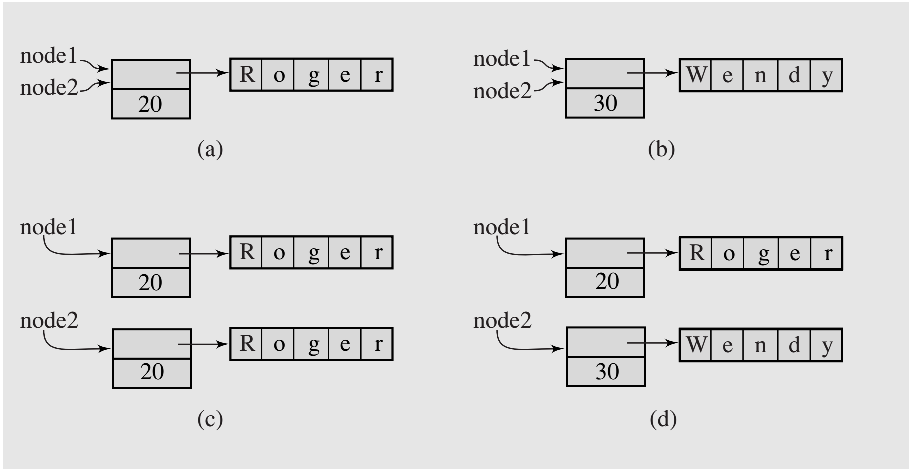
```
System.out.println(node1.name+" "+node1.age+" "+node2.name+" "+ node2.age);
```

generates the output

```
Wendy 30 Wendy 30
```

Both the ages and names in the two objects are the same. What happened? Because `node1` and `node2` are pointers, the declarations of `node1` and `node2` result in the situation illustrated in Figure 1.2a. After the assignments to the two fields of `node1`, the situation is as in Figure 1.2b. To prevent this from happening, we have to create a new copy of the object referenced by `node1` and then make `node2` become a

FIGURE 1.2 Illustrating the necessity of using the method `clone()`.



Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

reference to this copy. This can be done by defining the method `clone()` marked in the interface `Cloneable`. A new definition of `Node` is now:

```
class Node implements Cloneable {
    String name;
    int age;
    Node(String n, int a) {
        name = n; age = a;
    }
    Node() {
        this("",0);
    }
    public Object clone() {
        return new Node(name,age);
    }
    public boolean equals(Node n) {
        return name.equals(n.name) && age == n.age;
    }
}
```

With this definition, the declarations should be

```
Node node1 = new Node("Roger",20), node2 = (Node) node1.clone();
```

CENGAGEbrain.com

which results in the situation shown in Figure 1.2c, so that the two assignments

```
node2.name = "Wendy";
node2.age  = 30;
```

affect only the second object (Figure 1.2d).

The Java pointers are screened from the programmer. There is no pointer type in Java. The lack of an explicit pointer type is motivated by the desire to eliminate harmful behavior of programs. First, it is not possible in Java to have a nonnull reference to a nonexisting object. If a reference variable is not null, it always points to an object because the programmer cannot destroy an object referenced by a variable. An object can be destroyed in Pascal through the use of the function `dispose()` and in C++ through `delete`. The reason for using `dispose()` or `delete` is the need to return to the memory manager memory space occupied by an unneeded object. Directly after execution of `dispose()` or `delete`, pointer variables hold addresses of objects already returned to the memory manager. If these variables are not set to null or to the address of an object accessible from the program, the so-called *dangling reference problem* arises, which can lead to a program crash. In Java, the dangling reference problem does not arise. If a reference variable `p` changes its reference from one object to another, and the old object is not referenced by any other variable `q`, then the space occupied by the object is reclaimed automatically by the operating system through garbage collection (see Chapter 12). There is no equivalent in Java of `dispose()` or `delete`. Unneeded objects are simply abandoned and included in the pool of free memory cells automatically by the garbage collector during execution of the user program.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

Another reason for not having explicit pointers in Java is a constant danger of having a reference to a memory location that has nothing to do with the logic of the program. This would be possible through the pointer arithmetic that is not allowed in Java, where such assignments as

```
(p + q).ch = 'b';
(++p).n = 6;
```

are illegal.

Interestingly, although explicit pointers are absent in Java, Java relies on pointers more heavily than C/C++. An object declaration is always a declaration of reference to an object; therefore, an object declaration

```
Node p;
```

should be followed by initializing the variable `p` either by explicitly using a constructor, as in

```
p = new Node();
```

or by assigning a value from an already initialized variable, as in

```
p = q;
```

Because an array is also an object, the declaration

CENGAGEbrain.com

```
int a[10];
```

is illegal; this declaration is considered an attempt to define a variable whose name is `a[10]`. A declaration has to be followed with initialization, which is often combined with the declaration, as in

```
int[] a = new int[10];
```

In this way, Java does not allow for variables that name objects directly. Thus, the dot notation used to access fields of the object, as in `p.name`, is really an indirect reference to the field `name` because `p` is not the name of the object with field `name`, but a reference to (address of) this object. Fortunately, the programmer does not have to be very concerned about this distinction because it is all a matter of language implementation. But as mentioned, an understanding of these implementation details helps explain the results of some operations, as illustrated earlier by the operator ==.

