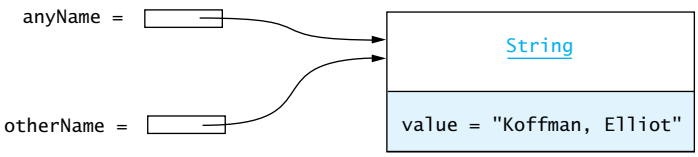
EXAMPLE A.8 If you execute the statement

```
String otherName = anyName;
```

the variables `anyName` and `otherName` reference the same `String` object:



All of the following conditions are then `true`.

```
(anyName == otherName)
(anyName.equals(otherName))
(anyName.compareTo(otherName) == 0)
(anyName.equalsIgnoreCase(otherName))
```

If they had referenced different strings with the same contents, only the first condition would be **`false`** because of the address comparison. If they referenced different strings that contained the same words but one's contents were in uppercase and the other's contents were in lowercase, only the last condition would be **`true`**.

The `compareTo` and `compareToIgnoreCase` operators return negative or positive values according to whether the argument, in dictionary order, follows or precedes the string to which the method is applied. If keyboard contains the string `"qwerty"`, the expression

```
keyboard.compareTo("rest")
```

is negative because `"r"` follows `"q"`. For the same reason, the expression

```
"rest".compareTo(keyboard)
```

is positive.

The `compareTo` method performs a case-sensitive comparison, in which all of the uppercase letters precede all of the lowercase letters. If keyboard (containing `"qwerty"`) is compared with `"Rest"`, the results are the opposite of what we have just shown for comparing keyboard with `"rest"`: `keyboard.compareTo("Rest")` is positive because `"R"` precedes `"q"`, and `"Rest". compareTo(keyboard)` is negative.

To compare the contents of two strings in alphabetical order regardless of case, use the `compareToIgnoreCase` method: the expressions

```
keyboard.compareToIgnoreCase("rest")
keyboard.compareToIgnoreCase("Rest")
```

are both negative because `"r"` follows `"q"`.

The `String.format` Method

Java uses a default format for converting the primitive types to `Strings`. This default formatting is applied when you output a primitive value using `System.out.print` or `System.out.println`. For example, the output lines displayed by the statement

```
System.out.println(n + "\t" + Math.sqrt(n));
```

for `n = 1` and `n = 2` are:

```
1     1.0
2     1.4142135623730951
```

Note that the numbers in each column are left-justified and that a large number of significant digits is shown for the square root of 2, but only one zero is shown for the square root of 1.

The `Formatter` class and the `String.format` method give us better control over the formatting of numeric values.

Using the `String.format` method, we can rewrite the earlier `println` statement as

```
System.out.println(String.format("%2d%10.2f", n, Math.sqrt(n)));
```

Now the `format` method is called to build a formatted output string before `println` executes. This statement displays the following output lines for `n = 1` and `n = 2`:

```
1       1.00
2       1.41
```

The `format` method takes a variable number of arguments. The first argument is a format string that specifies how the output string should be formed. The format string above contains a sequence of two format codes, `%2d` and `%10.2f`. Each format code describes how its corresponding argument should be formatted.

A format code begins with a `%` character and is optionally followed by an integer for width, a decimal point and an integer for precision (optional), and a type conversion specification (e.g., `d` for integer, `f` for real number, and `s` for string). The format code `%2d` means use two characters to represent the integer value of its corresponding argument (`n`); the format code `%10.2f` means use a total of 10 characters and 2 decimal places of precision to represent the real value of its corresponding argument (`Math.sqrt(n)`).

The width specifier gives the minimum number of characters that are used to represent a value. If more are required, then they will be inserted, but if fewer are required, then leading spaces are used to fill the character count (to achieve right-justification). If the width specifier is negative, then trailing spaces are used to fill the character count (to achieve left-justification). If the width specifier is omitted, then the exact number of characters required to represent the value with the prescribed precision will be used.

The precision specifier (e.g., `.2`) is optional and applies only to the `f`-type conversion specification. It indicates the number of digits following the decimal point. If omitted, six digits are displayed following the decimal point.

You can also have characters other than format codes inside a format string. The arguments to be formatted are inserted in the formatted string exactly where their format specifiers appear. For example, when `n` is 2, the statement

```
String.format("Value of square root of %d is %.3f", n, Math.sqrt(n))
```

creates the `String`

```
Value of square root of 2 is 1.414
```

The value 2 and the square root of 2 replace their format specifiers in the formatted string. Because the width specifiers are omitted, the exact number of characters required to represent the values to the desired precision is used.

Other format conversion characters used in this text are `s` (for string) and `n` for newline. The format code `%10s` causes its corresponding string argument to be formatted using 10 characters. Like numbers, strings are displayed right-justified. The format code `%-10s` will cause the string to be displayed left-justified. The format code `%n` will cause an operating system-specific newline sequence to be generated. On some operating systems, the newline is indicated by the `\n` character, or by the sequence `\r\n`, and on others by `\r`. The `println` method always terminates its output with the correct sequence for the operating system on which the program is executing. Using `%n` achieves the same result when using method `format`.

The `Formatter` Class

You can also use the `java.util.Formatter` class to create `Formatter` objects for writing formatted output to the console (or elsewhere). The statement

```
Formatter fOut = new Formatter(System.out);
```

creates a Formatter object fOut associated with the console. The statements

```
fOut.format("%2d%10.2f\n", 1, Math.sqrt(1));
fOut.format("%2d%10.2f\n", 2, Math.sqrt(2));
```

write to object fOut the pair of formatted strings shown earlier, each ending with a newline character (\n). Each string written to fOut will be displayed in the console window. You can actually apply the format method or the new printf method directly to System.out without wrapping System.out in a Formatter object.

```
System.out.printf("%2d%10.2f\n", 1, Math.sqrt(1));
System.out.format("%2d%10.2f\n", 2, Math.sqrt(2));
```

The String.split Method

Often we want to process individual pieces, or tokens, in a string. For example, in the string "Doe, John 5/15/65", we are likely to be interested in one or more of the particular pieces "Doe", "John", "5", "15", and "65". These pieces would have to be extracted from the string as *tokens*. You can retrieve tokens from a String object using the String.split method.

Introduction to Regular Expressions

The argument to the split method is a special kind of string known as a *regular expression*. A regular expression is a string that describes another string or family of strings.

The simplest regular expression is a string that does not include any special characters, which matches itself. For example, the string " " represents a single space, and the string ", " represents a comma followed by a space. For the statements

```
String personData = "Doe, John 5/15/65";
String[] tokens = personData.split(", ");
```

tokens is declared to be an array of references to String objects. The first String reference is stored in tokens[0]. The argument of split, the string ", ", matches the two characters following the letter e, so tokens[0] is "Doe" and tokens[1] is "John 5/15/65". This is not quite what we desire, so tokens[1] needs to be split further. The string personData is not changed by this operation. The character sequence comma followed by space is often called a *delimiter* because it separates the tokens.

Matching One of a Group of Characters

A string enclosed in brackets ([and]) matches any one of the characters in the string, unless the first character in the string is ^, in which case the match is to any character not in the string that follows the ^. For example, the string "[, /]" will match a comma, a space, or a slash, and the string "[^abc]" will match any character that is not a, b, or c. To match a range of characters, separate the start and end character of the range with a '-'. For example, "[a-z]" will match any lowercase letter. For the statement

```
tokens = personData.split("[, /]");
```

tokens[0] is "Doe", tokens[1] is an empty string because the space character in personData is matched immediately by the space in the delimiter string, tokens[2] is "John", tokens[3] is "5", tokens[4] is "15", and tokens[5] is "65", so we are closer to what we desire.

Qualifiers

Character groups match a single character. Qualifiers are applied to regular expressions to define a new regular expression that conditionally performs a match. These qualifiers are shown in Table A.8.

TABLE A.8
Regular Expression Qualifiers

Qualifier	Meaning
`X?`	Optionally matches the regular expression *X*
`X*`	Matches zero or more occurrences of regular expression *X*
`X+`	Matches one or more occurrences of regular expression *X*

In the statement

```
tokens = personData.split("[, /]+");
```

the argument `"[, /]+"` will match a string of one or more space, comma, and slash characters. Therefore, `tokens[0]` is `"Doe"`, `tokens[1]` is `"John"`, `tokens[2]` is `"5"`, and so on. This is what we desire.

PITFALL

Not Using the + Qualifier to Define a Delimiter Regular Expression

As we explained above, if we had omitted the + qualifier from the delimiter string `"[, /]+"`, there would have been an empty string in the array of tokens. The reason for this is that the comma after `Doe` would match the comma in the delimiter. The split method would then save `"Doe"` in `tokens[0]` and search for another match to the delimiter. It would immediately find the space, and since there were no characters between the previously found delimiter and this one, an empty string would be stored. Using the + qualifier ensures that the comma and space in `personData` are treated as a single delimiter, not as two separate delimiters.

Defined Character Groups

Several character groups are defined and are indicated by a letter preceded by two backslash characters. The defined groups are shown in Table A.9.

TABLE A.9
Defined Character Groups

Regular Expression	Equivalent Regular Expression	Description
`\\d`	`[0-9]`	A digit
`\\D`	`[^0-9]`	Not a digit
`\\s`	`[ \t\n\x0B\f\r]`	A whitespace character (space, tab, newline, vertical tab, formfeed, or return)
`\\S`	`[^\s]`	Not a whitespace character
`\\w`	`[a-zA-Z_0-9]`	A word character (letter, underscore, or digit)
`\\W`	`[^\w]`	Not a word character

EXAMPLE A.9 We want to extract the symbols from an expression. We define a symbol as a string of letter or digit characters. The symbols are separated by one or more whitespace characters. The statement

```
String[] symbols = expression.split("\\s+");
```

will split the string `expression` into an array of symbols separated by whitespace characters.

EXAMPLE A.10 We want to extract the words from a text. We define a word as a string of letter or digit characters. The characters can be in any language. Thus, the delimiters are any string that consists of one or more characters that are not letters or digits (e.g., whitespace, punctuation symbols, and parentheses). The regular expression `"[^\\P{L}\\P{N}]+"` represents a string consisting of one or more characters that are not letters or digits. The statement

```
String[] words = line.split("[^\\p{L}\\p{N}]+");
```

will split the string `line` consisting of letters, digits, special characters, and punctuation symbols into separate elements of array `words`. The meaning of `\\p{L}` and `\\p{N}` is discussed next.

Unicode Character Class Support

The groups shown in Table A.9 apply only to the first 128 Unicode characters, which is adequate for processing English text. However, Java uses the Unicode characters that can be used to represent languages other than English. In these other languages, a–z do not represent all of the letters or may not be letters at all. For example, French includes the letters à, á, and â, which are distinct from a. Greek uses characters such as α, β, and γ. Selected character groups based on the Unicode character category are shown in Table A.10.

The `StringBuilder` and `StringBuffer` Classes

Java provides a class called `StringBuilder` that, like `String`, also stores character sequences. However, unlike a `String` object, the contents of a `StringBuilder` object can be changed. Use a `StringBuilder` object to store a string that you plan to change; otherwise, use a `String` object to store that string. Table A.11 describes the methods of class `StringBuilder`. In Table A.11, "this `StringBuilder`" means the `StringBuilder` object to which the method is applied through the dot notation. Methods `append`, `delete`, `insert`, and `replace` modify this `StringBuilder` object.

TABLE A.10
Regular Expressions for Selected Unicode Character Categories

Regular Expression	Description
`\\p{L}`	Letter
`\\p{Lu}`	Uppercase letter
`\\p{Ll}`	Lowercase letter
`\\p{Lt}`	Titlecase letter
`\\p{N}`	Numbers
`\\p{P}`	Punctuation
`\\p{S}`	Symbols
`\\p{Zs}`	Spaces

TABLE A.11
`StringBuilder` Methods in `java.lang.StringBuilder`

Method	Behavior
`StringBuilder append(`*anyType*`)`	Appends the string representation of the argument to this `StringBuilder`. The argument can be of any data type
`int capacity()`	Returns the current capacity of this `StringBuilder`
`StringBuilder delete(int start, int end)`	Removes the characters in a substring of this `StringBuilder`, starting at position `start` and ending with the character at position `end - 1`
`StringBuilder insert(int offset,` *anyType* `data)`	Inserts the argument `data` (any data type) into this `StringBuilder` at position `offset`, shifting the characters that started at `offset` to the right
`int length()`	Returns the length (character count) of this `StringBuilder`
`StringBuilder replace(int start, int end, String str)`	Replaces the characters in a substring of this `StringBuilder` (from position `start` through position `end - 1`) with characters in the argument `str`. Returns this `StringBuilder`
`String substring(int start)`	Returns a new string containing the substring that begins at the specified index `start` and extends to the end of this `StringBuilder`
`String substring(int start, int end)`	Return a new string containing the substring in this `StringBuilder` from position `start` through position `end - 1`
`String toString()`	Returns a new string that contains the same characters as this `StringBuilder` object

The `StringBuilder` class was introduced as a replacement for the `StringBuffer`. The `StringBuffer` has the same methods as the `StringBuilder`, but is designed for programs that have multiple threads of execution. All programs presented in this text are single-thread.

EXAMPLE A.11 The following statements declare three `StringBuilder` objects using three different constructors. The default capacity of an empty `StringBuilder` object is 16 characters. The capacity of a `StringBuilder` object is automatically doubled as required to accommodate the character sequence that is stored.

```
var sB1 = new StringBuilder();         // Capacity is 16
var sB2 = new StringBuilder(30);       // Capacity is 30
var sB3 = new StringBuilder("happy"); // Stores "happy"
                                       // Capacity 16
```

The following statements result in the character sequence `"happy birthday to you"` being stored in `sB3`.

```
sB3.append("day me");       // "happyday me"
sB3.insert(9, "to ");       // "happyday to me"
sB3.insert(5, " birth");    // "happy birthday to me"
sB3.replace(18, 20, "you"); // "happy birthday to you"
```

PITFALL

String Index Out of Bounds

If an index supplied to any `String`, `StringBuilder`, or `StringBuffer` method is outside the valid range of character positions for the string object (i.e., if the index is less than 0 or greater than or equal to the string length), a `StringIndexOutOfBoundsException` will occur. This is a run-time error and will terminate program execution. We will discuss exceptions in more detail in Section A.11.

StringJoiner Class

Assume you have the array of `Strings`, `names`, as shown below:

names	string
[0]	"Tom"
[1]	"Dick"
[2]	"Harry"

You can format this into the `String` `"Tom, Dick, Harry"` using a `StringBuilder`:

```
var stb = new StringBuilder();
stb.append(names[0]);
for (int i = 1; i < names.length; i++) {
    stb.append(", ");
    stb.append(names[i]);
}
String result = stb.toString();
```

Note that you needed to apply special handling for the first element in the array, and then start the loop at 1. The `StringJoiner` class was introduced in Java 8 to construct a sequence of characters separated by a delimiter. It also provides for an optional `prefix` and `suffix`. You can construct the string `"Tom, Dick, Harry"` with less effort using a `StringJoiner`:

```
var sj = new StringJoiner(", ");
for (int i = 0; i < names.length; i++) {
    sj.add(names[i]);
}
String result = sj.toString();
```

The first line creates a new `StringJoiner` `sj` and specifies that the string `", "` will be used as the delimiter. In the loop, each element of array `names` is appended to `sj`.

Table A.12 describes the methods of class `StringJoiner`. The interface `CharSequence` is implemented by the `String`, `StringBuffer`, and `StringBuilder` classes, so objects of these classes may be used as arguments that are of type `CharSequence`.

TABLE A.12
`StringJoiner` Methods in `java.util.StringJoiner`

Method	Behavior
`StringJoiner(CharSequence delimiter)`	Constructs an empty `StringJoiner` with the provided `delimiter` and no prefix or suffix
`StringJoiner(CharSequence delimiter, CharSequence prefix, CharSequence suffix)`	Constructs an empty `StringJoiner` with the provided `delimiter`, `prefix`, and `suffix`
`StringJoiner add(CharSequence newElement)`	Adds a copy of the given input to the `StringJoiner`
`int length()`	Returns the length of the resulting `String`
`StringJoiner merge(StringJoiner other)`	Adds the contents of the other `StringJoiner` to this `StringJoiner`. The other `StringJoiner's delimiter`, `prefix`, and `suffix` are not copied
`StringJoiner setEmptyValue(CharSequence emptyValue)`	Sets the `emptyValue` to be returned by the `toString` method if no elements have been added via the `add` method
`String toString()`	Returns a `String` consisting of the `prefix`, followed by the contents with each element separated by the `delimiter` and followed by the `suffix`. If no contents were added via a call to `add`, then a `String` consisting of the `prefix` followed by the `suffix` is returned, unless an empty value has been set

EXERCISES FOR SECTION A.5

SELF-CHECK

1. Evaluate each of these expressions.
```
"happy".equals("Happy")
"happy".compareTo("Happy")
"happy".equalsIgnoreCase("Happy")
"happy".equals("happy".charAt(0) + "Happy".substring(1))
"happy" == "happy".charAt(0) + "Happy".substring(1)
```
2. You want to extract the words in the string `"Nancy* has thirty-three*** fine!! teeth."` using the `split` method. What are the delimiter characters, and what should you use as the argument string?
3. Rewrite the following statements using `StringBuilder` objects:
```
String myName = "Elliot Koffman";
String myNameFirstLast = myName;
myName = myName.substring(7) + ", " + myName.substring(0, 6);
```
4. Rewrite the statements of Exercise 3 using a `StringJoiner` object.
5. What is stored in `result` after the following statements execute?
```
StringBuilder result = new StringBuilder();
String sentence = "Let's all learn how to program in Java";
String[] tokens = sentence.split(\\s+);
for (int i = 0; i < tokens.length; i++) {
    result.append(tokens[i]);
 }
```
6. Revise Exercise 5 to insert a newline character between the words in `result`.

PROGRAMMING

1. Write statements to extract the individual tokens in a string of the form `"Doe, John 5/15/65"`. Use the `indexof` method to find the string `", "` and the symbol `/` and use the substring method to extract the substrings between these delimiters.
2. Write statements to extract the words in Self-Check Exercise 2 and then create a new `String` object with all the words separated by commas. Use `StringBuilder` to build the new string.
3. For Self-Check Exercise 4, write a loop to display all the tokens that are extracted.
4. Use `StringJoiner` to create a string with a prefix of `"("`, a suffix of `")"`, and a delimiter of `" + "`. Use `StringJoiner.add` to store the elements of `String` array `symbols` in the `StringJoiner`.
5 Redo Programming Exercise 4 using a `StringBuilder`.

A.6 Wrapper Classes for Primitive Types

We have seen that the primitive numeric types are not objects, but sometimes we need to process primitive-type data as objects. For example, we may want to pass a numeric value to a method that requires an object as its argument. Java provides a set of classes called *wrapper classes* whose objects contain primitive-type values: `Float`, `Double`, `Integer`, `Boolean`, `Character`, and so on. These classes provide constructor methods to create new objects that "wrap" a specified value. They also provide methods to "unwrap," or extract, an object's value and methods to compare two objects. Table A.13 shows some methods for wrapper class `Integer` (part of `java.lang`). The other numeric wrapper classes also provide these methods, except that method `parseInt` is replaced by a method `parseClassType`, where `ClassType` is the data type wrapped by that class.

TABLE A.13
Methods for Class `Integer`

Method	Behavior
`int compareTo(Integer anInt)`	Compares two `Integer`s numerically
`double doubleValue()`	Returns the value of this `Integer` as a **`double`**
`boolean equals(Object obj)`	Returns **`true`** if the value of this `Integer` is equal to its argument's value; returns **`false`** otherwise
`int intValue()`	Returns the value of this `Integer` as an **`int`**
`static int parseInt(String s)`	Parses the string argument as a signed integer

`static Integer valueOf(char ch`
`static Integer valueOf(String s)`
`static Integer valueOf(int i)`
	Accepts a numeric string, character, or integer as its argument and returns an `Integer` object with the corresponding value. If the argument is a character, it returns an `Integer` object with its Unicode value.