1.5 Vectors in `java.util`

A useful class in the `java.util` package is the class `Vector` although it is considered today to be a legacy class. A vector is a data structure with a contiguous block of memory, just like an array. Because memory locations are contiguous, they can be randomly accessed so that the access time of any element of the vector is constant. Storage is managed automatically so that on an attempt to insert an element into a full vector, a larger memory block is allocated for the vector, the vector elements are copied to the new block, and the old block is released. Vector is thus a flexible array; that is, an array whose size can be dynamically changed.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

The class hierarchy in the package `java.util` is as follows:

`Object` ⇒ `AbstractCollection` ⇒ `AbstractList` ⇒ `Vector`

Figure 1.3 lists alphabetically the methods of class `Vector`. Some of these methods are inherited from `AbstractList`; others are from `AbstractCollection`. Figure 1.3 lists most of the methods of the class. Only methods `iterator()` and `listIterator()`, inherited from class `AbstractList`, and methods `finalize()`, `getClass()`, `notify()`, `notifyAll()`, and `wait()`, inherited from class `Object`, are not included.

FIGURE 1.3 An alphabetical list of member functions in the class `java.util.Vector`.

Method	**Operation**
`void add(Object ob)`	insert object `ob` at the end of the vector
`void add(int pos, Object ob)`	insert object `ob` at position `pos` after shifting elements at positions following `pos` by one position; throw `ArrayIndexOutOfBoundsException` if `pos` is out of range
`boolean addAll(Collection c)`	add all the elements from the collection `c` to the end of the vector; return `true` if `c` is not empty; throw `ArrayIndexOutOfBoundsException` if `pos` is out of range and `NullPointerException` if `c` is null
`boolean addAll(int pos, Collection c)`	add all the elements from the collection `c` at the position `pos` of the vector after shifting the objects following position `pos`; throw `ArrayIndexOutOfBoundsException` if `pos` is out of range and `NullPointerException` if `c` is null
`void addElement(Object ob)`	insert object `ob` at the end of the vector; same as `add(ob)`
`int capacity()`	return the number of objects that can be stored in the vector
`void clear()`	remove all the objects from the vector
`Object clone()`	return a clone of the vector
`boolean contains(Object ob)`	return `true` if the vector contains the object `ob`
`boolean containsAll (Collection c)`	return `true` if the vector contains all of the objects in the collection `c`; throw `NullPointerException` if `c` is null
`void copyInto(Object a[])`	copy objects from the vector to the object array `a`; throw `IndexOutOfBoundsException` if the array is not large enough to accommodate all objects from the vector and `NullPointerException` if `a` is null

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

FIGURE 1.3 *(continued)*

`Object elementAt(int pos)`	return the object at position `pos`; throw `ArrayIndexOutOfBoundsException` if `pos` is out of range; same as `get(pos)`
`Enumeration elements()`	return an `Enumeration` object that enumerates all the objects in the vector
`void ensureCapacity(int minCap)`	extend the size of the vector to accommodate at least `minCap` objects; do nothing if the size of the vector exceeds the minimum capacity `minCap`
`boolean equals(Object v)`	return `true` if the current vector and object `v` contain equal objects in the same order
`Object firstElement()`	return the first element in the vector; throw `NoSuchElementException` if the vector is empty
`Object get(int pos)`	return the object at position `pos`; throw `ArrayIndexOutOfBoundsException` if `pos` is out of range
`int hashCode()`	return the hash code for the vector
`int indexOf(Object ob)`	return the position of the first occurrence of object `ob` in the vector; return –1 if `ob` is not found
`int indexOf(Object ob, int pos)`	return the position of the first occurrence of object `ob` in the vector beginning the search at position `pos`; return –1 if `ob` is not found; throw `IndexOutOfRangeException` if `pos` < 0
`void insertElementAt(Object ob, int pos)`	insert object `ob` at position `pos` after shifting elements at positions following `pos` by one position; throw `ArrayIndexOutOfBoundsException` if `pos` is out of range; same as `add(ob,pos)`
`boolean isEmpty()`	return `true` if the vector contains no elements, `false` otherwise
`Object lastElement()`	return the last element in the vector; throw `NoSuchElementException` if the vector is empty
`int lastIndexOf(Object ob)`	return the position of the last occurrence of object `ob` in the vector; return –1 if `ob` is not found
`int lastIndexOf(Object ob, int pos)`	return the position of the last occurrence of object `ob` in the vector beginning the backward search at position `pos`; return –1 if `ob` is not found; throw `IndexOutOfRangeException` if `pos` is greater than or equal to the size of the vector
`boolean remove(Object ob)`	remove the first occurrence of `ob` in the vector and return `true` if `ob` was in the vector