`String toString()`	Returns a `String` object representing this `Integer`'s value

In earlier versions of Java, a programmer could not mix type `int` values and type `Integer` objects in an expression. If you wanted to increment the value stored in `Integer` object `nInt`, you would have to unwrap the value, increment it, and then wrap the value in a new `Integer` object:

```
int n = nInt.intValue();
Integer nInt = new Integer(n++);
```

Java 5.0 introduced a feature known as autoboxing/unboxing for primitive types. This enables programmers to use a primitive type in contexts where an `Object` is needed or to use a wrapper object in contexts where a primitive type is needed. Using autoboxing/unboxing, you can rewrite the statements above as

```
int n = nInt;
nInt = n++;
```

or even as the single statement:

```
nInt++;
```

EXAMPLE A.12 The first pair of the following statements creates two `Integer` objects. The next pair unboxes the **`int`** value contained in each object and performs the indicated operation . In the next pair, method `parseInt` parses its string argument to an **`int`** (not `Integer`) value. Method `valueOf` autoboxes the decimal value (78) of the Unicode for `'M'` and stores it in `Integer i5`. The last statement displays the value (35) wrapped in `Integer` object `i1`.

```
Integer i1 = 35;                          // Autoboxes 35.
Integer i2 = 1234;                        // Autoboxes 1234.
Integer i3 = i1 + i2;                     // Unboxes i1 and i2, autoboxes
                                          // their sum 1269, and assigns
                                          // it to i3.
int i2Val = i2++;                         // Unboxes i2, increments it to
                                          // 1235 and autoboxes it, and
                                          // assigns 1234 to i2Val.
int i3Val = Integer.parseInt("-357");     // Parses "-357" to -357 and
                                          // assigns it to i3Val.
Integer i4 = Integer.valueOf("753");      // Parses  "753" to 753, autoboxes
                                          // it and assigns it to i4.
Integer i5 = Integer.valueOf('N');        // Autoboxes 78 and assigns it to i5
System.out.println(i1);                   // Automatically calls
                                          // toString() and displays 35.
```

EXERCISES FOR SECTION A.6

SELF-CHECK

1. Do you think objects of a wrapper type are immutable or not? Explain your answer.
2. For objects `i1`, `i2` in Example A.12, what do the following two statements display?

```
System.out.println(i1 + i2);
System.out.println(i1.toString() + i2.toString());
```

PROGRAMMING

1. Write statements that double the value stored in the `Integer` object referenced by `i1`. Draw a diagram showing the objects referenced by `i1` before and after these statements execute.

2. There is no * (multiply) operator for type Integer objects. Suppose you have Integer objects i1, i2, i3. Write a statement to multiply the three type int values in these objects and store the product in an Integer object i4. Show how you would do this without using autoboxing/unboxing.

A.7 Defining Your Own Classes

We mentioned earlier that a Java program is a collection of classes; consequently, when you write a Java program, you will develop one or more classes. We will show you how to write a Java class next.

A class Person might describe a group of objects, each of which is a particular human being. For example, instances of class Person would be yourself, your mother, and your father. A Person object could store the following data:

- Given name
- Family name
- ID number
- Year of birth

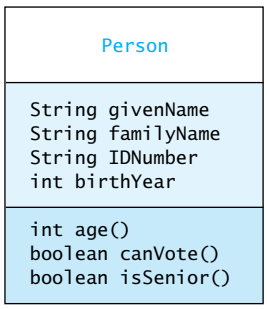
The following are a few of the operations that can be performed on a Person object:

- Calculate the person's age
- Test whether two Person objects refer to the same person
- Determine whether the person is old enough to vote
- Determine whether the person is a senior citizen
- Get one or more of the data fields for the Person object
- Set one or more of the data fields for the Person object

Figure A.6 shows a diagram of class Person. This figure uses the *Unified Modeling Language*™ (*UML*) to represent the class. UML diagrams are a standard means of documenting class relationships that is widely used in industry. The class is represented by a box. The top compartment of the box contains the class name. The data fields are shown in the middle compartment, and some of the methods are shown in the bottom compartment. Data fields are also called *instance variables* because each class instance (object) has its own storage for them. We discuss UML further in Appendix B.

Figure A.7 shows how two objects or instances of the class Person (author1 and author2) are represented in UML. A curved arrow from the reference variable for each object (author1,

FIGURE A.6 Class Diagram for Person



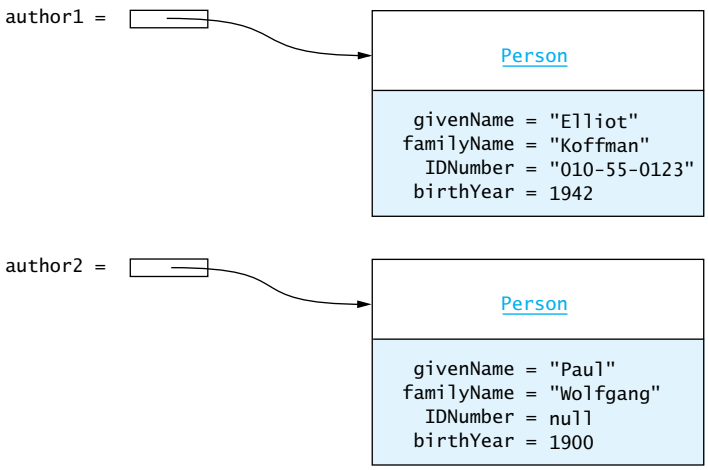


FIGURE A.7
Object Diagrams of Two Instances of Class `Person`

`author2`) points to the object, as we have shown in previous figures. Each object is represented by a box in which the top compartment contains the class name (`Person`), underlined, and the bottom compartment contains the data fields and their values. (For simplicity, we show the value of each `String` data field instead of a reference to a `String` object.)

Listing A.1 shows class `Person` and the instance methods for this class. The lines that are delimited by `/**` and `*/` are comments. They are program documentation, extremely important for human programmers, but ignored by the compiler. We discuss the form of the comments used in class `Person` at the end of this section.

We declare four data fields and two constants (all uppercase letters) before the methods (although many Java programmers prefer to declare methods before data fields). In the constant declarations, the modifier **`final`** indicates that the constant value may not be changed. The modifier **`static`** indicates that the constant is being defined for the class and does not have to be replicated in each instance. In other words, storage for the constant `VOTE_AGE` is allocated once, regardless of how many instances of `Person` are created.

LISTING A.1
Class `Person`

```
/** Person is a class that represents a human being. */
public class Person {
    // Data Fields
    /** The given name */
    private String givenName;
    /** The family name */
    private String familyName;
    /** The ID number */
    private String IDNumber;
    /** The birth year */
    private int birthYear = 1900;

    // Constants
    /** The age at which a person can vote */
    private static final int VOTE_AGE = 18;
    /** The age at which a person is considered a senior citizen */
    private static final int SENIOR_AGE = 65;

    // Constructors
    /** Construct a person with given values
```

```
    @param first The given name
    @param family The family name
    @param ID The ID number
    @param birth The birth year
 */
public Person(String first, String family, String ID, int birth) {
    givenName = first;
    familyName = family;
    IDNumber = ID;
    birthYear = birth;
}

/** Construct a person with only an IDNumber specified.
    @param ID The ID number
 */
public Person(String ID) {
    IDNumber = ID;
}

// Modifier Methods
/** Sets the givenName field.
    @param given The given name
 */
public void setGivenName(String given) {
    givenName = given;
}

/** Sets the familyName field.
    @param family The family name
 */
public void setFamilyName(String family) {
    familyName = family;
}

/** Sets the birthYear field.
    @param birthYear The year of birth
 */
public void setBirthYear(int birthYear) {
    this.birthYear = birthYear;
}

// Accessor Methods
/** Gets the person's given name.
    @return the given name as a String
 */
public String getGivenName() { return givenName; }

/** Gets the person's family name.
    @return the family name as a String
 */
public String getFamilyName() { return familyName; }

/** Gets the person's ID number.
    @return the ID number as a String
 */
public String getIDNumber() { return IDNumber; }

/** Gets the person's year of birth.
    @return the year of birth as an int value
 */
public int getBirthYear() { return birthYear; }
```

```
    // Other Methods
    /** Calculates a person's age at this year's birthday.
        @param year The current year
        @return the year minus the birth year
     */
    public int age(int year) {
        return year - birthYear;
    }

    /** Determines whether a person can vote.
        @param year The current year
        @return true if the person's age is greater than
                or equal to the voting age
     */
    public boolean canVote(int year) {
        int theAge = age(year);
        return theAge >= VOTE_AGE;
    }

    /** Determines whether a person is a senior citizen.
        @param year the current year
        @return true if person's age is greater than or
        equal to the age at which a person is
        considered to be a senior citizen
     */
    public boolean isSenior(int year) {
        return age(year) >= SENIOR_AGE;
    }

    /** Retrieves the information in a Person object.
        @return the object state as a string
     */
    public String toString() {
        return "Given name: " + givenName + "\n"
               + "Family name: " + familyName + "\n"
               + "ID number: " + IDNumber + "\n"
               + "Year of birth: " + birthYear + "\n";
    }

    /** Compares two Person objects for equality.
        @param per The second Person object
        @return true if the Person objects have same
        ID number; false if they don't
     */
    public boolean equals(Person per) {
        if (per == null)
            return false;
        else
            return IDNumber.equals(per.IDNumber);
    }
}
```

Private Data Fields, Public Methods

The modifier **`private`** sets the visibility of each variable or constant to *private visibility*. This means that these data fields can be accessed only within the class definition. Only class members with *public visibility* can be accessed outside of the class.

The reason for having private visibility for data fields is to control access to an object's data and to prevent improper use and processing of an object's data. If a data field is private, it can be processed outside of the class only by invoking one of the public methods that are part of the class. Therefore, the programmer who writes the public methods controls how the data

field is processed. Also, the details of how the private data are represented and stored can be changed at a later time by the programmer who implements the class, and the other programs that use the class (called the class's *clients*) will not need to be changed.

Constructors

In Listing A.1, the two methods that begin with `public Person` are constructors. One of these methods is invoked when a new class instance is created. The constructor with four parameters is called if the values of all data fields are known before the object is created. For example, the statement

```
Person author1 = new Person("Elliot", "Koffman", "010-055-0123", 1942);
```

creates the first object shown in Figure A.7, initializing its data fields to the values passed as arguments.

The second constructor is called when only the value of data field `IDNumber` is known at the time the object is created.

```
Person author2 = new Person("030-555-5555");
```

In this case, data field `IDNumber` is set to `"030-555-5555"`, but all the other data fields are initialized to the default values for their data type (see Table A.14) unless a different initial value is specified (1900 for `birthYear`). The `String` data fields are initialized to `null`, which means that no `String` object is referenced. You can use the modifier methods at a later time to set the values of the other data fields. The statement

```
author2.setGivenName("Paul");
```

sets the data field `givenName` to reference the `String` object `"Paul"`. Note that there is no `setIDNumber` method, so this data field value can't be assigned or changed at a later time.

The No-Parameter Constructor

A constructor with no parameters is called the *no-parameter constructor* (or *no-argument constructor*). This constructor is sometimes called the default constructor because Java automatically defines this constructor with an empty body for a class that has no constructor definitions. However, if you define one or more constructors for a class, you must also explicitly define the no-parameter constructor, or it will be undefined for that class. Because two constructors are defined for class `Person`, but the no-parameter constructor is not, the statement

```
Person p = new Person(); // Invalid call to no-parameter constructor.
```

will not compile.

TABLE A.14
Default Values for Data Fields

Data Field Type	Default Value
`int` (or other integer type)	`0`
`double` (or other real type)	`0.0`
`boolean`	**`false`**
`char`	`\u0000` (the smallest Unicode character: the null character)
Any reference type	**`null`**

Modifier and Accessor Methods

Because the data fields have private visibility, we need to provide public methods to access them. Normally, we want to be able to get or retrieve the value of a data field, so each data field in class `Person` has an accessor method (also called getter) that begins with the word `get` and ends with the name of the data field (e.g., `getFamilyName`). If we want to allow a class user to update or modify the value of a data field, we provide a modifier method (also called mutator or setter) beginning with the word `set` and ending with the name of the data field (e.g., `setGivenName`). Currently, there is an accessor for each data field in this example and a modifier for all but the `IDNumber` data field. The reason for this is to deny a client the ability to change a person's ID number.

The modifier methods are type `void` because they are executed for their effect (to update a data field), not to return a value. In the method `setBirthYear`,

```
public void setBirthYear(int birthYear) {
    this.birthYear = birthYear;
}
```

the assignment statement stores the integer value passed as an argument in data field `birthYear`. (We explain the reason for **`this`**. in the next subsection.)

The accessor method for data field `givenName`,

```
public String getGivenName() { return givenName; }
```

is type `String` because it returns the `String` object referenced by `givenName`. If the class designer does not want other users (clients) of the class to be able to access or change the data field values, these methods can be given private visibility.

Use of `this.` in a Method

Method `setBirthYear` uses the statement

```
this.birthYear = birthYear;
```

to store a value in data field `birthYear`. We can use **`this`**.*aDataField* in a method to access a data field of the current object. Because we used `birthYear` as a parameter in method `setBirthYear`, the Java compiler will translate `birthYear` without the prefix **`this`**. as referring to the parameter `birthYear`, not to the data field. The reason is the declaration of `birthYear` as a parameter is local to the method and, therefore, hides the data field declaration.

The Method `toString`

The last two methods, `toString` and `equals`, are found in most Java classes. The method `toString` creates a `String` object that represents the information stored in an object (the *state* of an object). The escape sequence `\n` is the newline character, and it terminates an output line when the string is displayed. A client of class `Person` could use the statement

```
System.out.println(author1.toString());
```

to display the state of `author1`. In fact, the statement

```
System.out.println(author1);
```

would also display the state of `author1` because `System.out.println` and `System.out.print` automatically apply method `toString` to an object that appears in their argument list. The following lines would be displayed by this statement.

```
Given name: Elliot
Family name: Koffman
ID number: 010-055-0123
Year of birth: 1942
```

PROGRAM STYLE

Using `toString` Instead of Displaying Data Fields

Java programmers use method `toString` to build a string that represents the object state. This string can then be displayed at the console, written to a file, displayed in a dialog window, or displayed in a Graphical User Interface (GUI). This is more flexible than the approach taken in many programming languages, in which each data field is displayed or written to a file.

The Method `equals`

The method `equals` compares the object to which it is applied (*this* object) to the object that is passed as an argument. It returns **`true`** if the objects are determined to be the same based on the data they store. It returns **`false`** if the argument is **`null`** or if the objects are not the same. We will assume that two `Persons` are the same if they have the same ID number.

```
public boolean equals(Person per) {
    if (per == null)
        return false;   // exit - no object defined

    return IDNumber.equals(per.IDNumber);
}
```

The second `return` statement returns the result of the method call

```
IDNumber.equals(per.IDNumber)
```

Note that we can look at parameter `per`'s private `IDNumber` because `per` references an object of this class (type `Person`). Because `IDNumber` is type `String`, the `equals` method of class `String` is invoked with the `IDNumber` of the second object as an argument. If the two `IDNumber` data fields have the same contents, the `String equals` method will return **`true`**; otherwise, it will return **`false`**. The `Person equals` method returns the result of the `String equals` method. In Section 3.5, we discuss the `equals` method in more detail and show you a better way to write this method.

PROGRAM STYLE

Returning a `boolean` Value

Some programmers unnecessarily write **`if`** statements to return a **`boolean`** value. For example, instead of writing

```
return IDNumber.equals(per.IDNumber);
```

they write

```
if (IDNumber.equals(per.IDNumber))
    return true;
else
    return false;
```

Resist this temptation. The **`return`** statement by itself returns the value of the **`if`** statement condition, which must be **`true`** or **`false`**. It does this in a clear and succinct manner using one line instead of four.

Declaring Local Variables in Class Person

There are three other methods declared in class `Person`. Methods `age`, `canVote`, and `isSenior` are all passed the current year as an argument. Method `canVote` calls method `age` to determine the person's age. The result is stored in local variable `theAge`. The result of calling method `canVote` is the value of the **`boolean`** expression following the keyword **`return`**.

```
public boolean canVote(int year) {
    int theAge = age(year); // Local variable
    return theAge >= VOTE_AGE;
}
```

It really was not necessary to introduce local variable `theAge`; the call to method `age` could have been placed directly in the **`return`** statement (as it is in method `isSenior`). We wanted, however, to show you how to declare local variables in a Java method. The variable `theAge` is called a *local variable* because it, like the parameter `year`, can only be referenced in the body of method `canVote`.

PITFALL

Referencing a Data Field or Parameter Hidden by a Local Declaration

If you happen to declare a local variable (or parameter) with the same name as a data field, the Java compiler will translate the use of that name in a method as meaning the local variable (or parameter), not the data field. So if `theAge` was also declared as a data field in class `Person`, the statement

```
theAge++;
```

would increment the local variable, but the data field value would not change. To access the data field instead of the local variable, use the prefix **`this`**., just as we did earlier when a parameter had the same name as a data field.

PITFALL

Using Visibility Modifiers with Local Variables

Using a visibility modifier with a local variable would cause a syntax error because a local variable is visible only within the method that declares it. Therefore, it makes no sense to give it public or private visibility.

An Application that Uses Class Person

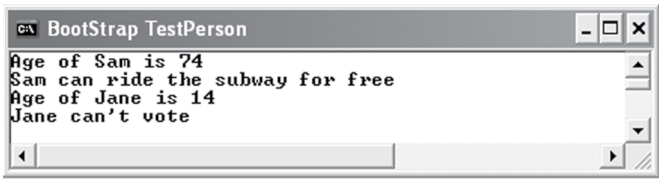
To test class `Person`, we need to write a Java application program that contains a `main` method. The `main` method should create one or more instances of class `Person` and display the results of applying the class methods. Listing A.2 shows a class `TestPerson` that does this. To execute the `main` method, you must compile and run class `TestPerson`. As long as `Person` and `TestPerson` are in the same folder (disk directory), the application program will run. Figure A.8 shows a sample run.

LISTING A.2
Class `TestPerson`

```
/** TestPerson is an application that tests class Person. */
public class TestPerson {
    public static void main(String[] args) {
        var p1 = new Person("Sam", "Jones", "1234", 1947);
        var p2 = new Person("Jane", "Jones", "5678", 2007);
        System.out.println("Age of " + p1.getGivenName() +
                           " is " + p1.age(2021));
        if (p1.isSenior(2021))
            System.out.println(p1.getGivenName() +
                               " can ride the subway for free");
        else
            System.out.println(p1.getGivenName() + "
                               must pay to ride the subway");

        System.out.println("Age of " + p2.getGivenName() +
                           " is " + p2.age(2021));
        if (p2.canVote(2021))
            System.out.println(p2.getGivenName() + " can vote");
        else
            System.out.println(p2.getGivenName() + " can't vote");
    }
}
```

FIGURE A.8
Sample Run of Class `TestPerson`



Although we will generally write separate application classes such as `TestPerson`, you could also insert the `main` method directly in class `Person` and then compile and run class `Person`. Program execution will start at the `main` method, and the result will be the same. If you use separate classes, make sure that you put them in the same folder (directory).

Objects as Arguments

We stated earlier that Java arguments are passed by value. For primitive-type arguments, this protects the value of a method's argument and ensures that its value can't be changed by the method. However, this is not the case for arguments that are objects. If an argument is an object, its address is passed to the method, so the method parameter will reference the same object as the method argument. If the method happens to change a data field of its object parameter, that change will be made to the object argument. We illustrate this next.

EXAMPLE A.13 Suppose method `changeGivenName` is defined as follows:

```
public void changeGivenName(Person per) {
    per.givenName = this.givenName;
}
```

Also suppose a client program declares `firstMan` and `firstWoman` as reference variables of type `Person`. After the method call

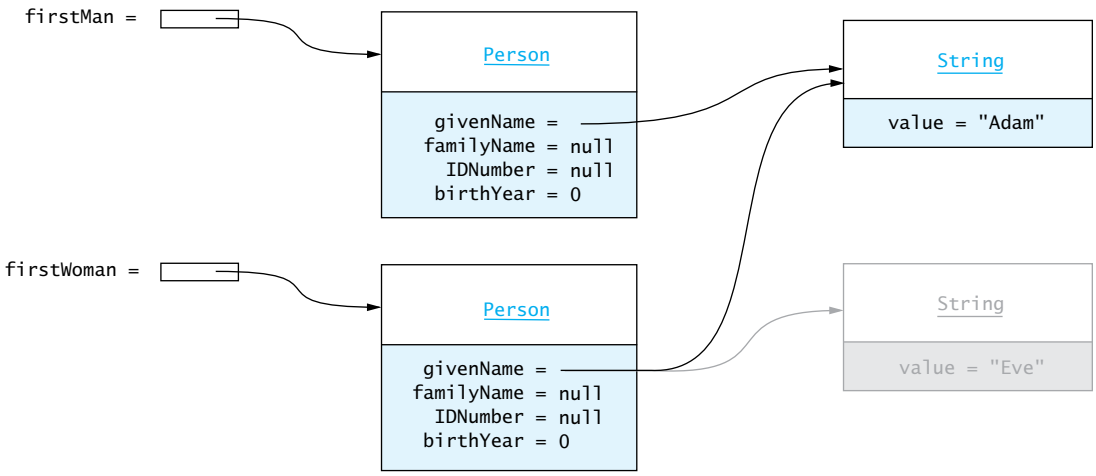
```
firstMan.changeGivenName(firstWoman)
```

parameter `per` (declared in method `changeGivenName`) and reference variable `firstWoman` (declared in the client) will reference the same object. The statement

```
per.givenName = this.givenName;
```

will set the `givenName` data field of the object referenced by `per` (and `firstWoman`) to reference the same string as the `givenName` field of `this` object (the object referenced by `firstMan`). Figure A.9 shows the `givenName` data field of the objects referenced by `firstMan` and `firstWoman` after the foregoing statement executes.