Continues

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

FIGURE 1.3 *(continued)*

`Object remove(int pos)`	remove the object at position `pos`; throw `ArrayIndexOutOfBoundsException` if `pos` is out of range
`boolean removeAll(Collection c)`	remove from the vector all the objects contained in collection `c`; return `true` if any element was removed; throw `NullPointerException` if `c` is null
`boolean removeElement(Object ob)`	remove from the vector the first occurrence of `ob`; return `true` if an occurrence of `ob` was found; same as `remove (ob)`
`void removeElementAt(int pos)`	remove the object at position `pos`; throw `ArrayIndexOutOfBoundsException` if `pos` is out of range
`void removeAllElements()`	remove all the objects from the vector; same as `clear()`
`void removeRange(int first, int last)`	remove objects starting at position `first` and ending at `last`-1 and then shift all the succeeding objects to fill the hole (protected method)
`boolean retainAll(Collection c)`	remove from the vector all objects that are not in the collection `c`; return `true` if any object was removed; throw `ArrayIndexOutOfBoundsException` if `pos` is out of range
`Object set(int pos, Object ob)`	assign object `ob` to position `pos` and return the object that occupied this position before the assignment; throw `ArrayIndexOutOfBoundsException` if `pos` is out of range
`void setElementAt(Object ob, int pos)`	assign object `ob` to position `pos`; throw `ArrayIndexOutOfBoundsException` if `pos` is out of range
`void setSize(int sz)`	set size of the vector to `sz`; if current size is greater than `sz`, add new cells with null objects; if the current size is smaller than `sz`, discard the overflowing objects; throw `ArrayIndexOutOfBoundsException` if `sz` < 0
`int size()`	return the number of object in the vector
`List subList(int first, int last)`	return the sublist of the list (not its copy) containing elements from first to `last`-1; throw `ArrayIndexOutOfBoundsException` if either first or `last` is out of range and `IllegalArgumentException` if `last < first`
`Object[] toArray()`	copy all objects from the vector to a newly created array and return the array

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

FIGURE 1.3 *(continued)*

`Object[] toArray(Object a[])`	copy all objects from the vector to the array `a` if `a` is large enough or to a newly created array and return the array; throw `ArrayStoreException` if type of `a` is not a supertype of the type of every element in the vector and `NullPointerException` if `a` is null
`String toString()`	return a string representation of the vector that contains the string representation of all the objects
`void trimToSize()`	change the capacity of the vector to the number of objects currently stored in it
`Vector()`	construct an empty vector
`Vector(Collection c)`	construct a vector with objects copied from collection `c`; throw `NullPointerException` if `c` is null
`Vector(int initCap)`	construct a vector with the specified initial capacity; throw `IllegalArgumentException` if `initCap < 0`
`Vector(int initCap, int capIncr)`	construct a vector with the specified initial capacity and capacity increment

An application of these methods is illustrated in Figure 1.4. The contents of affected vectors are shown as comments on the line in which the methods are called. The contents of vectors are output with an implicit call to the method `toString()` in

```
System.out.println("v1 = " + v1);
```

but in the program in Figure 1.4, only one such line is shown.

To use the class `Vector`, the program has to include the `import` instruction

```
import java.util.Vector;
```

Vector `v1` is declared empty, and then new elements are inserted with the method `addElement()`. Adding a new element to a vector is usually fast unless the vector is full and has to be copied to a new block. But if the vector has some unused cells, it can accommodate a new element immediately in constant time.

The status of the vector can be tested with two methods: `size()`, which returns the number of elements currently in the vector, and `capacity()`, which returns the number of cells in the vector. If the vector's capacity is greater than its size, then a new element can be inserted at the end of the vector immediately. How frequently a vector is filled and has to be copied depends on the interplay between these two parameters, size and capacity, and the third parameter, capacity increment. By default, a new empty vector has capacity 10, and its capacity is doubled every time its size reaches the current capacity. For a large vector, this may lead to wasted space. For example, a full vector containing 50,000 elements has 100,000 cells after a new element arrives, but

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

FIGURE 1.4 A program demonstrating the operation of vector member functions.

```
import java.io.*;
import java.util.Vector;

class testVectors {
    public static void main(String a[]) {
        Vector v1 = new Vector();            // v1 = [], size = 0, capacity = 10
        for (int j = 1; j <= 5; j++)
            v1.addElement(new Integer(j));   // v1 = [1, 2, 3, 4, 5], size = 5,
                                             // capacity = 10
        System.out.println("v1 = " + v1);
        Integer i = new Integer(3);
        System.out.println(v1.indexOf(i) + " " + v1.indexOf(i,4));    // 2 -1
        System.out.println(v1.contains(i) + " " + v1.lastIndexOf(i)); // true 2
        Vector v2 = new Vector(3,4);         // v2 = [], size = 0, capacity = 3
        for (int j = 4; j <= 8; j++)
            v2.addElement(new Integer(j));   // v2 = [4, 5, 6, 7, 8], size = 5,
                                             // capacity = 7
        v2.ensureCapacity(9);                // v2 = [4, 5, 6, 7, 8], size = 5,
                                             // capacity = 11
        Vector v3 = new Vector(2);           // v3 = [], size = 0, capacity = 2
        v3.setSize(4);                       // v3 = [null, null, null, null],
                                             // size = cap = 4
        v3.setElementAt(new Integer(9),1);   // v3 = [null, null, null, 9]
        v3.setElementAt(new Integer(5),3);   // v3 = [null, 9, null, 5]
        v3.insertElementAt(v3.elementAt(3),1); // v3 = [null, 5, 9, null, 5],
                                             // size = 5, cap = 8
        v3.ensureCapacity(9);                // v3 = [null, 5, 9, null, 5],
                                             // size = 5, cap = 16
        v3.removeElement(new Integer(9));    // v3 = [null, 5, null, 5]
        v3.removeElementAt(v3.size()-2);     // v3 = [null, 5, 5]
        java.util.Enumeration ev = v3.elements();
        while (ev.hasMoreElements())
            System.out.print(ev.nextElement() + " ");
        System.out.println();
        v3.removeElementAt(0);               // v3 = [5, 5]
        v3.addAll(v1);                       // v3 = [5, 5, 1, 2, 3, 4, 5]
        v3.removeAll(v2);                    // v3 = [1, 2, 3] = v3 - v2
        v3.addAll(2,v1);                     // v3 = [1, 2, 1, 2, 3, 4, 5, 3]
        v3.retainAll(v2);                    // v3 = [4, 5] = intersection(v3,v2)
        v1.subList(1,3).clear();             // v1 = [1, 4, 5]
        Vector v4 = new Vector(), v5;
        v4.addElement(new Node("Jill",23));
        v5 = (Vector) v4.clone();            // v4 = [(Jill, 23)]
        ((Node)v5.firstElement()).age = 34; // v4 = v5 = [(Jill, 34)]
    }
}
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

the user may never include more elements than 50,001. In such a situation, the method `trimToSize()` should be used to reduce the waste.

When the user is reasonably sure of the maximum number of elements inserted in a vector, the method `ensureCapacity()` should be used to set capacity to the desired number so that all insertions are immediate. Otherwise, the user may set the capacity increment to a certain value so that when the vector is full, it is not doubled but increased by the capacity increment. Consider the declaration of vector `v2`:

```
Vector v2 = new Vector (3,4);
```

Initially, capacity is set to 3, and because the capacity increment equals 4, the capacity of the vector after inserting the fourth element equals 7. With this capacity, the statement

```
v2.ensureCapacity(9);
```

raises the capacity to 11 because it uses the capacity increment. Because the capacity increment for vector `v3` is not specified in its declaration, then when its capacity equals 8, the statement

```
v3.ensureCapacity(9);
```

causes the capacity to be doubled to 16.

The method `ensureCapacity()` affects only the capacity of the vector, not its content. The method `setSize()` affects its content and possibly the capacity. For example, the empty vector `v2` of capacity 2 changes to `v2 = [null, null, null, null]` after execution of

CENGAGEbrain.com

```
v2.setSize(4);
```

and its capacity equals 4.

The contents of `v2` are potentially dangerous if a method is executed that expects nonnull objects. For example, `v2.toString()` used in a printing statement raises `NullPointerException`. (To print a vector safely, a loop should be used in which `v.elementAt(i)` is printed.)

The method `addElement()` adds an element at the end of the vector. The insertion of an element in any other position can be performed with `insertElementAt()`. This reflects the fact that adding a new element inside the vector is a complex operation because it requires that all the elements are moved by one position to make room for the new element.

The method `elements()` puts vector elements in an object of `Enumeration` type. The loop shown in the program works the same for any data structure that returns an `Enumeration` object. In this way, data contained in different data structures become comparable by, as it were, equalizing the data structures themselves by using the common ground, the type `Enumeration`.

The method `clone()` should be used carefully. This method clones the array implementing the vector, but not the objects in the array. After the method is finished, the cloned vector includes references to the same objects as the vector from which it was cloned. In Figure 1.4, vector `v4` contains one object of type `Node` (as defined in Section 1.4), and then vector `v5`, a clone of `v4`, references the very same object from position 0. This is evident after the object is updated through reference `v5`; both `v4` and `v5` now reference the same updated object.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

1.6 Data Structures and Object-Oriented Programming

Although the computer operates on bits, we do not usually think in these terms; in fact, we would not like to. Although an integer is a sequence of 32 bits, we prefer seeing an integer as an entity with its own individuality, which is reflected in operations that can be performed on integers but not on variables of other types. As an integer uses bits as its building blocks, so other objects can use integers as their atomic elements. Some data types are already built into a particular language, but some data types can be, and need to be, defined by the user. New data types have a distinctive structure, a new configuration of their elements, and this structure determines the behavior of objects of these new types. The task given to the data structures domain is to explore such new structures and investigate their behavior in terms of time and space requirements. Unlike the object-oriented approach, where we start with behavior and then try to find the most suitable data type that allows for an efficient performance of desirable operations, we now start with a data type specification with some data structure and then look at what it can do, how it does it, and how efficiently. The data structures field is designed for building tools to be incorporated in and used by programs and for finding data structures that can perform certain operations speedily and without imposing too much burden on computer memory. This field is interested in building classes by concentrating on the mechanics of these classes, on their gears and cogs, which in most cases are not visible to the user of the classes. The data structures field investigates the operability of these classes and its improvement by modifying the data structures to be found inside the classes, because it has direct access to them. It sharpens tools and advises the user to what purposes they can be applied. Because of inheritance, the user can add some more operations to these classes and try to squeeze from them more than the class designer did.

The data structures field performs best if done in the object-oriented fashion. In this way, it can build the tools it intends without the danger that these tools will be inadvertently misused in the application. By encapsulating the data structures into a class and making public only what is necessary for proper usage of the class, the data structures field can develop tools whose functioning is not compromised by unnecessary tampering.

1.7 Case Study: Random Access File

From the perspective of the operating systems, files are collections of bytes, regardless of their contents. From the user's perspective, files are collections of words, numbers, data sequences, records, and so on. If the user wants to access the fifth word in a text file, a searching procedure goes sequentially through the file starting at position 0, and checks all of the bytes along the way. It counts the number of sequences of blank characters, and after it skips four such sequences (or five if a sequence of blanks begins the file), it stops because it encounters the beginning of the fifth nonblank sequence or the fifth word. This word can begin at any position of the file. It is impossible to go to a particular position of any text file and be certain that this is a starting position of the fifth word of the file. Ideally, we want to go directly to a certain position of the file and

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

be sure that the fifth word begins in it. The problem is caused by the lengths of the preceding words and sequences of blanks. If we know that each word occupies the same amount of space, then it is possible to go directly to the fifth word by going to the position 4·*length*(word). But because words are of various lengths, this can be accomplished by assigning the same number of bytes to each word; if a word is shorter, some padding characters are added to fill up the remaining space; if it is longer, then the word is trimmed. In this way, a new organization is imposed on the file. The file is now treated not merely as a collection of bytes, but as a collection of records; in our example, each record consists of one word. If a request comes to access the fifth word, the word can be directly accessed without looking at the preceding words. With the new organization, we created a random access file.

A random access file allows for direct access of each record. The records usually include more items than one word. The preceding example suggests one way of creating a random access file, namely, by using fixed-length records. Our task in this case study is to write a generic program that generates a random access file for any type of record. The workings of the program are illustrated for a file containing personal records, each record consisting of five fields (social security number, name, city, year of birth, and salary), and for a student file that stores student records. The latter records have the same fields as personal records, plus information about academic major. This allows us to illustrate inheritance.

In this case study, a generic random access file program inserts a new record into a file, finds a record in the file, and modifies a record. The name of the file has to be supplied by the user, and if the file is not found, it is created; otherwise, it is open for reading and writing. The program is shown in Figure 1.5.

The program uses a class `IOmethods` and the interface `DbObject`. A user-defined class that specifies one record in the database is the extension `IOmethods` of an implementation of `DbObject`.

The class `Database` is generic so that it can operate on any random access file. Its generic character relies on polymorphism. Consider the method `find()`, which determines whether a record is in the file. It performs the search sequentially, comparing each retrieved record `tmp` to the sought record `d` using the method `equals()` defined for the particular class (or rather, redefined because the method is inherited from the class `Object` from which any other class is derived). The object `d` is passed in as a parameter to `find()`. But `d` must not be changed because its value is needed for comparison. Therefore, another object is needed to read data from the file. This object is created with the method `copy()`, which takes a one-cell array as a parameter and assigns the reference to a copy of `d` created by `new` in `copy()` to the only cell of the array. If parameter `copy()` were of type `DbObject`, not `DbObject[]`, the reference would be discarded because the parameter would be passed by value. Now, the array is also passed by value, but its cell is changed permanently. The cell `tmp[0]` now contains a copy of `d`—in particular, its type—so that the system uses methods of the particular class; for example, `tmp[0].readFromFile()` is taken from class `Personal` if `d` is an object of this type, and from `Student` if `d` is a `Student` object.

The method `find()` uses to some extent the fact that the file is random by scrutinizing it record by record, not byte by byte. To be sure, the records are built out of bytes and all the bytes belonging to a particular record have to be read, but only the bytes required by the equality operator are participating in the comparison.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

FIGURE 1.5 Listing of a program to manage random access files.