FIGURE A.9
Reference Variables `firstMan` and `firstWoman`



Classes as Components of Other Classes

Class `Person` has three data fields that are type `String`, so `String` objects are *components* of a `Person` object. In Figure A.10, this component relationship is indicated by the solid diamond symbol at the end of the line drawn from the box representing class `String` to the box representing class `Person`. Like the class diagram in Figure A.6 and the object diagrams in Figure A.7, Figure A.10 is a UML diagram, showing the relationships between classes. We will follow the UML conventions for documenting class relationships described in Appendix B.

FIGURE A.10
UML Diagram Showing That String Objects Are Components of Class Person

Java Documentation Style for Classes and Methods

Java provides a standard form for writing comments and documenting classes, which we will use in this book. If you use this form, you can run a program called *Javadoc* (part of the JDK) to generate a set of HTML pages describing each class and its data fields and methods. These pages will look just like the ones that document the Java API classes on the Oracle Corporation Java website (`https://docs.oracle.com/java/javase/11/`).

The Javadoc program focuses on text that is enclosed within the delimiters `/**` and `*/`. The introductory comment that describes the class is displayed on the HTML page exactly as it is written, so you should write that carefully. The lines that begin with the symbol `@` are Javadoc tags. They are described in Table A.15. In this book, we will use one `@param` tag for each method parameter. We will not use a `@return` tag for **`void`** methods. The first sentence of the comment for each method appears in the method summary part of the HTML page. The information provided in the tags will appear in the method detail part. Figures A.11 through A.13 show part of the documentation generated by running Javadoc for a class `Person` similar to the `Person` class in this chapter.

To run the Javadoc program, change to the directory that contains the source files that you would like to process. Then, to create the HTML documentation files, enter the command

`javadoc` *className1*`.java` *className2*`.java`

where the Java source file names (*className1*`.java`, *className2*`.java`, etc.) follow the `javadoc` command.

TABLE A.15
Javadoc Tags

Javadoc Tag and Example of Use	Purpose
`@author` *`Koffman and Wolfgang`*	Identifies the class author
`@param first` *`The given name`*	Identifies a method parameter
`@return` *`The person's age`*	Identifies a method return value

FIGURE A.11
Field Summary for Class `Person`

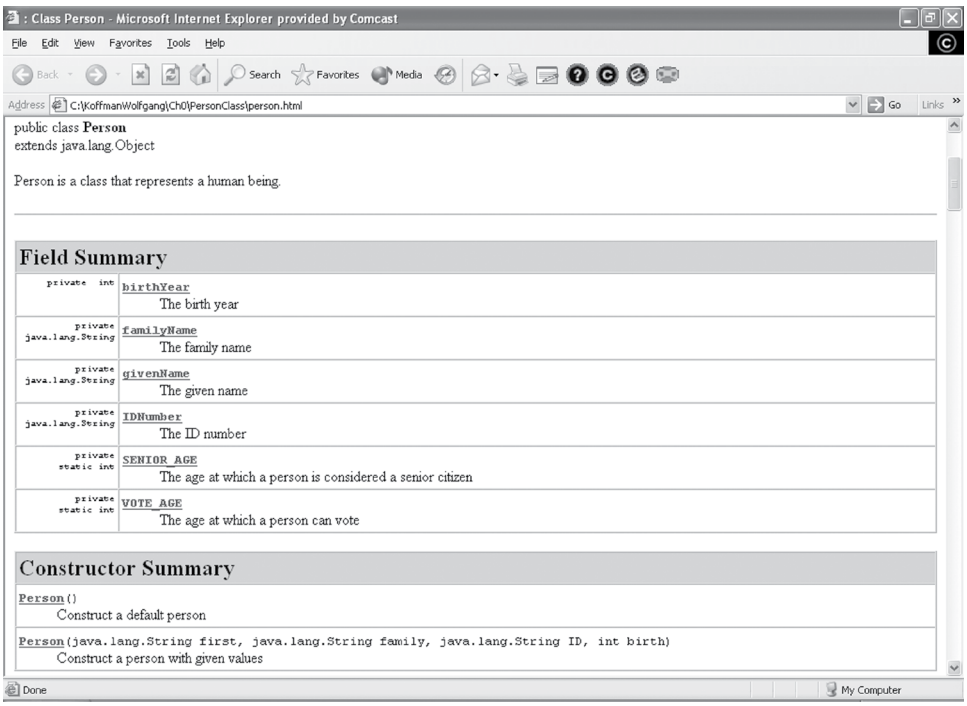


FIGURE A.12
Method Summary for Class `Person`

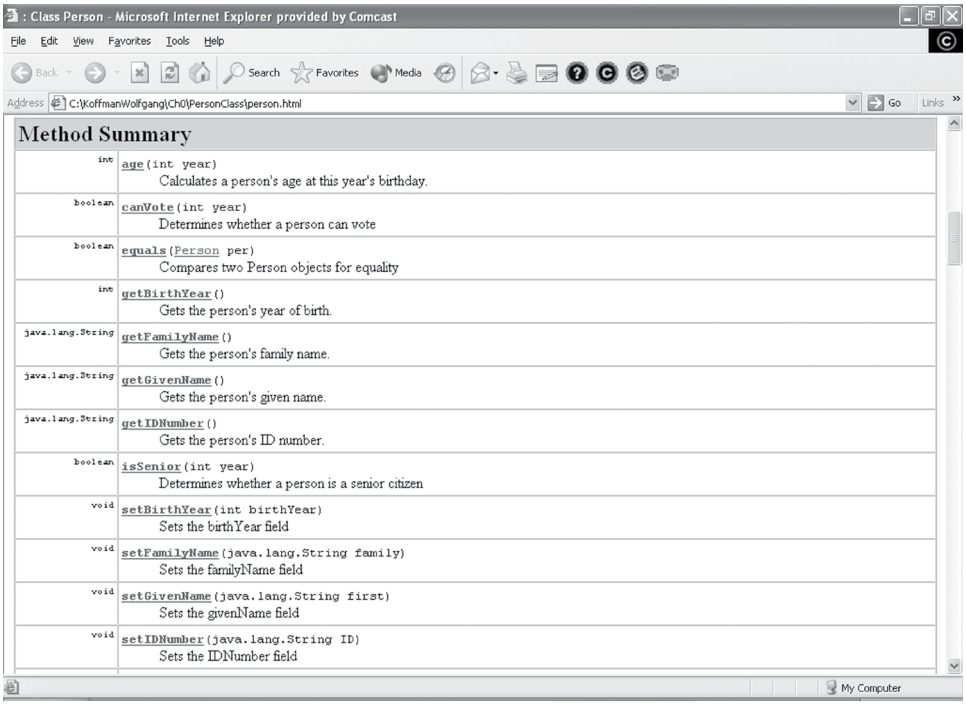
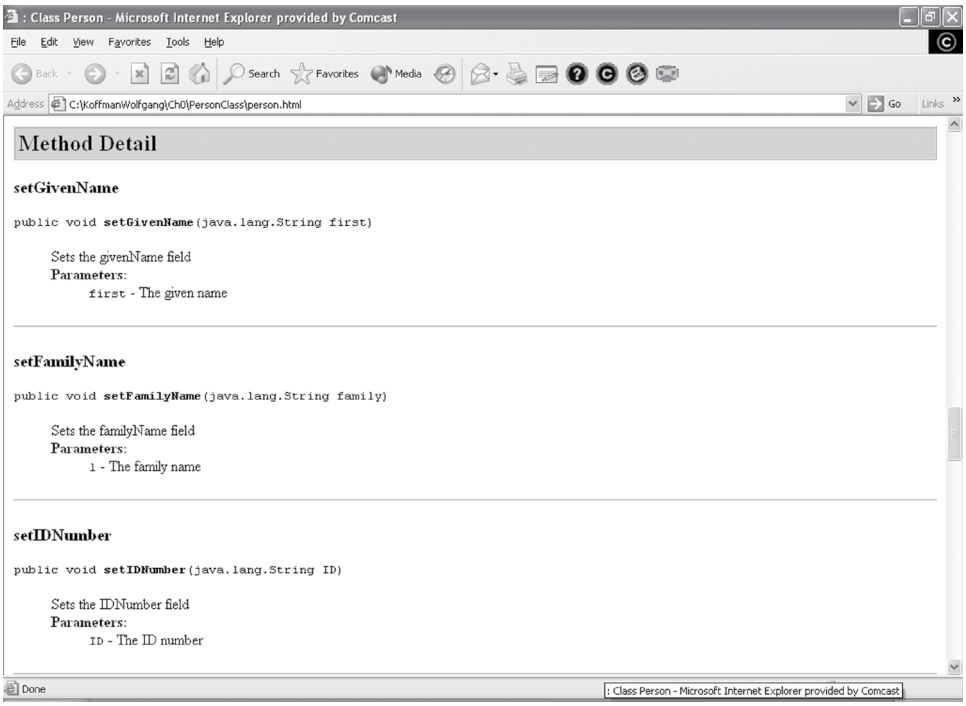


FIGURE A.13
Method Detail for Class `Person`



If you want to show the private data fields and methods, add the command line argument `-private`. If you want to create documentation files for all the `.java` files in the directory, use the wildcard `*` for the class name.

```
javadoc -private *.java
```

Another useful command line argument is `-d` *destinationFolder*, which allows you to specify a folder or directory other than the source folder for the Javadoc HTML files.

If you are using an IDE there may be special tools provided for generating a Javadoc file. In IntelliJ, you open the class you wish to document in the Edit window. Then select Generate JavaDoc from the Tools menu and you will be prompted to browse for the directory where you want the file saved. The Javadoc Web page will be created and saved as `index.html` in that directory and will also appear in a browser window.

EXERCISES FOR SECTION A.7

SELF-CHECK

1. Explain why methods have public visibility but data fields have private visibility.
2. Download file `Person.java` from the textbook website and run `javadoc` on it.
3. Trace the execution of the following statements.

```
Person p1 = new Person("Adam", "Jones", "wxyz", 0);
p1.setBirthYear(1990);
Person p2 = new Person();
p2.setGivenName("Eve");
p2.setFamilyName(p1.getFamilyName());
p2.setBirthYear(p1.getBirthYear() + 10);
if (p1.equals(p2))
    System.out.println(p1 + "\nis same person as\n\n" + p2);
else
    System.out.println(p1 + "\nis not the same person as\n\n" + p2);
```

PROGRAMMING

1. Write a method `getInitials` that returns a string representing a `Person` object's initials. There should be a period after each initial. Write Javadoc tags for the method.
2. Add a data field `motherMaidenName` to `Person`. Write an accessor and a modifier method for this data field. Modify class `toString` and class `equals` to include this data field. Assume two `Person` objects are equal if they have the same ID number and mother's maiden name. Write Javadoc tags for the method.
3. Write a method `compareTo` that compares two `Person` objects and returns an appropriate result based on a comparison of the ID numbers. That is, if the ID number of the object that `compareTo` is applied to is less than (is greater than) the ID number of the argument object, the result should be negative (positive). The result should be 0 if they have the same ID numbers. Write Javadoc tags for the method.
4. Write a method `switchNames` that exchanges a `Person` object's given and family names. Write Javadoc tags for the method.

A.8 Arrays

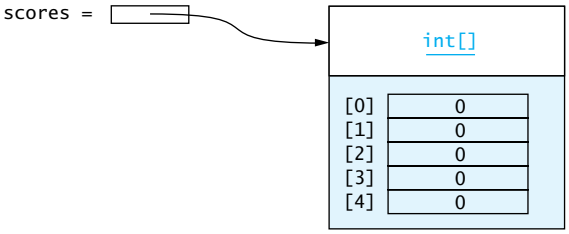
In Java, an array is also an object. The elements of an array are indexed and are referenced using a subscripted variable of the form:

arrayName[*subscript*]

Next, we show some different ways to declare arrays and allocate storage for arrays.

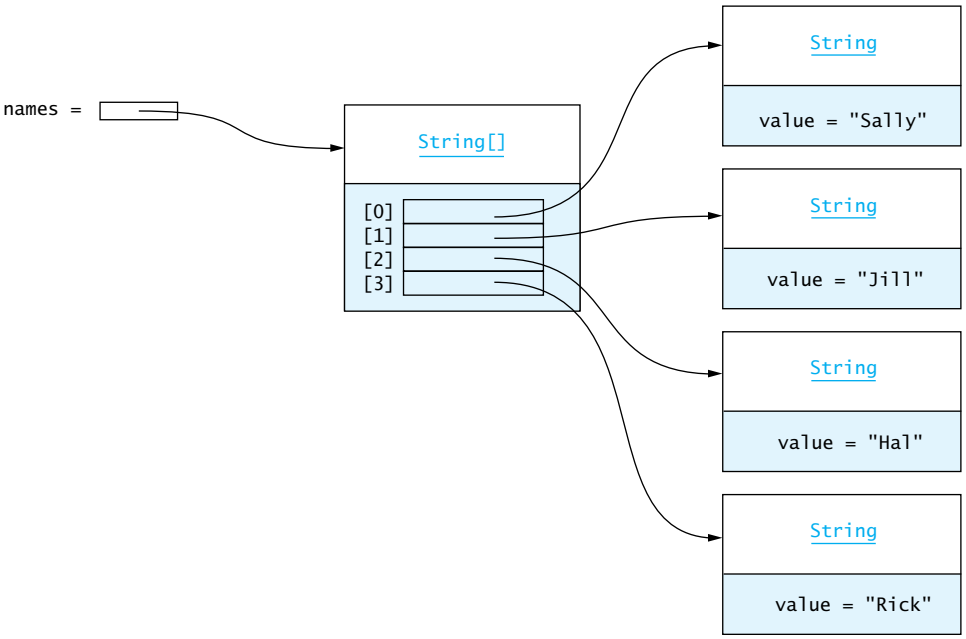
EXAMPLE A.14 The following statement declares a variable `scores` that references a new array object that can store five type **`int`** values (subscripts 0 through 4) as shown. Each element is initialized to 0.

```
int[] scores = new int[5]; // An array with 5 type int values
```



EXAMPLE A.15 The following statement declares a variable `names` that references a new array object that can store four type `String` objects. The values stored are specified in the *initializer list*.

```
String[] names = {"Sally", "Jill", "Hal", "Rick"};
```



PITFALL

Out-of-Bounds Subscripts

Some programming languages allow you to use an array subscript that is outside of the array bounds. For example, if you attempt to reference `scores[5]`, a C or C++ compiler would access the first memory cell following the array scores. This is considered an error, but it is not detected by the run-time system and will probably lead to another error that will be detected farther down the road (before it does too much damage, you hope). Java, however, verifies that the current value of each array subscript is within the array bounds. If it isn't, you will get an `ArrayIndexOutOfBoundsException` error.

EXAMPLE A.16 The first of the following statements declares a variable `people` that can reference an array object for storing type `Person` objects. Storage has not yet been allocated for the array object (or for the `Person` objects). The second statement assumes that `n` is defined, possibly through an input operation. The last statement allocates storage for an array object with `n` elements. Each array element can reference a type `Person` object, but initially each element has the value **`null`** (no object referenced).

```
// Declare people as type Person[].
Person[] people;
// Define n in some way.
int n = ...
// Allocate storage for the array.
people = new Person[n];
```

We can create some `Person` objects and store them in the array. The following statements store two `Person` objects in array people.

```
people[0] = new Person("Elliot", "Koffman", "010-055-0123", 1942);
people[1] = new Person("Paul", "Wolfgang", "015-023-4567", 1945);
```

PITFALL

Forgetting to Declare Storage for an Array

As just shown, you can separate the declaration of variable `people` (the array reference variable) from the step that actually allocates storage (`people = new . . .`). However, you can't reference the array elements before you allocate storage for the array. Similarly, if the array elements reference objects, you must separately allocate storage for each object.

Data Field `length`

A Java array has a `length` data field that can be used to determine the array's size. The value of `names.length` is 4; the value of `people.length` is the same as the value of `n` when storage was allocated for the array. The subscripted variable `people[people.length - 1]` references the last element in array `people` whereas the subscript of the first element is 0. The following **`for`** statement can be used to display all the `Person` objects stored in array `people`, regardless of the array size.

```
for (int i = 0; i < people.length; i++ )
    if (people[i] != null)
        System.out.println(people[i] + "\n");
```

PITFALL

Using `length` Incorrectly

The value of data field `length` is set when storage is allocated for the array, and it is final. Therefore, it can't be changed by the programmer. A statement such as

```
people.length++; // invalid attempt to increment length
```

would cause a syntax error.

Another common error is using parentheses with `length`. The expression `people.length()` causes a syntax error because `length` is a data field, not a method, of an array.

Method Arrays.copyOf

Although you can't change the length of a particular array object, you can copy the values stored in one array object to another array object using method `Arrays.copyOf`. This method returns a copy of a given array and either truncates or pads the copy to a new length. The method is overloaded for arrays of each primitive type, and there is a generic form for copying arrays of class types.

EXAMPLE A.17 The following statements create a new array `tempScores` that is twice the size of array scores and copy the elements in array `scores` into the first half of array `tempScores`. The remaining entries of `tempScores` are set to zero. Finally, we reset variable `scores` to reference the same array as `tempScores` (see Figure A.14). The storage originally allocated to store the elements of array `scores` can now be reclaimed by the garbage collector.

```
int[] tempScores = Arrays.copyOf(scores, 2 * scores.length);
scores = tempScores;
```

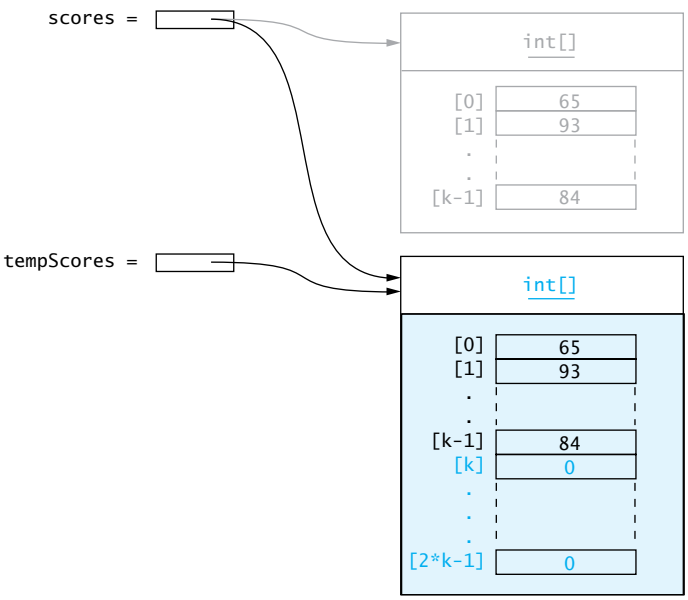


FIGURE A.14 Doubling the Size of the Array Referenced by Scores

Method System.arrayCopy

The method `Arrays.copyOf` makes a copy of the whole array. A general method that will copy a selected portion of an array into another array is `System.arrayCopy`, which has the form

`System.arraycopy(`*source, sourcePos, destination, destPos, numElements*`);`

The parameters *sourcePos* and *destPos* specify the starting positions in the *source* and *destination* arrays, respectively. The parameter *numElements* specifies the number of elements to copy. If this number is too large, an `ArrayIndexOutOfBoundsException` error occurs.

Method `System.arraycopy` is effectively the following:

```
for (int k = 0; k < numElements; k++) {
    destination[destPos + k] = source[sourcePos + k];
}
```

It is implemented within the JVM using native machine code instructions that efficiently copy a block of data from one location to another.

Array Data Fields

It is very common in Java to encapsulate an array, together with the methods that process it, within a class. Rather than allocate storage for a fixed-size array, we would like the client to be able to specify the array size when an object is created. Therefore, we should define a constructor with the array size as a parameter and have the constructor allocate storage for the array. Class `Company` in Listing A.3 has a data field `employees` that references an array of `Person` objects. Both constructors allocate storage for a new array when a `Company` object is created. The client of this class can specify the size of the array by passing a type `int` value to the constructor parameter `size`. If no argument is passed, the no-parameter constructor sets the array size to `DEFAULT_SIZE`.

LISTING A.3
Class Company

```
/** Company is a class that represents a company.
    The data field employees provides storage for
    an array of Person objects.
 */
public class Company {
    // Data Fields
    /** The array of employees */
    private Person[] employees;

    /** The default size of the array */
    private static final int DEFAULT_SIZE = 100;

    // Methods
    /** Creates a new array of Person objects.
        @param size The size of array employees
     */
    public Company(int size) {
        employees = new Person[size];
    }

    public Company() {
        employees = new Person[DEFAULT_SIZE];
    }

    /** Sets field employees.
        @param emp The array of employees
     */
    public void setEmployees(Person- emp) {
        employees = emp;
    }

    /** Gets field employees.
        @return employees array
     */
    public Person getEmployees() {
        return employees;
    }

    /** Sets an element of employees.
        @param index The position of the employee
        @param emp The employee
     */
    public void setEmployee(int index, Person emp) {
        if (index >= 0 && index < employees.length)
            employees[index] = emp;
    }
```

```
    /** Gets an employee.
        @param index The position of the employee
        @return The employee object or null if not defined
     */
    public Person getEmployee(int index) {
        if (index >= 0 && index < employees.length)
            return employees[index];
        else
            return null;
    }

    /** Builds a string consisting of all employee
        data, with newline characters between employees.
        @return The object's state
     */
    public String toString() {
        var result = new StringBuilder();
        for (int i = 0; i < employees.length; i++) {
            result.append(employees[i] + "\n");
        }
        return result.toString();
    }
}
```

There are modifier and accessor methods that process individual elements of array `Company` (`setEmployee` and `getEmployee`). Method `getEmployee` returns the type `Person` object at position `index`, or **`null`** if the value of `index` is out of bounds.

The `toString` method returns a string representing the contents of array `employees`. In the **`for`** loop, the argument in each call to method `append` is the string returned by applying method `Person.toString` to the current employee followed by a newline character. This string is appended to the string representing the data for all employees with smaller subscripts.

The following `main` method illustrates the use of class `Company` and displays the state of object `comp`.

```
    public static void main(String[] args) {
        Company comp = new Company(2);
        comp.setEmployee(0, new Person("Elliot", "K", "123", 1942));
        comp.setEmployee(1, new Person("Paul", "W", "234", 1945));
        System.out.println(comp);
    }
```

Array Results and Arguments

Method `setEmployees` in class `Company` takes a single argument `emp` that is type `Person[]`. The assignment statement

```
employees = emp;
```

resets array `employees` to reference the array argument. Storage allocated to the array previously referenced by `employees` can then be reclaimed by the garbage collector.

The return value of method `getEmployees` is type `Person[]`. The statement

```
return employees;
```

returns a reference to the array `employees`.

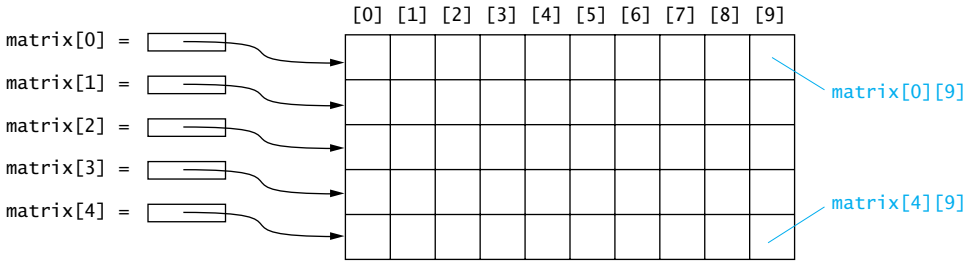
Arrays of Arrays

A Java array can have other arrays as its elements. If all these arrays are of the same size, then the array of arrays is a two-dimensional array.

EXAMPLE A.18 The declaration

```
double[][] matrix = new double[5][10];
```

allocates storage for a two-dimensional array, `matrix`, that stores 50 real numbers in 5 rows and 10 columns. The variable `matrix[i][j]` references the number with row subscript `i` and column subscript `j`. You can also declare arrays with more than two dimensions.



In Java you can have two-dimensional arrays with rows of different sizes. We illustrate this in the next two examples.

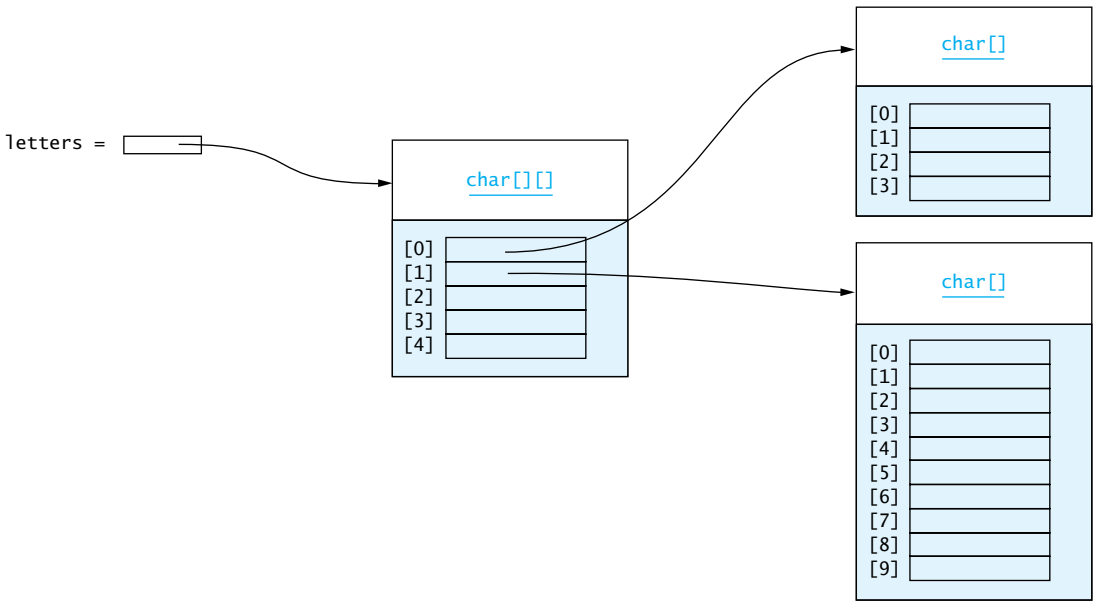
EXAMPLE A.19 The declaration

```
char[][] letters = new char[5][];
```

allocates storage for a two-dimensional array of characters with five rows, but the number of columns in each row is not specified. The statements

```
letters[0] = new char[4];
letters[1] = new char[10];
```

define the size of the first two rows and allocate storage for them. The subscripted variable `letters[0]` references the first row; `letters.length` is 5, the number of rows in the array; `letters[1].length` is 10, the number of elements in the row with subscript 1.



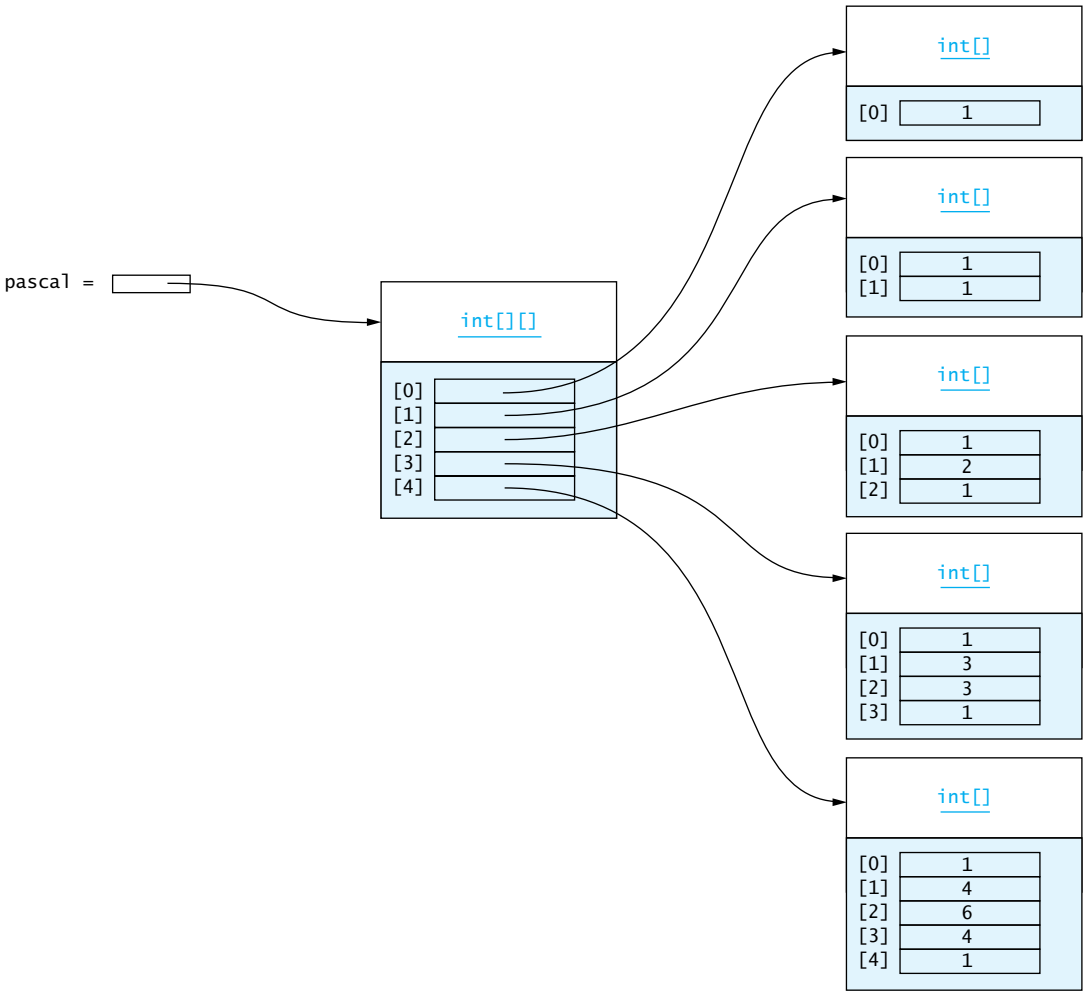
EXAMPLE A.20 The declaration

```
int[][] pascal = {
                                    {1},      // row 0
                                  {1, 1},     // row 1
                                {1, 2, 1},
                              {1, 3, 3, 1},
                            {1, 4, 6, 4, 1},
                        };
```

allocates storage for an array of arrays with five rows. The initializer list provides the values for each row, starting with row 0. The subscripted variable `pascal[0]` references the one-element array `{1}`, and `pascal[4]` references the array `{1, 4, 6, 4, 1}`. Each row has one more element than the previous one. The values shown above form a well-known mathematical entity called Pascal's triangle. Each element in a row, except for the first and last elements, is the sum of the two elements on either side of it in the previous row. For example, the number 6 in the last row is the sum of the numbers 3, 3 in the previous row. In mathematical notation,

`pascal[i + 1][j] = pascal[i][j - 1] + pascal[i][j].`

The first and last elements in each row are 1.



The following nested **for** statements sum all values in the Pascal triangle. In the outer **for** loop header, the expression `pascal.length` is the number of rows in the triangle. In the inner `for` loop header, the expression `pascal[row].length` is the number of columns in the array with subscript `row`.

```
int sum = 0;
for (int row = 0; row < pascal.length; row++)
    for (int col = 0; col < pascal[row].length; col++)
        sum += pascal[row][col];
```

EXERCISES FOR SECTION A.8

SELF-CHECK

1. Show the output that would be displayed by method `main` following Listing A.3.
2. Show that the formula for the interior elements of a Pascal triangle row is correct by evaluating it for each interior element of the last row.
3. What is the output of the following sample code fragment?

```
int[] x;
int[] y;
int[] z;
x = new int[20];
x[10] = 0;
y = x;
x[10] = 5;
System.out.println(x[10] + ", " + y[10]);
x[10] = 15;
z = new int[x.length];
System.arraycopy(x, 0, z, 0, 20);
x[10] = 25;
System.out.println(x[10] + ", " + y[10]+ ", " + z[10]);
```

4. What happens if you make a copy of an array of object references using method `System.arraycopy`? If the objects referenced by the new array are changed, how will this affect the original array?
5. Assume there is no initializer list for the Pascal triangle and you are trying to build up its rows. If row `i` has been defined, write statements to create row `i + 1`.

PROGRAMMING

1. Write code for a method

```
public static boolean sameElements(int[] a, int[] b)
```

that checks whether two arrays have the same elements in some order, with the same multiplicities. For example, two arrays

```
121    144    19    161    19    144    19    11
```

and

```
11    121    144    19    161    19    144    19
```

would be considered to have the same elements because 19 appears three times in each array, 144 appears twice in each array, and all other elements appear once in each array.

2. Write an `equals` method for class `Company`. The result should be **`true`** if the employees of one company match element for element with the employees of a different company. Assume that the objects referenced by each array `employees` are in order by ID number.

3. For the two-dimensional array `letters` in Example A.19, assume `letters[i]` is going to be used to store an array that contains the individual characters in `String` object `next`. Allocate storage for `letters[i]` based on the length of `next` and write a loop that stores each character of `next` in the corresponding element of `letters[i]`. For example, the first character in `next` should be stored in `letters[i][0]`.

A.9 Enumeration Types

In Java, we use classes and objects to represent things in the application domain. However, there are cases where the things we wish to represent are discrete objects with no attributes. For example, the colors of a traffic light: RED, YELLOW, and GREEN. One approach is to assign arbitrary integer values.

```
final static int RED = 0;
final static int YELLOW = 1;
final static int GREEN = 2;
```

This approach has limitations. For example, assume we also wanted to represent the colors of crayons:

```
final static int RED = 0;
final static int YELLOW = 1;
final static int BLUE = 2;
final static int GREEN = 3;
final static int ORANGE = 4;
final static int BROWN = 5;
final static int VIOLET = 6;
final static int BLACK = 7;
```

If these were both in the same program, there would be a conflict between the `GREEN` traffic light and the `GREEN` crayon.

Another limitation is input/output. We want our user to be able to enter `"GREEN"` instead of knowing that 2 represents green, and we want to output the string `"GREEN"` instead of the number 2.

The Java enumeration type allows us to declare a set of identifiers that can then be used to represent the concept in our application domain.

SYNTAX Enumeration Type Declaration

FORM:

`enum` *enumName* {*enumIdentifiers*}

EXAMPLE:

```
enum Traffic {RED, YELLOW, GREEN}
```

INTERPRETATION:

An enumeration type *enumName* is defined with the listed identifiers as members. An enumeration type is a class and the *enumIdentifiers* are objects of the class. These objects are constants. You cannot create additional instances of the enumeration type. The type declaration can be inserted in your class or in a public class `Traffic`.

Using Enumeration Types

You can create variables of enumeration types, compare enumeration values for equality, and use them as case labels in `switch` statements. For example:

```
switch (lightColor) {
    case Traffic.GREEN:
        // keep going
        ...
        break;
    case Traffic.YELLOW:
        if (condition)  // true if can stop at stop line
            // apply brakes
            ...
        else
            // keep going
            ...
        break;
    case Traffic.RED:
        // apply brakes
        ...
        break;
}
```

Each enumeration type is a subtype of `Enum<E>` where the `<E>` is a generic parameter representing the enumeration type. (We explain generic parameters in more detail in Section 2.2.) Thus, our `Traffic` example is a subclass of `Enum<Traffic>`. Table A.16 shows the methods defined in `Enum<E>`.

The statement below calls `Arrays.toString` to return the string `"[RED, YELLOW, GREEN]"` representing the contents of array `Traffic.values`. This string is stored in `trafficVals`

```
var trafficVals = Arrays.toString(Traffic.values());
```

The expression

```
Traffic.valueOf("RED")
```

will result in `Traffic.RED`. The statement below reads a color string from `Scanner scan` (See Section A.10) and stores its corresponding `Traffic` constant in `lightColor`.

```
var lightColor = Traffic.valueOf(scan.next());
```

TABLE A.16
Methods Defined in `Enum<E>`

Method	Behavior
`public static E[] values()`	Returns an array view of the enumeration. The indices are assigned in the order in which the enumerations were declared
`public static E valueOf(String name)`	Returns the enumeration constant with the specified `name`. The string must match exactly the identifier used to declare this enumeration value. If not, an `IllegalArgumentException` is thrown as described in Section A.12.

Assigning Values to Enumeration Types

An enumeration type declaration is a class declaration. Consequently, you can add additional members to the enumeration class. For example, you can assign arbitrary values by adding a field and providing a constructor. By default, the constructor is private and it is invoked when the enumeration constants are declared. For example, you could declare an enumeration `Coin` in which each enumeration identifier has a field `value` in parentheses which is the number of pennies it represents:

```
enum Coin {
    PENNY(1), NICKEL(2), DIME(10), QUARTER(25), HALF_DOLLAR(50);
    Coin(int value) {
        this.value = value;
    }
    private final int value;
    public int getValue() {return value;}
}
```

EXERCISES FOR SECTION A.9

SELF-CHECK

1. Define an enumeration `Suit` to represent the suits in a deck of cards: `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES`.
2. Define an enumeration `Rank` to represent the cards in a suit: `DEUCE`, `TREY`, `FOUR`, `FIVE`, . . . `NINE`, `TEN`, `JACK`, `QUEEN`, `KING`, and `ACE`.

PROGRAMMING

1. Define a class `Card`. This class should contain the `Suit` and `Rank` enumerations and two final fields to contain the `Suit` and `Rank`. Define a `toString` method that returns the string *rank* `of` *suit*, and a constructor that takes two `Strings` specifying the rank and suit.

A.10 I/O Using Streams, Class `Scanner`, and Class `JOptionPane`

In this section, we will show you the basics of using streams for I/O in Java. An input stream is a sequence of bytes representing program data. An output stream is a sequence of bytes representing program output. You can store program data in the stream associated with the console, `System.in`. When you type data characters at the console keyboard, they are appended to `System.in`. The console window is associated with `System.out`, the standard output stream. We have used methods `print` and `println` to write information to this stream.

Besides using the console for I/O, you can create and save a text file (using a word processor or editor) and then use it as an input stream for a program. Similarly, a program can write characters to an output stream and save it as a disk file. Classes `BufferedReader` and `FileReader` are subclasses of `Reader`.

Internally Java works with characters. The conversion from bytes to characters is performed by a `Reader` for input, and the conversion from characters to bytes is performed by a `Writer` for output. We discuss the details of this conversion later in this section.

The Scanner

The `Scanner` greatly simplifies the process of reading data from the console or an input file because it breaks its input into *tokens* that are character sequences separated by *whitespace* (blanks and the *newline* character). For console input, the `next` and `hasNext` methods suspend execution until input is provided. Table A.17 summarizes selected methods of this class.

To use a `Scanner` (part of `java.util`) to read from the console, you need to create a new `Scanner` object and connect it to the console (`System.in`)

```
Scanner scanConsole = new Scanner(Sytem.in);
```

We illustrate the use of a `Scanner` in the next example.

TABLE A.17
Selected Methods of the `java.util.Scanner` Class

Constructor	Behavior
`Scanner(File source)`	Constructs a `Scanner` that reads from the specified file
`Scanner(InputStream source)`	Constructs a `Scanner` that reads from the specified `InputStream`
`Scanner(Readable source)`	Constructs a `Scanner` that reads from the specified `Readable` object. The interface `Readable` is the superclass for `Readers`
`Scanner(String source)`	Constructs a `Scanner` that reads from the specified `String object`
Method	**Behavior**
`boolean hasNext()`	Returns `true` if there is another token available for input
`boolean hasNextDouble()`	Returns `true` if the next token can be interpreted as a `double` value
`boolean hasNextInt()`	Returns `true` if the next token can be interpreted as an `int` value
`boolean hasNextLine()`	Returns `true` if there is another line available for input
`IOException ioException()`	Returns the `IOException` last thrown by the `Readable` object that is used to read the input. (The `Scanner` constructors create a `Readable` object to perform the actual input.)
`String next()`	Returns the next token
`double nextDouble()`	Returns the next token as a `double` value. Throws `InputMismatchException` if the input is not in the correct format
`int nextInt()`	Returns the next token as an `int` value. Throws `InputMismatch-Exception` if the input is not in the correct format
`String nextLine()`	Returns the next line of input as a string. A line is a sequence of characters ending with the newline character (`\n`). It may contain several tokens. The newline character is processed, but it is not included in the string
`String findInLine(String pattern)`	Attempts to find the next occurrence of a substring that matches the regular expression defined by `pattern`. Returns the substring if found, or **`null`** if not found.

EXAMPLE A.21 The program in Listing A.4 prompts the user for a name, an integer value, another name, and another integer value. It displays the sum of the two integer values read. A sample interaction follows:

```
For your blended family,
enter the wife's name: Jane Wilson
Enter the number of her children: 3
Enter the husband's name: William Smith
Enter the number of his children: 2
Jane Wilson and William Smith have 5 children.
```

LISTING A.4
A Program for Counting Children in a Blended Family

```
import java.util.Scanner;

/** A class to count and display children in a blended family.
 */
public class BlendedFamily {
    public static void main(String[] args) {
        var sc = new Scanner(System.in);
        System.out.print("For your blended family, \nenter the wife's name: ");
        String wife = sc.nextLine();
        System.out.print("Enter the number of her children: ");
        int herKids = sc.nextInt();

        System.out.print("Enter the husband's name: ");
        sc.nextLine(); // Skip over trailing newline character.
        String husband = sc.nextLine();
        System.out.print("Enter the number of his children: ");
        int hisKids = sc.nextInt();

        System.out.println(wife + " and " + husband + " have "
                           + (herKids + hisKids) + " children.");
    }
}
```

PITFALL

Not Skipping the Newline Character before Reading a String

In the preceding example, we used the statement pair

```
sc.nextLine(); // Skip over trailing newline character.
String husband = sc.nextLine();
```

to read a string into `husband`. The purpose of the first statement is to process the newline character at the end of the line containing the data value 3. If we did not include the first statement, the newline character following the data value 3 would terminate the scanning process immediately, storing an empty string in `husband`. Because data entry for `husband` was completed without the need for typing in additional data, the line

```
Enter the husband's name: Enter the number of his children:
```

would be displayed. At this point, if you enter the husband's name, you will get an `InputMismatchException`. If you enter an integer, you will get the incomplete output line:

```
Jane Wilson and have 5 children.
```

Using a Scanner to Read from a File

You can also use a `Scanner` to read from a file. To do this, you need to create a new `Scanner` object and connect it to the file. The statement below is a **`try`** with resources statement and will be discussed later in this section.

```
String fileName = "dataFile.txt";  // data file name
try (var scanFile = new Scanner(new File(fileName))) {
     // code to read from scanFile
     ...
} catch (FileNotFoundException ex) {
    System.err.println(fileName + " not found");
    System.exit(1);  // Exit with an error indication
}
```

We explain the **`try-catch`** statement in more detail in Section A.11. Before reaching the **`try`** block, the data file name is stored in `String filename`. The code in parentheses at the start of the **`try`** block declares `scanFile` as a `Scanner` object and connects it to data file `fileName`. The `fileName` is passed as an argument to the `File` constructor. Class `File` is in `java.io`, which must be imported (as well as `java.util.Scanner`). When the code within the **`try`** block finishes execution, the `Scanner` and the associated `File` are closed and control is transferred to the statement following the **`try-catch`**. The **`catch`** block executes only if the specified file cannot be found.