```
//************************  DbObject.java  ************************

import java.io.*;

public interface DbObject {
    public void writeToFile(RandomAccessFile out) throws IOException;
    public void readFromFile(RandomAccessFile in) throws IOException;
    public void readFromConsole() throws IOException;
    public void writeLegibly() throws IOException;
    public void readKey() throws IOException;
    public void copy(DbObject[] db);
    public int size();
}

//************************  Personal.java  ************************

import java.io.*;

public class Personal extends IOmethods implements DbObject {
    protected final int nameLen = 10, cityLen = 10;
    protected String SSN, name, city;
    protected int year;
    protected long salary;
    protected final int size = 9*2 + nameLen*2 + cityLen*2 + 4 + 8;
    Personal() {
    }
    Personal(String ssn, String n, String c, int y, long s) {
        SSN = ssn; name = n; city = c; year = y; salary = s;
    }
    public int size() {
        return size;
    }
    public boolean equals(Object pr) {
        return SSN.equals(((Personal)pr).SSN);
    }
    public void writeToFile(RandomAccessFile out) throws IOException {
        writeString(SSN,out);
        writeString(name,out);
        writeString(city,out);
        out.writeInt(year);
        out.writeLong(salary);
    }
    public void writeLegibly() {
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

FIGURE 1.5 *(continued)*

```
        System.out.print("SSN = " + SSN + ", name = " + name.trim()
                + ", city = " + city.trim() + ", year = " + year
                + ", salary = " + salary);
    }
    public void readFromFile(RandomAccessFile in) throws IOException {
        SSN = readString(9,in);
        name = readString(nameLen,in);
        city = readString(cityLen,in);
        year = in.readInt();
        salary = in.readLong();
    }
    public void readKey() throws IOException {
        System.out.print("Enter SSN: ");
        SSN = readLine();
    }
    public void readFromConsole() throws IOException {
        System.out.print("Enter SSN: ");
        SSN = readLine();
        System.out.print("Name: ");
        name = readLine();
        for (int i = name.length(); i < nameLen; i++)
            name += ' ';
        System.out.print("City: ");
        city = readLine();
        for (int i = city.length(); i < cityLen; i++)
            city += ' ';
        System.out.print("Birthyear: ");
        year = Integer.valueOf(readLine().trim()).intValue();
        System.out.print("Salary: ");
        salary = Long.valueOf(readLine().trim()).longValue();
    }
    public void copy(DbObject[] d) {
        d[0] = new Personal(SSN,name,city,year,salary);
    }
}

//************************  Student.java  ************************

import java.io.*;

public class Student extends Personal {
    public int size() {
```