Exceptions

Exceptions are program errors that occur during the execution of a program. We will discuss exceptions in great detail in Section A.11. In this section, we will tell you just enough about them to enable you to use streams for I/O.

When you process streams, there is a reasonable chance that a system error will occur. For example, the system may not be able to locate your file, or an error could occur during a file read operation. For this reason, Java requires you to perform all file-processing operations within the **`try`** block of a **`try-catch`** sequence, as follows:

```
try {
    // Statements that perform file-processing operations
...
} catch (IOException ex) {
    ex.printStackTrace(System.err); // Display stack trace
    System.exit(1); // Exit with an error indication
}
```

If all operations in the **`try`** block execute without error, the **`catch`** block is skipped. If an `IOException` or error occurs, the **`try`** block is exited and the **`catch`** block executes. This **`catch`** block simply displays the sequence of method calls that led to the error (starting with the most recent one and working backward) in the console window (`System.err`—the standard error stream) and then exits with an error indication. If we did not exit the **`catch`** block after catching an error, the program would continue with the first statement following the **`catch`** block.

Tokenized Input

Often a data line will consist of a group of data items separated by spaces. In Section A.5, we discussed how to extract the individual items (tokens) from each line in order to process them. You can also use a `Scanner`. The following loop adds all the numbers read from input stream `ins`.

```
double sum = 0.0;
Scanner sc = new Scanner(ins);
while (sc.hasNextDouble()) {
    nextNum = sc.nextDouble();
    sum += nextNum;
}
```

Extracting Tokens Using Scanner.findInLine

You can use a Scanner to scan the characters in a string as well as the data in a file. The statement

```
Scanner scan = new Scanner(line);
```

creates a Scanner object to scan, or process the characters, in string line.

You can also extract substrings that match a specified pattern using Scanner method findInLine. If method findInLine is applied to scan, it will extract each sequence of characters in line matched by its regular expression argument. When there are no characters remaining that match the regular expression, findInLine will return **null**. The statements below store in token each sequence of digit and letter characters and display the tokens. Note that this regular expression is the same as the one used with method split with the ^ (not) symbol removed because we are extracting sequences of letters and digits instead of looking for delimiter characters that are not letters or digits.

```
String token;
while ((token = scan.findInLine("[\\p{L}\\p{N}]+")) != null) {
    System.out.println(token);
}
```

Using a BufferedReader to Read from an Input Stream

To use files as input streams, you must import java.io:

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

You also need to create a BufferedReader object:

```
String fileName = "InputData.txt";
var ins = new BufferedReader(new FileReader(fileName));
```

Although this looks fairly complicated, you can think of it as "boilerplate" (or a template for creating a BufferedReader). The only part of this code that can change is the String argument passed to the FileReader constructor (fileName in this example). The string referenced by filename is the name of the data file for this program and is connected to stream ins by the second statement above. The file InputData.txt is a text file stored in the same directory as the Java program that will read it.

As an alternative to listing the file name in the program, you can pass its name through the args parameter for method main. In the statement

```
String fileName = args[0];   // the first argument
```

variable fileName references the string passed as the first parameter (args[0]) to method main. This should be the name of a data file.

The BufferedReader constructor needs a parameter that is type FileReader (or type InputStreamReader). The BufferedReader class defines a method readLine that can be used to read the next data line in a file (or typed at the console); the method returns a String object that contains the characters in that data line.

Output Streams

To write to an output file, you need to create an output stream. Use statements such as

```
String outFileName = args[1]; // The second main parameter
var outs = new PrintStream(new FileOutputStream(outFileName));
```

Or you can specify the output file name in the first statement above instead of passing it through args[1]. You can apply method print or println to the PrintStream object outs.

The string stored in variable `outFileName` should be the external name of a file. When object `outs` is created, the stream it references is always empty. Any information previously stored in the corresponding disk file will be lost.

Passing Arguments to Method `main`

Earlier we set the variable `fileName` to reference the same string as `args[0]`, the first parameter for method `main`. You must specify the `main` method parameters before you run an application. When you are using the JDK and therefore running your applications from the console command line (such as an "MS-DOS Prompt" window in Windows), you list the parameters after the name of the class you are executing. For example, if you are running application `FileTest.java` with parameters `indata.txt` and `output.txt`, use the command line

```
java FileTest indata.txt output.txt
```

When you are using an IDE, you can also specify parameters before running an application. In IntelliJ, click on Edit Configuration in the Run Menu. This will cause a dialog window to pop-up. In the Program arguments field type the complete path to each file as a string in quotation marks starting with `args[0]`. Use spaces between arguments.

Closing Streams

After processing streams, you must disconnect them from the application. The statement

```
outs.close();
```

does this for stream `outs`. Data to be written to a file is stored in an *output buffer* in memory before it is written to the disk. The `close` statement ensures that any data in the output buffer is written to disk.

PITFALL

Neglecting to Close an Output Stream

If you do not close a stream, it is not considered an error. However, you may find that not all the information written to the stream is actually stored in the corresponding disk file unless you close it.

Try with Resources

Properly closing the I/O streams can be tricky, especially if more than one stream is involved. One or both may not have been created or closed before an exception was thrown. Therefore, the code to close the streams must be in a **`finally`** block (discussed in Section A.11) and needs to check whether the stream was opened and whether an exception was thrown. The **`try`** with-resources syntax was introduced to automatically close any streams when the **`try`** block or a **`catch`** clause is exited either normally or through an exception. The syntax for **`try`** with resources follows and was illustrated earlier in the section "Using a `Scanner` to Read from a File".

```
try (// Declaration of input/output streams) {
     // statements that perform I/O processing
     ...
} catch (IOException ex) {
    ex.printStackTrace();
    System.exit(1);
}
```

A Complete File-Processing Application

We put all these pieces together in this example. In Listing A.5, the `main` method in class `FileTest` consists of a **`try`** with resources sequence. The **`try`** block creates two `BufferedReader` objects: `ins1` (associated with the first data file) and `ins2` (associated with the second data file). It also creates a `PrintStream` object `outs` (associated with an output file). The **`while`** loop alternates reading lines from each data file until all the data from one file is read. It first calls method `readLine` to read a data line from stream `ins1`, storing the information read in the `String` object `first`. When the end of the data file is reached, `first` will contain **`null`**. If `first` is not **`null`** (the normal situation when a data line is read), a line is read into `String` object `second`. If `second` is **`null`**, the loop is exited via the `break` statement.

Otherwise, the contents of `second` are appended to `first`, and the new string is written to the output file.

```
outs.println(first + ", " + second); // Append and write
```

This process continues until the end of either data file is reached, loop exit occurs, and the files are closed.

LISTING A.5

Class `FileTest`

```
import java.io.*;
Class FileTest
public class FileTest {
    /**
       Reads a line from one input file and then from a second input file.
       Concatenates the two lines and writes them to an output file.
       Does this until all input lines have been read from one of the files.
       @param args The command line arguments
       args[0] The first input file name
       args[1] The second input file name
       args[2] The output file name
     */
    public static void main(String[] args) {
       if (args.length < 3) {
           System.err.println("Please provide three file names");
           System.exit(1);
       }
       try (
           var ins1 = new BufferedReader(new FileReader(args[0]));
           var ins2 = new BufferedReader(new FileReader(args[1]));
           var outs = new PrintStream(new FileOutputStream(args[2]);
       ) {
           // Reads words and writes them to the output file until done.
           String first;
           while ((first = ins1.readLine()) != null) { // Read from file1
               String second = ins2.readLine(); // Read from file2
               if (second == null) {
                   break;
               }
               // Append and write
               outs.println(first + ", " + second);
           }
       } catch (FileNotFoundException ex1) {
           System.err.println(ex1);
       } catch (IOException ex2) {
           System.err.println(ex2);
       }
   }
}
```

```
If file1.txt contains the three lines:
    apple
    cat
    John
and if file2.txt contains the three lines:
    butter
    dog
    Doe
then the output file will contain:
    apple, butter
    cat, dog
    John, Doe
```

Input/Output Using Class `JOptionPane`

So far we have discussed console input and input from files. Many of the programs you interact with use dialog windows for input and message windows for output. Class `JOptionPane` (part of the `javax.swing` package) enables this type of interaction. Table A.18 shows two important `static` methods for this class. To use class `JOptionPane`, you should place the line

```
import javax.swing.JOptionPane; // Import class JOptionPane
```

before the class definition in your source file.

TABLE A.18
Methods from Class `JOptionPane`

Method	Behavior
`static String showinputDialog(String prompt)`	Displays a dialog window that shows the argument as a prompt and returns the character sequence typed by the user
`static void show Message Dialog(Object parent, String message)`	Displays a window containing a `message` string (the second argument) inside the specified container (the first argument)

EXAMPLE A.22 The statement

```
String name = JOptionPane.showInputDialog("Enter your name");
```

displays the dialog window shown on the left in Figure A.15. After the OK button is clicked or the Enter key is pressed, variable `name` references a `String` object that stores the character sequence `"Jane Doe"`. If Cancel is clicked, variable `name` stores **`null`**. The statement

```
JOptionPane.showMessageDialog(null, "Your name is " + name);
```

displays the message window shown on the right in Figure A.15. The first argument specifies the parent container in which this window will be placed. When the argument is `null`, the dialog window is placed in the middle of the screen (the window in which the program is executing).

FIGURE A.15
A Dialog Window (Left) and Message Window (Right)

Converting Numeric Strings to Numbers

A dialog window always returns a reference to a string. How can we convert numeric strings to numbers? As we discussed in Section A.6, class `Integer` provides a static method, `parseInt`, for converting strings consisting only of digit characters to numbers, and class `Double` provides a static method, `parseDouble`, for converting strings consisting of the characters for a real number (or integer) to a type `double` value (Table A.19).

TABLE A.19
Methods for Converting Strings to Numbers

Method	Behavior
`static int parseInt(String)`	Returns an `int` value corresponding to its argument string. A `NumberFormatException` occurs if its argument string contains characters other than digits
`static double parseDouble(String)`	Returns a `double` value corresponding to its argument string. A `NumberFormatException` occurs if its argument string does not represent a real number

EXAMPLE A.23 The next pair of statements stores a type `int` value in `numStu` if `answer` references a `String` object that contains digit characters only.

```
String answer = JOptionPane.showInputDialog("Enter number of students");
int numStu = Integer.parseInt(answer);
```

PITFALL

Using Nonnumeric Strings with `parseInt`, `parseDouble`

If you pass to `parseInt` a string that contains characters that are not digit characters, you will get a `NumberFormatException` error. If you pass to `parseDouble` a string that contains characters that can't be in a number, you will also get a `NumberFormatException` error.

GUI Menus Using Method `showOptionDialog`

Another useful method from class `JOptionPane` is method `showOptionDialog`. This method displays a menu of choices with a button for each choice (see Figure A.16). When a button is clicked, the method returns the index of the button pressed (0 for the first button, etc.). The index value can be used in a **`switch`** statement to select an alternative.

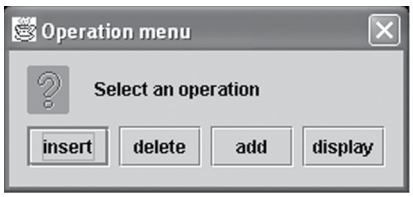
EXAMPLE A.24 The statements

```
String[] choices = {"insert", "delete", "add", "display"}; // button labels
int selection = JOptionPane.showOptionDialog(null,
                "Select an operation", // prompt
                "Operation menu", // title of window
```

```
                JOptionPane.YES_NO_CANCEL_OPTION,
                JOptionPane.QUESTION_MESSAGE, null,
                choices, choices[0]); // button labels start with choices[0]
```

display the menu shown in Figure A.16. The array `choices` defines the button labels. After a button is clicked, the value stored in `selection` will be the index of that button, an integer from 0 to 3.

FIGURE A.16
Displaying a Menu



EXERCISES FOR SECTION A.10

SELF-CHECK

1. Show the statements that would be required, using the console for input, to read and store the data for a `Person` object prior to calling the constructor with four parameters.
2. Answer Exercise 1 above using a data file instead of the console.
3. What would happen if the output file name matched the name of a file already saved on disk? What could happen if the user forgets to close an output file?
4. When does the **`catch`** block in a **`try-catch`** sequence execute?
5. Show the statements that would be required, using `JOptionPane`, to read and store the data for a `Person` object prior to calling the constructor with four parameters.

PROGRAMMING

1. Write a method for class `Person` that reads the data for a single employee from a `BufferedReader` object (the method argument). Assume there is one data item per line.
2. Write a method for class `Company` that reads the data for the `employees` array. This method should call the one needed for Programming Exercise 1.
3. Write a `main` method that reads the data for two `Person` objects, creates the objects, and displays the objects and a message indicating whether they represent the same `Person`.

A.11 Catching Exceptions

When an exception is thrown, the normal sequence of execution is interrupted because the execution of subsequent statements would most likely be erroneous. The default behavior is for the JVM to halt program execution and to display an error message indicating which type of exception was thrown and where in the program it was thrown. The JVM also displays

a stack trace that shows the sequence of method calls, starting at the method that threw the exception, then showing the method that called that method, and so on, all the way back to the `main` method.

The stack trace in Figure A.17 shows that an exception occurred during the execution of class `ExceptionDemo`. The exception was a `NullPointerException`. The exception was thrown in method `doSomethingElse` (at line 18 of class `ExceptionDemo`). Method `doSomethingElse` was called from method `doSomething` (at line 13). Method `doSomething` was called from method `main` (at line 7).

FIGURE A.17
Example of a Stack Trace for an Uncaught Exception

Catching and Handling Exceptions

In the next few subsections, you will see how to avoid the default behavior when you write a method that may throw an exception. You will also see why it is advantageous to do this.

The Try–Catch–Finally Sequence

One way to avoid uncaught exceptions is to write a **`try-catch`** sequence that actually "catches" an exception and "handles it" rather than relying on the JVM to do this.

```
try {
    // Statements that perform file-processing operations.
}
catch (IOException ex) {
    ex.printStackTrace(); // Display stack trace.
    System.exit(1);       // Exit with an error indication.
}
```

If all statements in the **`try`** block execute without error, the **`catch`** block is skipped. If an `IOException` occurs, the **`try`** block is exited and the **`catch`** block executes. This particular **`catch`** block simply displays the sequence of method calls that led to the error (starting with the most recent one and working backward) in the console window (`System.err` – the standard error stream) and then exits with an error indication.

Although this handles the exception, it basically duplicates the default behavior for uncaught exceptions. Next, we show you how to use the **`try-catch`** sequence to recover from errors and continue the execution of your program.

Handling Exceptions to Recover from Errors

In addition to reporting errors, exceptions provide us with the opportunity to recover from errors. One common source of exceptions is user input. For example, the method `JOptionPane.showInputDialog` displays a dialog window and allows the user to enter input. After the user enters input and presses the Enter key, the method will return a string containing

the input characters. If we are expecting an integer value, we need to convert this string to an integer. The conversion is performed by method `parseInt`, which can cause a `NumberFormatException` to be thrown.

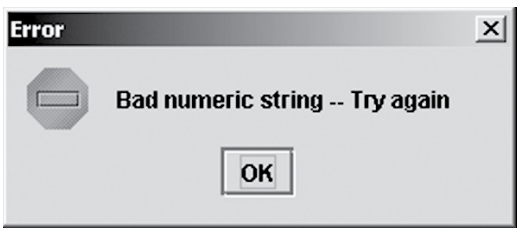
EXAMPLE A.25 Method `readInt` (Listing A.6) returns the integer value that was typed into a dialog window by the program user. The method argument is the dialog window prompt.

The **`while`** loop repetition condition (**`true`**) ensures that the **`try-catch`** sequence will execute "forever" or until the user enters a correct data item. The statements

```
String numStr = JOptionPane.showInputDialog(prompt);
return Integer.parseInt(numStr);
```

display the dialog window and return an integer value if `numStr` contains only digit characters. If not, a `NumberFormatException` is thrown, which is handled by the **`catch`** clause. The **`catch`** block displays an error message window by calling `JOptionPane.showMessageDialog`. The last argument, `JOptionPane.ERROR_MESSAGE`, causes a window with a stop sign to appear (see Figure A.18). After pressing OK, the user has another opportunity to enter a valid numeric string.

FIGURE A.18
Bad Numeric String Error



LISTING A.6
Method `readInt`

```
/** Method to return an integer data value.
    @param prompt Message
    @return The data value read as an int
 */
public static int readInt(String prompt) {
    while (true) { // Loop until valid number is read.
        try {
            String numStr = JOptionPane.showInputDialog(prompt);
            return Integer.parseInt(numStr);
        }
        catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(
                null,
                "Bad numeric string – Try again",
                "Error", JOptionPane.ERROR_MESSAGE);
        }
    }
}
```

The try Block

The syntax for the try block is as follows:

```
try {
    Code that may throw an exception
}
```

The catch Clauses and Blocks

Exceptions are caught by what is appropriately called a **catch** clause. A **catch** clause resembles a method and has the following syntax.

```
catch (ExceptionClass exceptionArgument) {
    Code to handle the exception
}
```

The code within the brackets is called the **catch** block. The **catch** clause(s) must follow a **try** block. There may be multiple **catch** clauses, one for each exception class that you wish to handle.

An exception matches a **catch** clause if the type of the exception is the same as the argument of the **catch** clause or is a subclass of the argument type. When an exception is thrown from within a **try** block, the associated **catch** clause(s) is (are) examined to see whether there is a match in the exception class for any **catch** clause. If so, that **catch** block executes. If not, a search is made back through the chain of method calls to see whether any of them occurs in a **try** block with an appropriate catch. If so, that **catch** block executes. We illustrate this next.

EXAMPLE A.26 The body of method `readIntTwo` below contains just the statements in the **try** block of method `readInt` (Listing A.6), but the **catch** clause is omitted.

```
public static int readIntTwo(String prompt) {
    String numStr = JoptionPane.showInputDialog(prompt);
    return Integer.parseInt(numStr);
}
```

In this case, if a `NumberFormatException` is thrown by method `parseInt`, method `readIntTwo` will not be able to handle it, so `readIntTwo` is exited and a search is made for an appropriate **catch** clause in the caller of method `readIntTwo`. If method `readIntTwo` is called to read a value into age by the following **try** block:

```
try {
    // Enter a value for age.
    Age = readIntTwo("Enter your age");
} catch (Exception ex) {
    System.err.println("Error occurred in call to readIntTwo");
    age = DEFAULT_AGE;
}
```

the **catch** clause will handle the `NumberFormatException` (because `NumberFormatException` is a subclass of `Exception`) and assign the value of `DEFAULT_AGE` to age. It will also display an error message on the console, and program execution will continue with the statement that follows this **try–catch** sequence.

Note that a **catch** clause is like a method, and the **catch** block is like a method body. The term **catch** clause refers to both the header and the body, whereas the term **catch** block refers to

the body alone. This distinction is not generally that important, and you may see the terms used interchangeably in other texts and documentation.

EXAMPLE A.27 `EOFException` is a subclass of `IOException`. The two **`catch`** clauses in the following code must appear in the sequence shown to avoid a `catch is unreachable` syntax error. The first **`catch`** block handles an `EOFException` that occurs when all the data in the file was processed (not an error). This **`catch`** block tells the program user so and then exits the program normally (`System.exit(0)`). The second **`catch`** clause processes any other I/O exception by calling method `printStackTrace` to display a stack trace like the one shown in Figure A.17. It exits the program with an error indication (`System.exit(1)`). (*Note:* The method `printStackTrace` is defined in the `Throwable` class and is inherited by all `Exception` objects.)

```
catch (EOFException ex) {
    System.out.print("End of file reached ")
    System.out.println(" - processing complete");
    System.exit(0);
}
catch (IOException ex) {
    System.err.println("Input/Output Error:");
    ex.printStackTrace();
    System.exit(1);
}
```

PITFALL

Unreachable `catch` Block

Note that only the `catch` block within the first `catch` clause having an appropriate exception class executes. All other `catch` blocks are skipped. If a `catch` clause exception type is a subclass of an exception type in an earlier `catch` clause, the `catch` block in the later `catch` clause cannot execute, so the Java compiler will display a `catch is unreachable` syntax error. To correct this error, switch the order of the `catch` clauses so that the `catch` clause whose exception class is the subclass comes first.

The `finally` Block

When an exception is thrown, the flow of execution is suspended and continues at the appropriate `catch` clause. There is no return to the `try` block. Instead, processing continues at the first statement after all of the `catch` clauses associated with the `try` from which the exception was thrown.

In some situations, allowing the program to continue after an exception without executing all the statements in the `try` block could cause problems. For example, if some calculations needed to be performed before returning from the method, these calculations would have to be duplicated in both the `try` block and every `catch` clause. To avoid such a duplication (which can be error-prone), the `finally` block can be used. The code in the `finally` block is executed either after the `try` block is exited or after a `catch` clause is exited (if one is executed). The `finally` block is optional. We show an example in the following syntax summary.

SYNTAX try-catch-finally Sequence

FORM:

```
try {
    Statements that may throw an exception
}
catch (ExceptionClass1 exceptionArgument1) {
    Statements to process ExceptionClass1
}
catch (ExceptionClass2 exceptionArgument2) {
    Statements to process ExceptionClass2
}
catch (ExceptionClassn exceptionArgumentn) {
    Statements to process ExceptionClassn
}
finally {
    Statements to be executed after the try block or a catch clause exits
}
```

EXAMPLE:

```
try {
    String sizeStr = JOptionPane.showInputDialog("Enter new size");
    size = Integer.parseInt(sizeStr);
}
catch (NumberFormatException ex) {
    size = DEFAULT_SIZE; // Use default value if input error.
}
finally {
    if (size > MAX_CAPACITY)
        size = MAX_CAPACITY;
}
```

MEANING:

The statements in the `try` block execute through to completion unless an exception is thrown. If there is a `catch` clause to handle the exception, its `catch` clause executes to completion. After the `try` block or `catch` clause executes, the `finally` block executes to completion.

If there is no `catch` clause to handle the exception thrown in the `try` block, the `finally` block is executed, and then the exception is passed up the call chain until either it is caught by some other method in the call chain or it is processed by the JVM as an uncaught exception.

Reporting the Error and Exiting

There are many cases in which an exception is thrown but there is no obvious way to recover. For example, reading from a file or the system console can result in an `IOException` being thrown. In this case, the **`catch`** clause should print the stack trace and exit, as follows:

```
catch (IOException ex) {
    ex.printStackTrace();
    System.exit(1);
}
```

The method call `System.exit(1)` causes a return to the operating system with an error indication.

Checked and Unchecked Exceptions

There are two categories of exceptions: checked and unchecked. A *checked exception* is caused by an error that is beyond the programmer's control, such as an input/output error (`IOException`). An *unchecked exception* is caused by a program error. An example is an `IndexOutOfBoundsException`. Checked exceptions must always be handled in some way (discussed next). There is no requirement to handle unchecked exceptions.

PITFALL

Ignoring Exceptions

Exceptions are designed to make the programmer aware of possible error conditions and to provide a way to handle them. Some programmers do not appreciate this feature and do the following:

```
catch (Exception e){}
```

Although this clause is syntactically correct and eliminates a lot of pesky error messages, it is almost always a bad idea. The program continues execution after the `try-catch` sequence with no indication that there was a problem. The statement that caused the exception to be thrown did not execute properly. The statements that follow it in the `try` block were not executed at all. The program will have hidden defects that will make its users very unhappy and could have even more serious consequences.

PROGRAM STYLE

Using Exceptions to Enable Straightforward Code

In computer languages that did not provide exceptions, programmers had to incorporate error-checking logic throughout their code to check for many possibilities, some of which were of low probability. The result was sometimes messy, as follows:

```
Step A
if (Step A successful) {
    Step B
    if (Step B successful) {
        Step C
    } else {
        Report error in Step B
        Cleanup after Step A
    }
} else {
    Report error in Step A
}
```

With exceptions this becomes much cleaner, as follows:

```
try {
    Step A
    Step B
    Step C
} catch (exception indicating Step B failed) {
    Report error in step B
    Cleanup after step A
} catch (exception indicating Step A failed) {
    Report error in step A
}
```

EXERCISES FOR SECTION A.11

SELF-CHECK

1. Assume that method `main` calls method `first` at line 10 of class `MyApp`, method `first` calls method `second` at line 10 of class `Others`, and method `second` calls method `parseInt` at line 20 of class `Other`. These calls result in a `NumberFormatException` at line 430 of class `Integer`. Show the stack trace.
2. Assume that you have **`catch`** clauses for exception classes `Exception`, `NumberFormatException`, and `RuntimeException` following a **`try`** block. Show the required sequence of **`catch`** clauses.

PROGRAMMING

1. For the try block

```
try {
    numStr = in.readLine();
    num = Integer.parseInt(numStr);
    average = total / num;
}
```

write a `try-catch-finally` sequence with **`catch`** clauses for `ArithmeticException`, `NumberFormatException`, and `IOException`. For class `ArithmeticException`, set average to zero and display an error message indicating the kind of exception, display the stack trace, and exit with an error indication. After exiting the **`try`** block or the **`catch`** block for `ArithmeticException`, display the message `"That's all folks"` in the **`finally`** block.

A.12 Throwing Exceptions

In the previous section, we showed how to catch and handle exceptions using the **`try-catch`** sequence. As an alternative to catching an exception in a lower-level method, you can allow it to be caught and handled by a higher-level method. You can do this in one of two ways:

1. You declare that the lower-level method may throw a checked exception by adding a `throws` clause to the method header.
2. You throw the exception in the lower-level method, using a `throw` statement, when the exception is detected.

The throws Clause

The next example illustrates the use of the **`throws`** clause to declare that a method may throw a particular kind of checked exception. This is a useful approach if a higher-level module already contains a **`catch`** clause for this exception type. If you don't use the **`throws`** clause, you must duplicate the **`catch`** clause in the lower-level method to avoid an unreported exception syntax error.

EXAMPLE A.28 Method `readData` reads two strings from the `BufferedReader` `console` associated with `System.in` (the system console) and stores them in data fields `firstName` and `lastName`. Each call to method `readLine` may throw a checked `IOException`, so method `readData` cannot compile without the `throws` clause. If you omit it, you will get the syntax error `unreported exception: Java.io.IOException; must be caught or declared to be thrown.`

```
public void readData() throws IOException {
    var console = new BufferedReader(
                  new InputStreamReader(System.in));
    System.out.print("Enter first name: ");
    firstName = console.readLine();
    System.out.print("Enter last name: ");
    lastName = console.readLine();
}
```

If method `readData` is called by method `setNewPerson`, method `setNewPerson` must have a **`catch`** block that handles exceptions of type `IOException`.

```
public void setNewPerson() {
    try {
        readData();
        // Process the data read.
        . . .
    } catch (IOException iOEx) {
        System.err.println("Call to readLine failed in readData");
        iOEx.printStackTrace();
        System.exit(1);
    }
}
```

If a method can throw more than one exception type, list them all after `throws` with comma delimiters. You will get an `unreported exception` syntax error if you omit any checked exception type. The compiler verifies that all class names listed are exception classes.

PROGRAM STYLE

Using Javadoc @throws for Unchecked Exceptions

Listing unchecked exceptions in the `throws` clause is legal syntax but is considered poor programming practice. Instead you should use the `Javadoc @throws` tag to document any unchecked exceptions that may reasonably be expected to occur but are not caught in the method.

The throw Statement

You can use a **`throw`** statement in a lower-level method to indicate that an error condition has been detected. When the **`throw`** statement executes, the lower-level method stops executing immediately, and the JVM begins the search for an exception handler as described earlier. This approach is usually taken if the exception is unchecked and is likely to be caught in a higher-level method. If the exception thrown is a checked exception, this exception must be declared in the **`throws`** clause of the method containing the **`throw`** statement.

EXAMPLE A.29 The method `addOrChangeEntry` takes two `String` parameters: `name` and `number`. The `number` parameter is intended to represent a valid phone number. Therefore, we wish to validate its format to ensure that only validly formatted numbers are entered. Assuming that we have a method `isPhoneNumberFormat` that checks for a valid phone number, we could code the `addOrChangeEntry` method as follows:

```
public String addOrChangeEntry(String name, String number) {
    if (!isPhoneNumberFormat(number)) {
        throw new IllegalArgumentException("Invalid phone number: " + number);
    }
    // Add/change the number.
    . . .
}
```

The `throw` statement creates and throws a new `IllegalArgumentException`, which can be handled farther back in the call chain or by the JVM if it is uncaught. The constructor argument (`"Invalid phone number: " + number`) for the new exception object is a message that describes the cause of the error.

If we call this method using the following **`try-catch`** sequence:

```
try {
    addOrChangeEntry(myName, myNumber);
} catch (IllegalArgumentException ex) {
    System.err.println(ex.getMessage());
}
```

and `myNumber` references the string `"1xx1"`, which is not a valid phone number, the console output would be

```
Invalid phone number: 1xx1
```

SYNTAX throw Statement

FORM:

```
throw new ExceptionClass();
throw new ExceptionClass(detailMessage);
```

EXAMPLE:

```
throw new FileNotFoundException("File " + fileSource + " not found");
```

MEANING:

A new exception of type *ExceptionClass* is created and thrown. The optional `String` parameter *detailMessage* is used to specify an error message associated with this exception. If the higher-level method that catches this exception has the `catch` clause

```
catch (ExceptionClass ex) {
    System.err.println(ex.getMessage());
    System.exit(1);
}
```

the *detailMessage* will be written to the system error stream before system exit occurs.

EXAMPLE A.30 Listing A.7 shows a second method `readInt` that has three arguments. As in method `readInt` in Listing A.6, the first argument is a prompt. The second and third arguments represent the end points for a range of integer numbers. The method returns the first integer value entered by the program user that is between the end points.

The `if` statement tests whether the end points define an empty range (`minN > maxN`). If so, the statement

```
throw new IllegalArgumentException("In readInt, minN " + minN +
                                   " not <= maxN " + maxN);
```

throws an `IllegalArgumentException`, creating an instance of this class. The message passed to the constructor gives the cause of the exception. This message would be displayed by `printStackTrace` or returned by `getMessage` or `toString`.

If the range is not empty, the while loop executes. Its repetition condition (`!inRange`) is **`true`** as long as the user has not yet entered a value that is within the range defined by the end points. The **`try`** block displays a dialog window with a prompt that shows the valid range of values. The statement

```
inRange = (minN <= n && n <= maxN);
```

sets `inRange` to **`true`** when the value assigned to `n` is within this range. If so, the loop is exited and this value is returned. However, if the user enters a string that is not numeric, the **`catch`** block displays an error message. If the string is not numeric or its value is not in range, `inRange` remains **`false`**, so (`!inRange`) is **`true`** and the loop repeats, giving the user another opportunity to enter a valid number.

LISTING A.7

Method **`readInt`** (part of `MyInput.java`) with Three Parameters

```
/** Method to return an integer data value between two
    specified end points.
    @param prompt Message
    @param minN Smallest value in range
    @param maxN Largest value in range
    @throws IllegalArgumentException
    @return The first data value that is in range
 */
public static int readInt(String prompt, int minN, int maxN) {
    if (minN > maxN) {
        throw new IllegalArgumentException("In readInt, minN " + minN
                                           + " not <= maxN " + maxN);
    }
    // Arguments are valid, read a number.
    boolean inRange = false; // Assume no valid number read.
    int n = 0;
    while (!inRange) { // Repeat until valid number read.
        try {
            String line = JOptionPane.showInputDialog(
                          prompt + "\nEnter an integer between "
                          + minN + " and " + maxN);
            n = Integer.parseInt(line);
            inRange = (minN <= n && n <= maxN);
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(
                null,
                "Bad numeric string - Try again",
                "Error", JOptionPane.ERROR_MESSAGE);
        }
    }   // End while
    return n; // n is in range
}
```

PROGRAM STYLE

Reasons for Throwing Exceptions

You might wonder what is gained by intentionally throwing an exception. If it is not caught farther back in the call chain, it will go uncaught and will cause your program to terminate. However, in the examples in this section, it would not make any sense to continue with either an empty range (in `readInt`) or an invalid phone number (in `addOrChangeEntry`). In fact, the loop in method `readInt` would execute forever if the range of acceptable values was empty. Because the boundary parameters `minN` and `maxN` are defined in a higher-level method, it would also make no sense to try to get new values in `readInt`. However, if the exception is passed back and caught at the point where the boundary points are defined, the programmer can get new boundary values and call method `readInt` again instead of terminating the program.

Catching versus Throwing Exceptions

You can always avoid handling exceptions where they occur by declaring that they are thrown or by throwing them and letting them be handled farther back in the call chain. In general, though, it is better to handle an exception where it occurs rather than to pass it back. This gives you the opportunity to recover from the error and to continue on with the execution of the current method. We did this, for example, for `NumberFormatExceptions` in both `readInt` methods (see Listings A.6 and A.7). If an error is a nonrecoverable error, however, and is also likely to occur farther back in the call chain, you might as well allow the exception to be handled at the farthest point back in the call chain rather than duplicate the error-handling code in several methods. We recommend the following guidelines:

- If an exception is recoverable in the current method, handle the exception in the current method.
- If a checked exception is likely to be caught in a higher-level method, declare that it can occur using a **`throws`** clause, and use a @throws tag to document this in the Javadoc comment for this method.
- If an unchecked exception is likely to be caught in a higher-level method, use a @throws tag to document this fact in the Javadoc comment for the method. However, it is not necessary to use a **`throws`** clause with unchecked exceptions.

EXERCISES FOR SECTION A.12

SELF-CHECK

1. Explain the difference between the `throws` clause and the `throw` statement.
2. When would it be better to declare an exception rather than catch it in a method?
3. When would it be better to throw an exception rather than catch it in a method?
4. What kind of exceptions should appear in a `throws` clause?

5 For the following situations, indicate whether it would be better to catch an exception, declare an exception, or throw an exception in the lower-level method. Explain your answer and show the code required for the lower-level method to do it.

a. A lower-level method contains a call to method `readLine`; the higher-level method that calls it contains a **`catch`** clause for class `IOException`.

b. A method contains a call to method `readLine` to enter a value that is passed as an argument to a lower-level method. The lower-level method's argument must be a positive number.

c. A lower-level method contains a call to method `readLine`, but the higher-level method that calls it does not have a **`catch`** clause for class `IOException`.

d. A lower-level method reads a data string and converts it to type `int`. The higher-level method contains a **`catch`** clause for class `NumberFormatException`.

e. A lower-level method detects an unrecoverable error that is an unchecked exception.

PROGRAMMING

1. The syntax display for the `throw` statement had the following example: `throw new FileNotFoundException("File " + fileSource + " not found");` Write a **`catch`** clause for a method farther back in the call chain that handles this exception.

2. Method `setElementOfX` shown below validates that the parameters `index` and `val` are in bounds before accessing array `x`. Rewrite this method so that it throws exceptions during array access if `val` is out of bounds or if `index` is out of bounds. Pass an appropriate detail message to the new exception object. Your modified method should be type **`void`** because there is no longer a reason to return a **`boolean`** error indicator. Show **`catch`** blocks for a higher-level method that would handle these exceptions.

```
public boolean setElementOfX(int index, int val) {
    if (index >= 0 && index < x.length
            && val >= MIN_VAL && val <= MAX_VAL) {
        x[index] = val;
        return true;
    } else {
        return false;
    }
}
```

Appendix Review

- A Java program is a collection of classes. A programmer can use classes defined in the Java API to simplify the task of writing new programs and can define new classes to use as building blocks in future programs. Use

  `import packageName.*;`

  or

  `import packageName.ClassName;`

  to make the public names defined in a package or class accessible to the current file.

- The JVM enables a Java program written for one machine to execute on any other machine that has a JVM. The JVM is able to execute instructions that are written in Java byte code. The byte code instructions are found in the `.class` file that is created when a Java source file is compiled.

- Java defines a set of primitive data types that are used to represent numbers (**int**, **double**, **float**, etc.), characters (**char**), and **boolean** data. Characters are represented using Unicode. Primitive-type variables are used to store primitive data. The Java programmer can use reference variables to reference objects. Wrapper classes can be used to encapsulate (wrap) a primitive-type value in an object.
- The control structures of Java are similar to those found in other languages: sequence (a compound statement or block), selection (**if** and **switch**), and repetition (**while**, **for**, **do ... while**).
- There are two kinds of methods: `static` (or class) methods and instance methods. Static methods are called using

  ClassName.methodName(arguments)

  but instance methods must be applied to objects:

  objectReference.methodName(arguments)
- The `Java String`, `StringBuilder`, `StringJoiner`, and `StringBuffer` classes are used to reference objects that store character strings. `String` objects are immutable, which means they can't be changed, whereas `StringBuilder`, `StringJoiner`, and `StringBuffer` objects can be modified.
- Make sure you use methods such as `equals and compareTo` to compare the contents of two `String` objects (or any objects). The `operator == compares` the addresses of two objects, not their contents.
- You can use the `String.format` method or `Formatter` objects to create and display formatted strings.
- You can declare your own Java classes and create objects (instances) of these classes using the **new** operator. A constructor call must follow the **new** operator. A constructor has the same name as its class, and a class can define multiple constructors. The no-parameter constructor is defined by default if no constructors are explicitly defined.
- A class has data fields (instance variables) and instance methods. The default values for data fields are 0 or 0.0 for numbers, `\u0000` for characters, **false** for **boolean**, and **null** for reference variables. A constructor initializes data fields to values specified by its arguments. Generally, data fields have private visibility (accessible only within the class), whereas methods have public visibility (accessible outside the class).
- Array variables can reference array objects. You must use the **new** operator to allocate storage for the array object.

  ```
  int[] anArray = new int[mySize];
  ```

  The elements of an array can store primitive-type values or references to other objects. Arrays of arrays (multidimensional arrays) are permitted. The data field `length` represents the size of an array and is always accessible, but `length` can't be changed by the programmer. However, an array variable can be reset to reference a different array object with a different size.
- Class `JOptionPane` (part of `javax.swing`) can be used to display dialog windows for data entry (method `showInputDialog`) and message windows for output (method `showMessageDialog`).
- The `Scanner` class in `java.util` can be used to read numbers and strings from the console (`System.in`) using methods `nextInt`, `nextDouble`, and `nextLine`.
- The stream classes in `java.io` can enable you to read data from input files and write data to output files. Use statements like

  ```
  var ins1 = new Scanner(new File(inputFileName));
  ```

or

```
var ins2 = new BufferedReader(new FileReader(FileName));
```

for input files. For output files use

```
var outs = new PrintWriter(new FileWriter(outFileName));
```

These statements associate the input streams `ins1`, `ins2` and the output stream `outs` with specified files. Many file operations must be performed within a **`try-catch`** sequence that catches `IOException` exceptions. You must close an output file when you have finished writing all information to it. The **`try`** with resources statement enables this to be done automatically.

- The default behavior for exceptions is for the JVM to catch them by printing an error message and a call stack trace and then terminating the program. You can use the **`try-catch-finally`** sequence to catch and handle exceptions, possibly to recover from the error and continue, thereby avoiding the default behavior.
- There are two categories of exceptions: checked and unchecked. Checked exceptions are generally due to an error condition external to the program. Unchecked exceptions are generally due to a programmer error or a dire event.
- A method that can throw a checked exception must either catch it or declare that it is thrown using the `throws` declaration. If you throw it, you must catch it further back in the call sequence. Methods do not have to catch unchecked exceptions, and they should not be declared in the `throws` clause.
- Use the `throw` statement to throw an unchecked exception when you detect one in a method. You should catch this exception farther back in the call sequence, or it will be processed by the JVM as an uncaught exception.

Java Constructs Introduced in This Appendix

```
boolean
catch
char
class
double
final
finally
for
int
main
new
private
public
static
throw
throws
try
while
```

Java API Classes Introduced in This Appendix

```
java.io.BufferedReader
java.io.FileReader
java.io.InputStreamReader
java.io.IOException
java.io.OutputStreamWriter
java.io.PrintWriter
java.lang.Boolean
java.lang.Character
java.lang.Double
java.lang.Exception
java.lang.FileWriter
java.lang.Math
java.lang.Integer
java.lang.NumberFormatException
java.lang.Object
java.lang.String
java.lang.StringBuffer
java.lang.StringBuiIder
java.util.Formatter
java.util.InputMismatchException
java.util.Scanner
java.util.StringJoiner
javax.swing.JOptionPane
```

User-Defined Interfaces and Classes in This Appendix

`Company`	`HelloWorld`	`SquareRoots`
`FileTest`	`Person`	`TestPerson`

Quick-Check Exercises

1. The Java compiler translates Java source code to _____, which are executed by the _____.
2. A Java program is a collection of _____. Execution of a Java application begins at method _____.
3. Java classes declare _____ and _____. Generally, the _____ have public visibility and the _____ have private visibility.
4. An _____ method is invoked by applying it to an _____; a _____ method is not.
5. If you use the operator == with `objects`, you are comparing their _____, not their _____.
6. To associate an input stream with the console, you must pass the argument _____ to the constructor for a _____ object.
7. To associate an output stream with the console, you must wrap a _____ object in a _____ object.
8. To associate an input stream with a text file, you must wrap a _____ object in a _____ object.
9. Method _____ of class `JOptionPane` normally has _____ as its first argument and a _____ as its second argument.