Continues

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

FIGURE 1.5 *(continued)*

```
        return super.size() + majorLen*2;
    }
    protected String major;
    protected final int majorLen = 10;
    Student() {
        super();
    }
    Student(String ssn, String n, String c, int y, long s, String m) {
        super(ssn,n,c,y,s);
        major = m;
    }
    public void writeToFile(RandomAccessFile out) throws IOException {
        super.writeToFile(out);
        writeString(major,out);
    }
    public void readFromFile(RandomAccessFile in) throws IOException {
        super.readFromFile(in);
        major = readString(majorLen,in);
    }
    public void readFromConsole() throws IOException {
        super.readFromConsole();
        System.out.print("Enter major: ");
        major = readLine();
        for (int i = major.length(); i < nameLen; i++)
            major += ' ';
    }
    public void writeLegibly() {
        super.writeLegibly();
        System.out.print(", major = " + major.trim());
    }
    public void copy(DbObject[] d) {
        d[0] = new Student(SSN,name,city,year,salary,major);
    }
}

//************************  Database.java  ************************

import java.io.*;

public class Database {
    private RandomAccessFile database;
    private String fName = new String();;
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

FIGURE 1.5 *(continued)*

```
    private IOmethods io = new IOmethods();
    Database() throws IOException {
        System.out.print("File name: ");
        fName = io.readLine();
    }
    private void add(DbObject d) throws IOException {
        database = new RandomAccessFile(fName,"rw");
        database.seek(database.length());
        d.writeToFile(database);
        database.close();
    }
    private void modify(DbObject d) throws IOException {
        DbObject[] tmp = new DbObject[1];
        d.copy(tmp);
        database = new RandomAccessFile(fName,"rw");
        while (database.getFilePointer() < database.length()) {
            tmp[0].readFromFile(database);
            if (tmp[0].equals(d)) {
                 tmp[0].readFromConsole();
                 database.seek(database.getFilePointer()-d.size());
                 tmp[0].writeToFile(database);
                 database.close();
                 return;
            }
        }
        database.close();
        System.out.println("The record to be modified is not in the
database");
    }
    private boolean find(DbObject d) throws IOException {
        DbObject[] tmp = new DbObject[1];
        d.copy(tmp);
        database = new RandomAccessFile(fName,"r");
        while (database.getFilePointer() < database.length()) {
            tmp[0].readFromFile(database);
            if (tmp[0].equals(d)) {
                database.close();
                return true;
            }
        }
        database.close();
```

Continues

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

FIGURE 1.5 *(continued)*

```
        return false;
    }
    private void printDb(DbObject d) throws IOException {
        database = new RandomAccessFile(fName,"r");
        while (database.getFilePointer() < database.length()) {
            d.readFromFile(database);
            d.writeLegibly();
            System.out.println();
        }
        database.close();
    }
    public void run(DbObject rec) throws IOException {
        String option;
        System.out.println("1. Add 2. Find 3. Modify a record; 4. Exit");
        System.out.print("Enter an option: ");
        option = io.readLine();
        while (true) {
            if (option.charAt(0) == '1') {
                 rec.readFromConsole();
                 add(rec);
            }
            else if (option.charAt(0) == '2') {
                 rec.readKey();
                 System.out.print("The record is ");
                 if (find(rec) == false)
                     System.out.print("not ");
                 System.out.println("in the database");
            }
            else if (option.charAt(0) == '3') {
                 rec.readKey();
                 modify(rec);
            }
            else if (option.charAt(0) != '4')
                 System.out.println("Wrong option");
            else return;
            printDb(rec);
            System.out.print("Enter an option: ");
            option = io.readLine();
        }
    }
}
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

FIGURE 1.5 *(continued)*

```
//***********************  UseDatabase.java  ***********************

import java.io.*;

public class UseDatabase {
    static public void main(String[]) throws IOException {
//        (new Database()).run(new Personal());
        (new Database()).run(new Student());
    }
}
```

The method `modify()` updates information stored in a particular record. The record is first retrieved from the file, also using sequential search, and the new information is read from the user using the method `readFromFile()` defined for a particular class. To store the updated record `tmp[0]` in the file, `modify()` forces the file pointer `database` to go back to the beginning of the record `tmp[0]` that has just been read; otherwise, the record following `tmp[0]` in the file would be overwritten. The starting position of `tmp` can be determined immediately because each record occupies the same number of bytes; therefore, it is enough to jump back the number of bytes occupied by one record. This is accomplished by calling `database.seek(database.getFilePointer()-d.size())`, where `size()` must be defined for the particular class.

The generic `Database` class includes two more methods. Method `add()` places a record at the end of file. Method `printDb()` prints the contents of the file.

To see the class `Database` in action, we have to define a specific class that specifies the format of one record in a random access file. As an example, we define the class `Personal` with five fields, `SSN`, `name`, `city`, `year`, and `salary`. The first three fields are strings, but only `SSN` is always of the same size. To have slightly more flexibility with the other two strings, two constants, `nameLen` and `cityLen`, are defined.