Review Questions

1. Discuss how a Java source file is processed prior to execution and why this approach makes Java platform independent.
2. Declare storage for an array of arrays that will store a list of integers in its first row, the squares of all but the last integer in its second row, and the cubes of all but the last two integers in its third row. Assume that the size of the first row and its integer values are entered by the program user. Read this data into the array and store the required squares and cubes in the array.
3. Draw diagrams that illustrate the effect of each of the following statements.

```
String s1 = "woops";
String s2 = new String(s1);
String s3 = s1;
s1 = new String("Oops!");
```

   What are the values of `s1 == s2`, `s1 == s3`, and `s2 == s3`? What are the values of `s1.equals(s2)`, `s1.equals(s3)`, and `s2.equals(s3)`? What are the values of `s1.compareTo(s2)`, `s1.compareTo(s3)`, and `s2.compareTo(s3)`?
4. Write a class `Fraction` with integer `numerator` and `denominator` data fields. The default value of `denominator` should be 1. Define a constructor with two arguments for this class and one with just one argument (the value of the numerator). Define a method `multiply` that multiplies this `Fraction` object with the one specified by its argument and returns a new `Fraction` object as its result. Define a method `toDecimal` that returns the value of the fraction as a decimal number (be careful about integer division). Define a `toString` method for this class that represents a `Fraction` object as a string of the form *numerator* / *denominator*.

5. Write a `main` method that reads two `Fraction` objects using class `JOptionPane`. Multiply them and display their result as a fraction and as a decimal number using the instance methods defined in Review Question 4. Use class `JOptionPane` to display the results.

6. Write a `main` method that reads two `Fraction` objects from the console. Multiply them and display their result as a fraction and as a decimal number using the instance methods defined in Review Question 4. Use the console to display the results.

Programming Projects

1. Complete the definition of the `Fraction` class described in Review Question 4. Provide all the methods listed in that question and methods to add, subtract, and divide two fractions. Also, provide methods `equals` and `compareTo` to compare two `Fraction` objects.

2. Provide a class `MatrixOps` that has a two-dimensional array of **`double`** values as its data field. Provide the following methods:

```
MatrixOps() // Default constructor
MatrixOps(int numNows)  // Sets the number of rows
MatrixOps(int numRows, int numCols) // Sets the number of rows and columns
MatrixOps(double[][] mat) // Stores the specified array
void setMatrix(double[][] mat) // Stores the specified array
double[][] getMatrix()  // Gets the array
void setRow(int row, double[] rowVals) // Stores the array of rowVals in row
double[] getRow(int row)  // Returns the specified row
void setElement(int row, int col) // Sets the specified element
double getElement(int row, int col) // Returns the specified element
double sum() // Returns sum of the values in the array
double findMax() // Returns the largest value in the array
double findMin() // Returns the smallest value in the array
double[][] transpose() // Returns the transpose of the matrix
double[][] multiply(double[][] mat2) // Returns the product of two matrices
String toString() // Returns a string representing the array
```

3. Modify class `Person` to include a person's hours worked and hourly rate as data fields. Provide modifier and accessor methods for the new data fields and a method `calcSalary` that returns a person's salary. Also, modify method `toString`. Provide a method `calcPayroll` for class `Company` that returns the weekly payroll amount for a company (gross payroll only; don't be concerned about withholding, payroll taxes, etc.). Write a `main` method that reads the employee data for a `Company` object from a data file. Display the data stored and the calculated payroll in a message window using class `JOptionPane`.

4. Write a class that stores a collection of exam scores in an array. Provide methods to find the average score, to assign a letter grade based on a standard scale, and to display the scores. Test the methods of this class.

5. Write a class `Student` that stores a person's name, an array of scores for each person, an average exam score, and a letter grade. Write a class `Gradebook` that stores an instructor's name, a section ID, a course name, and an array of `Student` records. Write the following methods to process this array.
   - Load the array of `Student` records with data read from a text file.
   - Write all information stored to an output file.
   - Calculate and store each student's average exam score in that student's record.
   - Calculate and store the average score for each student in that student's record.
   - Assign a letter grade to each student based on that student's average exam score.

Write a client program that reads the data for a class and performs all the operations in the list above. Display the information in a `Gradebook` object after all the data is stored and again after all student information been calculated and stored.

Answer to Quick-Check Exercises

1. The Java compiler translates Java source code to *byte code instructions,* which are executed by the *JVM*.
2. A Java program is a collection of *classes*. Execution of a Java application begins at method *main*.
3. Java classes declare *data fields* and *methods*. Generally, the *methods* have public visibility and the *data fields* have private visibility.
4. An *instance* method is invoked by applying it to an *object*; a *static* (or *class*) method is not.
5. If you use the operator == with objects, you are comparing their *addresses*, not their *contents*.
6. To associate an input stream with the console, you must pass the argument `System.in` to a constructor for a `Scanner` object.
7. To associate an output stream with the console, you must wrap a `FileWriter` object in a `PrintWriter` object.
8. To associate an input stream with a text file, you must wrap a `FileReader` object in a `BufferedReader` object.
9. Method `showMessageDialog` of class `JOptionPane` normally has **`null`** as its first argument and a *prompt string* as its second argument.

Appendix B

Overview of UML

The Unified Modeling Language (UML) represents the unification of earlier object-oriented design modeling techniques. Specifically, notations developed by Grady Booch, Ivar Jacobson, and James Rumbaugh were adapted to form the initial version. This version was submitted to the Object Modeling Group for formal standardization. Since that initial submission, the UML standard has undergone several revisions and continues to be revised.

UML defines several diagrams, but in this text we will only use the Class Diagram and the Object Diagram. We use a notation that has been adapted from the UML standard to match the syntax of Java more closely.

Overview of UML

B.1 The Class Diagram

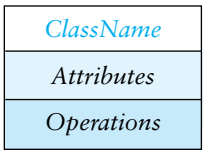
The *class diagram* shows the classes and their relationships. It is a static diagram that represents the structure of the program. The classes (including interfaces) are represented by rectangles, and lines between the classes represent the relationships. The style of a line, symbols on the ends of the lines, and text placed near the line are used to indicate the kind of relationship being modeled.

A large amount of information about the structure of a program can be represented in a class diagram. If all the possible information was presented, the diagram would become quite cluttered. Therefore, the practice is to show only the essential information. For example, in a class diagram, the complete method declaration can show the method's visibility, return type, name, and parameter types. Sometimes, only the method's name is necessary, in which case you would elect to suppress the other information. Also, some methods may not be significant to the discussion, so those methods need not be shown. Sometimes, only the class name is the essential item, and thus the methods and attributes are not shown.

Representing Classes and Interfaces

A class is represented by a rectangle divided into three segments as shown in Figure B.1.

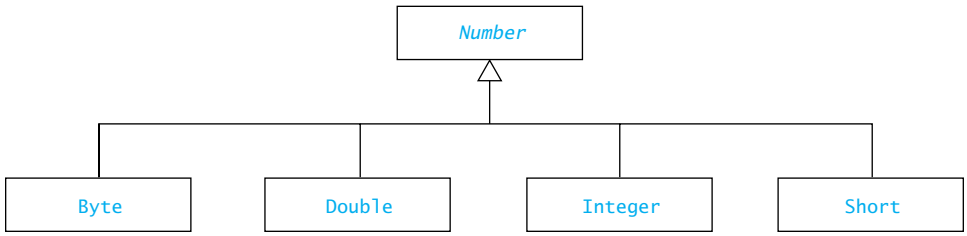
FIGURE B.1 General Representation of a Class



The Class Name

Every class has a name that distinguishes it from other classes. In Java, a class may be (and usually is) a member of a package, in which case we may show the complete name including the package name (e.g., `java.util.Stack`). In other cases, we just show the class name (e.g., `Node`). Italics indicate abstract classes. The class name is centered in the box representing the class. For example, Figure B.2 shows the abstract class `Number` and the concrete classes derived from it.

FIGURE B.2 The Abstract Class `Number` and Concrete Subclasses



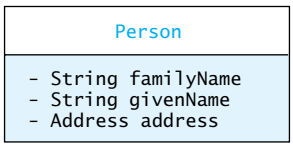
Interfaces

The word *interface* enclosed in double angle brackets (« and », called guillemets) placed before the class name is used to indicate that this class is an interface. Because interfaces, like abstract classes, cannot be instantiated, the name is shown in italics (see Figure B.3).

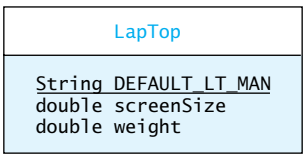
FIGURE B.3 The Interface `List`

The Attributes

The *attributes* of a class are the data fields. As a minimum we show the name. Optionally we can also show the visibility and type. We indicate visibility using the symbols + (public), – (private), and # (protected). In this text, we use the Java language syntax to indicate the type of an attribute by placing the type name before the attribute name. For example, the class `Person` could have the private attributes `familyName`, `givenName`, and `address`, as shown in the following figure:



Where it is not essential to the current discussion, we will omit the visibility indicator, the type, or both. Static attributes are indicated by underlining their name. For example, the class `LapTop` has the static attribute `DEFAULT_LT_MAN`.

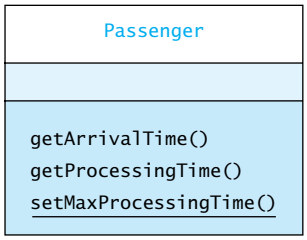


The Operations

The *operations* are the methods of the class. At a minimum, we show the method name followed by a pair of parentheses. An empty set of parentheses does not necessarily indicate that this method takes no parameters. Italics are used to indicate an abstract method, and underlining is used to indicate a static method. For example, Figure B.4 shows the class `Passenger`

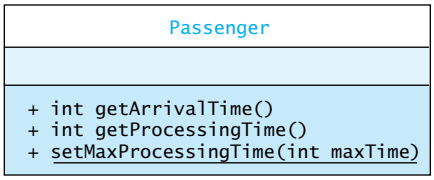
with the static method `setMaxProcessingTime` and the nonstatic methods `getArrivalTime` and `getProcessingTime`. The attributes are not shown.

FIGURE B.4
The Class `Passenger`



We may also show the visibility, the parameter types, and the return type. The visibility is shown using the same symbols as used for the attributes. In this text, we use the Java method declaration syntax, as shown in Figure B.5, to show the parameter types and return type. A return type of `void`, however, will not be shown.

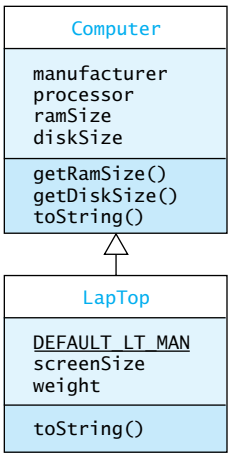
FIGURE B.5
Class `Passenger` Showing Visibility and Return and Parameter Types of Its Operations



Generalization

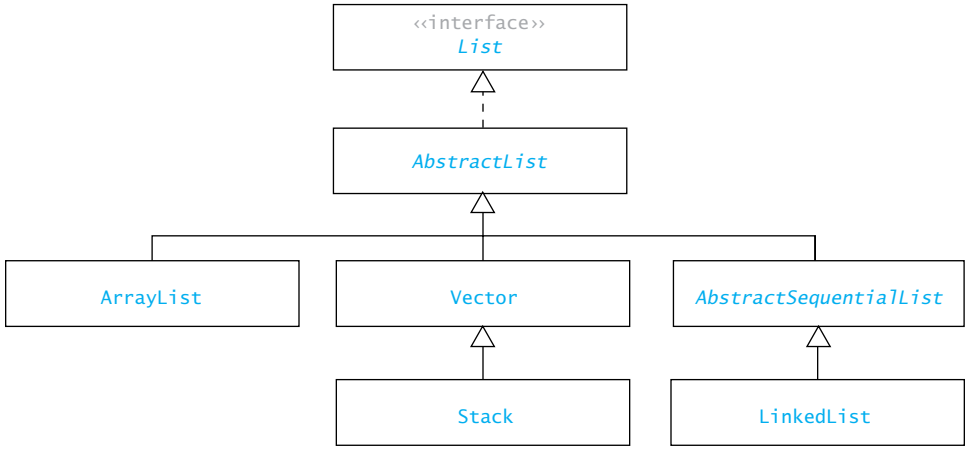
UML uses the term *generalization* to describe the relationship between a superclass and its subclasses. Drawing a solid line with a large open arrowhead pointing to the superclass shows generalization. Figure B.6 shows the class `LapTop` as a subclass of `Computer`.

FIGURE B.6
Class `Laptop` as a Subclass of `Computer`



A dashed line with a large open arrowhead is used to show that a class implements an interface. Figure B.7 shows that the abstract class `AbstractList` implements the `List` interface and that

FIGURE B.7
The `List` Interface and Classes that Implement It



the classes ArrayList, Vector, and AbstractSequentialList are subclasses of AbstractList. Stack is a subclass of Vector and LinkedList is a subclass of AbstractSequentialList.

Inner or Nested Classes

A class that is declared within the body of another class is called an inner or nested class. In UML, this relationship is indicated by a solid line between the two classes, with what the UML standard calls an *anchor* on the end connected to the enclosing class. The anchor is a cross inside a circle. For example, in Figure B.8, the class Node is declared as an inner class of the class KWLinkedList.

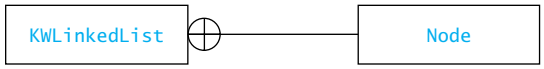


FIGURE B.8
Node as an Inner Class

Aggregation and Composition

In those cases where we wish to show that a class is a data field in another class, we place a diamond on the end of the line next to the class that will contain the data field. This represents the *has-a* relationship. If the diamond is open, this is called an *aggregation,* and if the diamond is filled, this is called a *composition*. The difference is that in a composition, the component objects are not considered to have an independent existence. For example, an Airplane is composed of two wings, a body, and a tail, none of which would exist unless it was a component of an Airplane. This would be modeled as shown in Figure B.9.

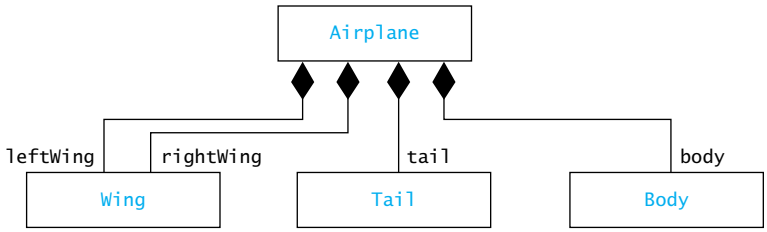


FIGURE B.9
Airplane Composed of Wing, Tail, and Body

However, a node in either a linked list or a tree has references to other nodes, but these other nodes are independent entities, and the value of the reference can be changed. Thus, we use the open diamond as shown in Figure B.10. Observe that the references prev and next are to the same class (Node).

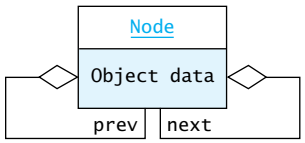


FIGURE B.10
A Node in a Double-Linked List

B.2 Object Diagrams

Objects are represented like classes using rectangles divided into two sections. The object's type and name in the top part and the attribute names and values in the bottom part. The object's type and name are underlined. The object's name is generally omitted. To show the value of references from one object to another, we use a small rectangle with a line to the referenced object.

Figure B.11 is an example of two object diagrams. It shows that the `Person` attribute `givenName` of the top one references a `String` object with the value of `"Adam"`.

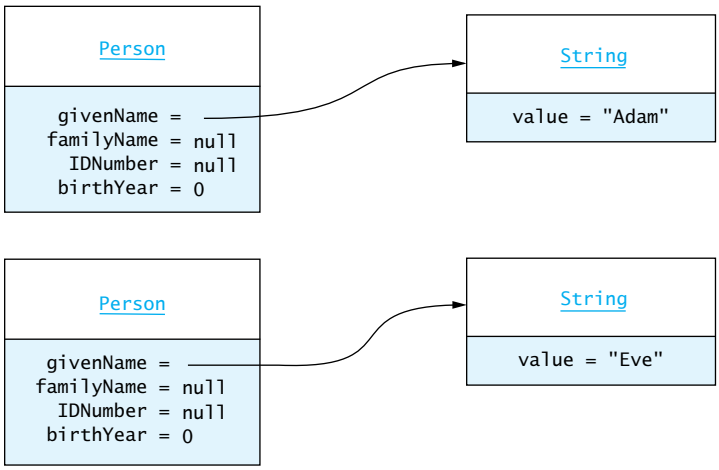


FIGURE B.11
Object Diagram of a `Person` Object

Glossary

2-3 tree A search tree in which each node may have two or three children.

2-3-4 tree A search tree in which each node may have two, three, or four children.

2-node A node in a 2-3 or 2-3-4 tree with two children.

3-node A node in a 2-3 or 2-3-4 tree with three children.

A* Algorithm A heuristic algorithm that finds the shortest path from a source vertex in a graph to a destination vertex.

abstract class A class that contains at least one abstract method.

abstract data type An implementation-independent specification of a set of data items and the operations performed on those data items.

abstraction A model of a physical entity or activity.

abstract method The specification of the signature of a method without its implementation. Abstract methods are declared in interfaces and abstract classes. A concrete class that is a subclass of an abstract class or an implementation of an interface must implement each abstract method declared in the abstract superclass or interface.

accelerator A key combination that invokes a menu item's action listeners without requiring the user to navigate the menu hierarchy.

acceptance testing A sequence of tests that demonstrates to the customer that a software product meets all of its requirements. Acceptance testing generally is observed by a customer representative.

action event An event caused by a user's action, such as pressing a key or clicking a GUI button.

action listener An object that contains a method that responds to an action event. An action listener is an event listener for an action event. See also *event listener*.

activation bar The thick line along the lifeline in a sequence diagram that indicates the time that a method is executing in response to the receipt of a message.

activation frame An area of memory allocated to store the actual parameters and local variables for a particular call to a method. In Java, references to activation frames are stored on the run-time stack. When a method is called, a new activation frame is pushed onto the stack, and when a method exits, the activation frame is popped.

actor An entity that is external to a given software system. In many cases an actor is a human user of the software system, but an actor may be another system.

adapter class A class that provides the same or very similar functionality as another class but with different method signatures. The actual work is performed by delegation to the methods in the other class.

address A number that represents an object's location in memory.

adjacency lists A representation of a graph in which the vertices (the destinations) adjacent to a given vertex (the source) are stored in a list associated with that vertex. The actual edge (source, destination, weight) from the source vertex to the destination may be stored.

adjacency matrix A representation of a graph in which the presence or absence of an edge is indicated by a value in a matrix that is indexed by the two vertices. The value stored is 0 for no edge, 1 for an edge in an unweighted graph, and the weight itself for a weighted graph.

adjacent [vertex] In a directed graph, a vertex, v, is adjacent to another vertex, u, if there is an edge, (u, v), from vertex u to vertex v. In an undirected graph, v is adjacent to u if there is an edge, $\{u, v\}$, between them.

aggregation An association between two classes in which one class is composed of a collection of objects of the other class.

analysis In the waterfall model, the phase of the software life cycle (workflow in the Unified Model) during which the requirements are clarified and the overall architecture of the solution is determined.

ancestor A node in a tree that is at a higher level than a given node and from which there is a path to that node (the descendant).

ancestor–descendant relationship A generalization of the parent–child relationship. (See *ancestor* and *descendant*.)

anchor The symbol ⊕ that is used in a UML class diagram to indicate that a class is an inner class of another class.

anonymous object An object for which there is no named reference. The Java **new** operator returns a reference to an anonymous object.

anonymous reference A reference to an object that itself has no name. Anonymous references are the result of a cast operation.

applet A top-level Java GUI class that is intended to be displayed in a frame that is under the control of a web browser.

assertion A statement about the current value of one or more variables.

association A relationship between two classes.

attributes The set of data values that determine the state of an object. Generally the attributes of a class are represented by data fields within the class.

auto-boxing A Java feature that performs automatic conversion between the primitive types and their corresponding wrapper classes.