Storing data from one object requires particular care, which is the task of the method `writeToFile()`. The `SSN` field is the simplest to handle. A social security number always includes nine digits; therefore, the output operator << can be used. However, the lengths of names and cities vary from record to record, and yet the sections of a record in the data file designated for these two fields should always have the same length. To guarantee this, the method `readFromConsole()` adds trailing blanks to the strings.

Another problem is posed by the numerical fields, `year` and `salary`, particularly the latter field. If salary is written to the file with the method `printLong()`, then the salary 50,000 is written as a 5-byte-long string `'50000'`, and the salary

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

100,000 as a 6-byte-long string `'100000'`, which violates the condition that each record in the random access file should be of the same length. To avoid the problem, the numbers are stored in binary form. For example, 50,000 is represented in the field `salary` as a string of 32 bits, 00000000000000001100001101010000. We can now treat this sequence of bits as representing not a long number, but a string of four characters, 00000000, 00000000, 11000011, 01010000; that is, the characters whose ASCII codes are, in decimal, numbers 0, 0, 195, and 80. In this way, regardless of the value of salary, the value is always stored in 4 bytes. This is accomplished in Java with the method `writeLong()`.

This method of storing records in a data file poses a readability problem, particularly in the case of numbers. For example, 50,000 is stored as 4 bytes: two null characters, a special character, and a capital P. For a human reader, it is far from obvious that these characters represent 50,000. Therefore, a special routine is needed to output records in readable form. This is accomplished by using the method `writeLegibly()`, which explains why this program uses two methods for reading records and two for writing records: one is for maintaining data in a random access file, and the other is for reading and writing data in readable form.

To test the flexibility of the `Database` class, another user class is defined, class `Student`. This class is also used to show one more example of inheritance.

Class `Student` uses the same data fields as class `Personal` by being defined as a class derived from `Personal` plus one more field, a string field `major`. Processing input and output on objects of class type `Student` is very similar to that for class `Personal`, but the additional field has to be accounted for. This is done by redefining methods from the base class and at the same time reusing them. Consider the method `writeToFile()` for writing student records in a data file in fixed-length format:

```
public void writeToFile(RandomAccessFile out) throws IOException{
    super.writeToFile(out);
    writeString(major,out);
}
```

The method uses the base class's `writeTofile()` to initialize the five fields, `SSN`, `name`, `city`, `year`, and `salary`, and initializes the field `major`. Note that a special variable `super` must be used to indicate clearly that `writeToFile()` being defined for class `Student` calls `writeToFile()` already defined in base class `Personal`. However, class `Student` inherits without the modification method `readKey()` and the method `equals()`, because the same key is used in both `Personal` and `Student` objects to uniquely identify any record, namely, `SSN`.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

1.8 Exercises

1. What should be the type of constructors defined in classes?
2. Assume that classA includes a private variable k, a variable m with no modifier, a private protected variable n, a protected variable p, and a public variable q. Moreover, classB is derived from classA, classC is not derived from classA, and all three classes are in the same package. In addition, classD is derived from classA, classE is not derived from classA, and classA is in a different package than classD and classE. Which of the five variables defined in classA can be used by any of the four other classes?
3. What happens if the declaration of C:

```
class C {
    void process1(char ch) {
        System.out.println("Inside process1 in C " + ch);
    }
    void process2(char ch) {
         System.out.println("Inside process2 in C " + ch);
    }
    void process3(char ch) {
        System.out.println("Inside process3 in C " + ch);
        process2(ch);
    }
}
```

is followed by the following declaration of its extension:

```
class ExtC extends C {
    void process1(int n) {
        System.out.println("Inside process1 in ExtC " + n);
    }
    void process2(char ch) {
        System.out.println("Inside process2 in ExtC " + ch);
    }
    void process4(int n) {
        System.out.println("Inside process4 in Ext C " + n);
    }
}
```

Which methods are invoked if the declaration of three objects

```
ExtC object1 = new ExtC();
C object2 = new ExtC(), object3 = new ExtC();
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

is followed by these statements; indicate any problems that these statements may cause:

```
object1.process1(1000);
object1.process4(2000);
object2.process1(3000);
object2.process4(4000);
object3.process1('P');
object3.process2('Q');
object3.process3('R');
```

4. For the declaration of `I1`:

```
interface I1 {
    int I1f1();
    void I1f2(int i);
}
```

identify the errors.

a. in the declaration of the interface `I2`:

```
interface I2 extends I1 {
    double I2f1();
    void I2f2(int i);
    int I1f1();
    double I2f1() { return 10; }
    private int AC1f4();
    private int n = 10;
}
```

b. in the declaration of class `CI1`:

```
class CI1 implements I1 {
    int I1f1() { . . . . . }
    void I1f2(int i) { . . . . . }
    int CI1f3() { . . . . . }
}
```

c. and in the declaration of object `c6`:

```
I1 c6 = new I1();
```

5. Identify the errors:

a.
```
abstract class AC1 {
    int AC1f1() { . . . . }
    void AC1f2(int i) { . . . . }
    int AC1f3();
}
```

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

b. `interface C6 extends CAC1 { . . . }`
   where `CAC1` is a class.

c. `class CAC1AC2 extends AC1, AC2 { . . . }`
   where `AC1` and `AC2` are two abstract classes.

d. `AC1 c7 = new AC1();`
   where `AC1` is an abstract class.

6. What happens if the class `SomeInfo` instead of the definition of `equals()` from Section 1.2.4 uses the following definition of this method:

```
public boolean equals(SomeInfo si) {
    return n == si.n;
}
```

1.9 PROGRAMMING ASSIGNMENTS

1. Write a `Fraction` class that defines adding, subtracting, multiplying, and dividing fractions. Then write a method for reducing factors and methods for inputting and outputting fractions.

2. Write a class `Quaternion` that defines the four basic operations of quaternions and the two I/O operations. Quaternions, as defined in 1843 by William Hamilton and published in his *Lectures on Quaternions* in 1853, are an extension of complex numbers. Quaternions are quadruples of real numbers, $(a,b,c,d) = a + bi + cj + dk$, where $1 = (1,0,0,0)$, $i = (0,1,0,0)$, $j = (0,0,1,0)$, and $k = (0,0,0,1)$ and the following equations hold:

$$i^2 = j^2 = k^2 = -1$$
$$ij = k,\ jk = i,\ ki = j,\ ji = -k,\ kj = -i,\ ik = -j$$
$$(a + bi + cj + dk) + (p + qi + rj + sk)$$
$$= (a + p) + (b + q)i + (c + r)j + (d + s)k$$
$$(a + bi + cj + dk) \cdot (p + qi + rj + sk)$$
$$= (ap - bq - cr - ds) + (aq + bp + cs - dr)i$$
$$+ (ar + cp + dq - bs)j + (as + dp + br - cq)k.$$

Use these equations in implementing a quaternion class.

3. Write a program to reconstruct a text from a concordance of words. This was a real problem of reconstructing some unpublished texts of the Dead Sea Scrolls using concordances. For example, here is William Wordsworth's poem, *Nature and the Poet,* and a concordance of words corresponding with the poem.

So pure the sky, so quiet was the air!
So like, so very like, was day to day!
Whene'er I look'd, thy image still was there;
It trembled, but it never pass'd away.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

Licensed to: iChapters User

The 33-word concordance is as follows:

1:1 so quiet was the *air!
1:4 but it never pass'd *away.
1:4 It trembled, *but it never
1:2 was *day to day!
1:2 was day to *day!
1:3 thy *image still was there;
. . . .
1:2 so very like, *was day
1:3 thy image still *was there;
1:3 *Whene'er I look'd,

In this concordance, each word is shown in context of up to five words, and the word referred to on each line is preceded with an asterisk. For larger concordances, two numbers have to be included, a number corresponding with a poem and a number of the line where the words can be found. For example, assuming that 1 is the number of *Nature and the Poet,* line "1:4 but it never pass'd *away." means that the word "away" is found in this poem in line 4. Note that punctuation marks are included in the context.

Write a program that loads a concordance from a file and creates a vector where each cell is associated with one line of the concordance. Then, using a binary search, reconstruct the text.

4. Modify the program from the case study by maintaining an order during insertion of new records into the data file. This requires defining the method `compareTo()` in `Personal` and in `Student` to be used in a modified method `add()` in `Database`. The method finds a proper position for a record `d`, moves all the records in the file to make room for `d`, and writes `d` into the file. With the new organization of the data file, `find()` and `modify()` can also be modified. For example, `find()` stops sequential search when it encounters a record greater than the record looked for (or reaches the end of file). A more efficient strategy can use binary search, discussed in Section 2.7.

5. Write a program that maintains an order in the data file indirectly. Use a vector of file position pointers (obtained through `getFilePointer()`) and keep the vector in sorted order without changing the order of records in the file.

6. Modify the program from the case study to remove records from the data file. Define method `isNull()` in classes `Personal` and `Student` to determine that a record is null. Define also method `writeNullToFile()` in the two classes to overwrite a record to be deleted by a null record. A null record can be defined as having a nonnumeric character (a tombstone) in the first position of the `SSN` field. Then define method `remove()` in `Database` (very similar to `modify()`), which locates the position of a record to be deleted and overwrites it with the null record. After a session is finished, a `Database` method `purge()` destructor should be invoked which copies nonnull records to a new data file, deletes the old data file, and renames the new data file with the name of the old data file.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.

BIBLIOGRAPHY

Object-Oriented Programming

Cardelli, Luca, and Wegner, Peter, "On Understanding Types, Data Abstraction, and Polymorphism," *Computing Surveys* 17 (1985), 471–522.

Ege, Raimund K., *Programming in an Object-Oriented Environment,* San Diego: Academic Press, 1992.

Khoshafian, Setrag, and Razmik, Abnous, *Object Orientation: Concepts, Languages, Databases, User Interfaces,* New York: Wiley, 1995.

Meyer, Bertrand, *Object-Oriented Software Construction,* Upper Saddle River, NJ: Prentice Hall, 1997.

Java

Dunn, Douglas, *Java Rules,* Reading, MA: Addison-Wesley, 2002.

Gittleman, Art, *Computing with Java,* El Granada, CA: Scott/Jones, 2002.

Gosling, James, Joy, Bill, and Bracha, Silad, *The Java Language Specification,* Boston: Addison-Wesley, 2000.

Naughton, Patrick, and Schildt, Herbert, *Java 2: The Complete Reference,* Berkeley, CA: Osborne McGraw-Hill, 1999.

Weber, Joe (ed.), *Using Java 2 Platform,* Indianapolis: Que, 1999.

Weiss, Mark A., *Data Structures and Problem Solving Using Java,* Reading, MA: Addison-Wesley, 2001.

Copyright 2010 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part.