AVL tree A self-balancing binary search tree in which the difference between the heights of the subtrees is stored in each tree node. The insertion and removal algorithms use rotations to maintain this difference within the range –1 to +1.

back edges An edge that is discovered during a depth-first search that leads to an ancestor in the depth-first search tree.

backtracking An approach to implementing a systematic trial-and-error search for a solution. When a dead end is reached, the algorithm follows a path back to the decision point that leads to the dead end, then moves forward along a different path.

balanced binary search tree A binary search tree in which the height of each pair of subtrees is approximately the same.

base case The case in a recursive algorithm that can be solved directly.

batch processing A way of using a computer in which a series of jobs (individual programs) are collected together and then executed sequentially.

big-O notation The specification of a set of functions that represent the upper bound of a given function. Formally the function $f(n)$ is said to be $\mathbf{O}(g(n))$ if there are constants $c > 0$ and $n_0 > 0$ such that for all $n > n_0$, $cg(n) \geq f(n)$.

binary search The process of searching a sorted sequence that begins by examining the middle element. If the middle element is greater than the target, then the search is applied recursively to the lower half; if it is less than the target, the search is applied recursively to the upper half.

binary search tree A binary tree in which the items in the left subtree of a node are all less than that node, and the items in the right subtree are all greater than that node.

binary tree A tree in which each node has 0, 1, or 2 children. The children are distinguished by the names left and right. If a node has one child, that child is distinguished as being a left child or a right child.

black-box testing A testing approach in which the internal structure of the item being tested is not known or taken into account in the design of the test cases. The test cases are based only on the functional requirements for the item being tested.

block A compound statement that may contain local variables and class declarations.

bottom-up design A design process in which the lower-level methods are designed first. A lowest-level method is one that does not depend on other methods to perform its function.

boundary condition A value of a variable that causes a different path to be taken. For example, in the statement `if (x > C) {...} else {...}`, the value of `C` is a boundary condition.

branch In a tree, the link between a parent node and one of its children.

branch coverage A measure of testing thoroughness. Each alternative from a decision point (**`if`**, **`switch`**, or **`while`** statement) is considered a branch. If a test exercises a branch, then that branch is considered covered. The ratio of the covered branches to the total number of branches is the branch coverage. See also *path coverage* and *statement coverage*.

breadth-first search A way of searching through a graph in which the vertices adjacent to a given vertex are all examined and placed into a queue. Once all the adjacent vertices are examined, the next vertex is removed from the queue. Thus vertices are examined in increasing distance (as measured by the number of edges) from the starting vertex.

breadth-first traversal See *breadth-first search*.

breakpoint A point in a program at which the debugger is instructed to suspend execution when it is reached. This allows for examination of the value of variables at a given point before execution is resumed.

B-tree A balanced search tree in which each node is a leaf or may have up to n children and $n - 1$ data items. The leaves are all at the bottom level. Each node (except for the root) is kept at least half full. That is, each node has between $(n - 1)/2$ and $n - 1$ data items. The root is either a single node (leaf) or it has at least one data item and two children.

bubble sort A sort algorithm that makes several passes through the sequence being sorted. During each pass, adjacent items are examined and, if out of order, swapped. If there are no exchanges on a given pass, then the process is complete. The effect of each pass is that the largest item in the unsorted part of the sequence is moved to (bubbles to) the end of the sequence.

bucket The list of keys stored in a hash table entry that uses chaining. All the keys in the list map to the index of that table entry.

bucket hashing See *chaining*.

byte code The platform-independent representation of a Java program that is the output of the Java compiler and is the input to the Java Virtual Machine (JVM). The JVM then interprets this input to execute the program.

casting When applied to a reference to an object, casting reinterprets that reference to refer to an object of a different type. The object must be of the target type (or a subclass of the target type) for the cast to be valid. When applied to a primitive numeric value, a cast represents a conversion to an equivalent value of the target primitive numeric type.

`catch` block The sequence of statements that will be executed when an exception is caught by a **`catch`** clause.

`catch` clause The specification of an exception type and the statements to be executed when an exception of that type is caught. One or more **`catch`** clauses follow a **`try`** block and will catch the exceptions thrown from that **`try`** block.

chaining An approach to hashing in which all keys that are mapped to a given entry in the hash table are placed into a list. The list is called a *bucket*.

check box A GUI component that may be either selected or deselected. The component is generally shown as a small square box, and the selected state is indicated by a check-mark symbol. Several check boxes may be presented to give the user the choice of one or more from a group of possible values See also *combo box* and *radio button*.

checked exception An exception that either must be declared in a **`throws`** declaration or caught by a **`try-catch`** sequence.

child A node in a tree that is the immediate descendant of another node.

class The fundamental programming unit in a Java program. A class consists of a collection of zero or more data fields (instance variables) and zero or more methods that operate on those data fields.

class diagram A UML diagram that shows a number of classes and the relationships between them.

class method See *static method*.

client A class or method that uses a given class.

cloning The process of making a *deep copy* of an object.

closed-box testing See *black-box testing*.

collection hierarchy The hierarchy of classes in the Java API that consists of classes designed to represent collections of other object.

collision The mapping of two or more keys into the same position in a hash table.

combo box A GUI component that is a combination of a button or editable field and a drop-down list. The drop-down list is a set (or menu) of choices, only one of which may be selected at a time. The current selection is displayed in the button of field, but when selected by a mouse click, the drop-down list is displayed, allowing the selection of one of the other choices. See also *check box* and *radio button*.

complete binary tree A binary tree in which each node is a leaf or has two children.

component In a GUI application, an object displayed on the screen that can interact with the user.

component class A class whose objects are part of another object. See *composition*.

component testing The testing of an individual part of a program by itself. In a Java program, a component may be a method or a class.

composition The association between two classes in which objects of one class are part of another class. The parts generally do not have an independent existence but are created when the parent object is created. For example, an `Airplane` object is composed of a `Body` object, two `Wing` objects, and a `Tail` object.

compound statement Zero or more statements enclosed within braces {. . .}.

concrete class (actual class) A class for which objects can be instantiated.

connected components A set of vertices within a graph for which there is a path between every pair of vertices.

connected graph A graph that consists of a single connected component.

construction phase The phase of the Unified Model of the software life cycle during which most of the activity is devoted to writing the software.

constructor A method that initializes an object when it is first created.

container In a GUI application, a component that contains other components.

content pane In a Swing GUI application, the component of a frame in which the application places the components to be displayed.

contract The specification of the pre- and postconditions of a method.

cost of a spanning tree The sum of the weights of the edges.

coverage testing See *branch coverage*.

cycle A path in a graph in which the first and final vertices are the same.

data abstraction The specification of the data items of a problem and the operations to be performed on these data items that does not specify how the data items will be represented and stored in memory. See also *abstract data type*.

data field (instance variable) A variable that is part of a class.

debugging The process of finding and removing defects (bugs) from a program.

deep copy A copy of an object in which data field values and references to immutable objects are simply duplicated, but each reference to a mutable object references a copy of that object. If there are mutuable references in any object that is copied, these also reference a copy of that object. The effect is that you can change any value in a deep copy of an object without modifying the original object.

default constructor The no-parameter constructor that is generated by the Java compiler if no constructors are defined.

default visibility The same as *package visibility*.

default method A method that is defined in an interface.

defensive programming An approach to designing a program that builds in statements to test the values of variables that might result in an exception or runtime error (to be sure that they are valid) before statements that use the variables are executed.

delegation The implementation of a method in one class that merely calls a method in another class.

delimiter characters Characters that are defined to separate a string into tokens.

depth (level) The number of nodes in a path from the root to a node.

depth-first search A method of searching a graph in which adjacent vertices are examined along a path until a dead end is reached. The search then backtracks until an unexamined vertex is found, and the search continues with that vertex.

depth-first traversal See *depth-first search*.

deque A data structure that combines the features of a stack and queue. Items may be inserted in one end and removed from either.

descendant In a tree, a lower node that can be reached by following a path from a given node.

design The process by which classes and methods are identified and defined to create a program that satisfies a given set of requirements.

detail message An optional string to be displayed when an exception is thrown that provides additional information about the conditions that led to the exception.

dialog In a GUI application, a window that provides information or asks for data entry.

digraph See *directed graph*.

Dijkstra's algorithm An algorithm that uses breadth-first search to find the shortest path in a graph from a vertex to all other vertices.

directed acyclic graph A directed graph that contains no cycles.

directed edge An edge in a directed graph.

directed graph A graph in which every edge is considered to have a direction. If u and v are vertices in a graph, then the presence of the edge (u, v) indicates that v is adjacent to u,

but *u* may not be adjacent to *v*. Contrast with *undirected graph*.

discovery order The order in which vertices are discovered in a depth-first search.

double buffering In a GUI application, updates to the image are made in one area of memory while the image being displayed is based on another area of memory. When all of the updates are complete, the memory areas are swapped and the new image is displayed.

downcast A reinterpretation of a reference from a superclass to a subclass. In Java, downcasts are tested for validity. See also *casting*.

driver A method whose purpose is to call a method being tested and provide it with appropriate argument values. Usually the result of executing the method is displayed immediately to the user.

edges In a graph, the links between pairs of vertices.

elaboration phase The phase in the Unified Model of the software life cycle during which the software architecture is defined.

escape sequence A sequence of characters beginning with the backslash (\), which is used to indicate another character that cannot be directly entered. For example, the sequence `\n` represents the newline character.

Euler tour A path around a tree, starting and ending with the root. The tree is always kept to the left of the path when viewed from the direction of travel along the path.

event The occurrence of an external input or an internal state change.

event listener An object that is registered to respond to an event. The object's class contains a method that is called when the event occurs.

exclusive or (XOR) A graphics drawing mode in which drawing a shape twice has the effect of erasing the original shape from the image.

extending The process of adding functionality by defining a new class that adds data fields and/or adds or overrides methods of an existing class.

external node See *leaf*.

Extreme Programming A software development process in which programmers work in pairs. One programmer writes methods, while the other designs tests for those methods. The programmers alternate roles. The programmers also share a workstation so that when one programmer is using the workstation, the other is observing.

factory method A method that is responsible for creating objects of a class. Generally a factory method will be associated with an abstract class or interface and will choose an appropriate concrete class that extends the abstract class or implements the interface based on parameters passed to the factory method and/or system parameters. Returns a reference to a new object of this concrete class.

`finally` block A block preceded by the key word **`finally`**. Part of the **`try-catch-finally`** sequence.

finish order The order in which the vertices are finished in a depth-first search. A vertex is considered finished when all of the paths to adjacent vertices have been finished.

firing an event The process of indicating that an event has occurred.

forest A collection of trees that may result from a depthfirst search of a directed graph or an unconnected graph.

frame A top-level container in a GUI application. A frame consists of a window with a border around it.

full binary tree A binary tree in which the nodes at all but the deepest level contain two children. At the deepest level, all nodes that have two children are to the left of those that have no children, and there is at most one node with a left child that is between these two groups.

functional testing Testing that concentrates on verifying that software meets its functional requirements.

garbage collector The process of reclaiming memory that no longer has a reference to it. This process generally runs in the background.

generalization The relationship between two classes in which one class is the superclass and the other is a subclass. The superclass is a generalization of the subclass.

generic class A class with type parameters that are specified when instances are created. These parameters specify the actual data type for the internal data fields of the object that is created.

generic method A method with type parameters that are used to represent the data type of its formal parameters. The type parameters are specified when the method is called and enable the method to process actual parameters of different data types.

generic type A type that is defined in terms of another type where that other type may be specified as a parameter. For example, the class `List<E>` is a `List` designed to hold objects of type `E`, where `E` may be any other class and is specified when the object is created.

glass-box testing Testing that takes the internal structure of the unit being tested into account.

graph A mathematical structure consisting of a set of vertices and edges. The edges represent a relationship between the vertices.

hash code A function that transforms an object into an integer value that may be used as an index into a hash table.

heapsort A sort algorithm in which the items being sorted are inserted into a heap, then removed one at a time.

height of a tree The number of nodes in a path from the root to the deepest leaf.

heuristic A problem-solving method that uses approximations to actual results to produce more accurate results.

Huffman code A varying-length binary code in which each symbol is assigned a code whose length is inversely proportional to the frequency with which that symbol appears (or is expected to appear) in a message. The resulting coded message is the minimum possible length.

image rendering The process of creating an image in a device-dependent form for display on that device. During this process, the values of individual pixels are determined.

immutable A class that is immutable has no methods to change the value of its data fields. An immutable object can't be changed.

implement (an interface) To provide in a class an implementation of all of the methods specified by an interface.

inception phase In the Unified Model of the software life cycle, the initial phase of a project in which the requirements are first identified.

increment operator The operator that has the side effect of adding 1 to its operand.

index A value that specifies a position within an array.

infix notation Mathematical notation in which the operators are between the operands.

information hiding The design principle that states that the internal data representations of a class cannot be used or directly modified by clients.

inherit To receive from an ancestor. In an object-oriented language, a subclass inherits the visible methods and data fields from its superclass. These inherited methods and data fields appear to clients of the subclass as if they were members of that class.

initializer list A list of values, enclosed in braces, that initializes the values in an array.

inner class A class that is defined within another class. Methods of inner classes have access to the data fields and methods of the outer class in which they are defined and vice versa.

inorder predecessor For a binary search tree, the inorder predecessor of an item is the largest item that is less than this item. The node containing an item's inorder predecessor would be visited just prior to that item in an inorder traversal.

insertion sort A sorting algorithm in which each item is inserted into its proper place in the sorted region.

instance See *object*.

instance method A method that is associated with an object. Contrast with *static method*.

`instanceof` operator The Java operator that returns **`true`** if a reference variable references an instance of a specified class or interface.

instance variables A variable of a class that is associated with an object (i.e., a data field of an object). Contrast with *static variable*.

integration testing Testing in which the interaction of the components or units of a software program is validated.

interface The external view of a class. In Java, an interface is a class that defines nothing more than public abstract methods and constants.

internal node A node in a tree that has one or more children. Contrast with *leaf*.

interpret To translate or understand the meaning of. The Java Virtual Machine interprets the machine-independent byte code in terms of specific machine-language instructions for the computer on which it is executing.

iteration In a loop, a complete execution of the loop body. In the Unified Model of the software life cycle, a sequence of activities that results in the release of a set of software artifacts.

iterator An object that accesses the objects contained in a collection one at a time.

Javadoc The commenting convention defined for Java programs. Also, the program that generates documentation from the comments that follow this convention in a program.

key A value or reference that is unique to a particular object and thereby identifies that object (e.g., a social security number).

Last In, First Out (LIFO) An organization of data such that the most recently inserted item is the one that is removed first.

last-line recursion A recursive algorithm or method in which the recursive call is the last executable statement.

layout manager An object in a GUI application that manages the visual arrangement of components in a container.

leaf (node) A node in a tree that has no children. Contrast with *internal node*.

left rotation The transformation of a binary search tree in which the right child of the current root becomes the new root and the old root becomes the left child of the new root.

level of a node The number of nodes in a path from the root to this node.

life line The dotted vertical line in a UML sequence diagram that indicates the lifetime of an object.

linear probing A collision resolution method in which sequential locations in a hash table are searched to find the item sought or an empty location.

linear search A search algorithm in which items in a sequence are examined sequentially.

link A reference from one node to another.

literal A constant value that appears directly in a statement.

logic error An error in the design of an algorithm or program. Contrast with *syntax error*.

logical view A description of the data stored in an object that does not specify the physical layout of the data in memory.

loop invariant An assertion that is true before each execution of the loop body and is true when the loop exits.

many-to-one mapping An association among items in which more than one item (the values) are associated with a single item (a key).

marker An interface that is defined with no methods or constants. It is used to give a common name to a family of interfaces or classes.

merge The process of combining two sorted sequences into a single sorted sequence.

merge sort A sorting algorithm in which sorted subsequences are merged to form larger sorted sequences.

message In an object-oriented design, a message represents an occurrence of a method call.

message to self A message that is passed from an object to itself. It represents a method calling another method within the same class.

method A sequence of statements that can be invoked (or called) passing a fixed number of values as arguments and optionally returning a value.

method declaration The specification of the name, parameters, and return type of a method. See also *signature*.

method overloading The presence of multiple methods in a class with the same name but different signatures.
method overriding The replacement of an inherited method with a different implementation in a subclass.
minimum spanning tree A subset of the edges of a connected graph such that the graph remains connected and the sum of the weights of the edges is the minimum.
mnemonic A character that can be used to select a menu item from the keyboard when the menu is displayed.
multiple inheritance Inheriting from more than one superclass.
multiplicity An indication of the number of objects in an association.

narrowing conversion A conversion from a type that has a larger range of values to a type that has a smaller one.
nested class See *inner class*.
network A system consisting of interconnected entities.
newline The special character that indicates the end of a line of input or output.
new operator The Java operator that creates objects (or instances) of a class.
node An object to store data in a linked list or tree. This object will also contain references to other nodes.

object An example or instance of a class. Internally, it is an area of memory that is structured as defined by a class. The methods of that class operate on the values defined within this memory area.
object-oriented design A design approach that identifies the entities, or objects, that participate in a problem or system and then designs classes to model these objects within a program.
onto mapping A mapping in which each value in the value set is mapped to by at least one member of the key set.
open-box testing See *glass-box testing*.
operations The methods defined in a class.
operator For classes, operator is another name for *method*. For primitive types, it represents a predefined function on one or two values (e.g., addition).
output buffer A memory area in which information written to an output stream is stored prior to being written to disk.
override Replace a method inherited from a superclass by one defined in a subclass.

package A grouping of classes under a common package name.
package visibility A level of visibility whereby variables and methods are visible to methods defined in classes within the same package.
panel A general-purpose GUI component that can be used as a drawing surface or to contain other GUI components.
parent The node that is directly above a node within a tree.
partitioning The process of separating a sequence into two sequences; used in quicksort.
path In a graph, a sequence of vertices in which each vertex is adjacent to its predecessor.
path coverage A measure of testing thoroughness. If a test exercises a path, then that path is considered covered. The ratio of the covered paths to the total number of paths is the path coverage. See also *branch coverage* and *statement coverage*.
phase In the Unified Model of the software life cycle, the span of time between two major milestones.
physical view A view of an object that considers its actual representation in computer memory.
pivot In the quicksort algorithm, a value in the sequence being sorted that is used to partition the sequence. The sequence is partitioned into values that are less than or equal to the pivot and values that are greater than the pivot.
polymorphism Many forms or many shapes. In a Java program, a method defined in a superclass (or interface) may be called through a reference to that superclass (or interface). The actual method executed is the one that overrides that method and is defined in the concrete subclass object that is referenced by the superclass (or interface) variable.
pop Remove the top element of a stack.
postcondition An assertion that will be true after a method is executed, assuming that the preconditions were true before the method is executed.
postfix increment The increment operator (e.g., `i++`) that has the side effect of incrementing the variable to which it is applied, but its current value is the value of the variable before the increment takes place (e.g., `i`).
postfix notation A mathematical notation in which the operators appear after their operands.
precedence The degree of binding of infix operators. Operators of higher precedence are evaluated before operators of lower precedence.
precondition An assertion that must be true before a method is executed for the method to perform as specified.
prefix increment The increment operator (e.g., `++i`) that has the side effect of incrementing the variable to which it is applied, and its current value is the value of the variable after the increment takes place (e.g., `i + 1`).
private visibility A level of visibility whereby variables and methods are visible only to methods defined in the same class.
procedural abstraction The philosophy that procedure (method) development should separate the concern of *what* is to be achieved by a procedure (or method) from the details of *how* it is to be achieved.
proof by induction A proof method which demonstrates that a proposition is true for a base case (usually 0) and then demonstrates that if the proposition is true for an arbitrary value (k), it is then true for the successor of that value ($k + 1$).
protected visibility A level of visibility whereby variables and methods are visible to methods defined in the same class, subclasses of that class, or the same package.
pseudocode A description of an algorithm that is structured similar to a programming language implementation but lacks the formal syntax and notation of a programming language. Generally pseudocode will use common programming language decision and looping constructs.
pseudorandom A computer-generated sequence of values that appear to be random because they pass various statistical tests that are consistent with those that would be produced by a truly random sequence.

public visibility A level of visibility whereby variables and methods are visible to all methods regardless of which class or package they are defined in.

quadratic probing In a hash table, a collision resolution technique in which the sequence of locations that are examined increases as the square of the number of probes made.

queuing theory The branch of mathematics developed to solve problems associated with queues by developing mathematical models for these problems.

quicksort A sorting algorithm in which a sequence is partitioned into two subsequences, one that is less than or equal to a pivot value and the other that is greater than the pivot value. The process is then recursively applied to the subsequences until a subsequence with one item is reached.

radio button A GUI component that may be either selected or deselected. The component is generally shown as a small open circle, and the selected state is indicated by a filled-in circle. Radio buttons are grouped into a button group so that only one item in the group may be selected at a time. See also *check box* and *combo box*.

random access The ability to access any object in a collection by means of an index.

recursive case A case in a recursive algorithm that is solved by applying the algorithm to a transformed version of its parameter.

recursive data structure A data structure that is defined in terms of itself.

recursive method A method that calls itself.

Red-Black tree A self-balancing binary search tree that maintains balance by distinguishing the nodes by one of two states: "red" or "black." Algorithms for insertion and deletion maintain balance by ensuring that the number of black nodes in any path from the root to a leaf is the same.

reference variable A variable that references an object.

registering (listener for event) The process by which a listener object is associated with an event. This is done by calling a method defined by the component that recognizes the event and passing a reference to the listener object.

regression testing Testing that ensures that changes to the item being tested do not invalidate previously verified functions.

rehashing The process of moving the items in one hash table to a larger hash table using hashing to find each item's new location.

request-response A program that issues a request to the user and then waits for input.

requirements Specifying what a program or system is to do without specifying how it is done.

reusable code Code written for one program that can be used in another.

right rotation The transformation of a binary search tree in which the left child of the current root becomes the new root and the old root becomes the right child of the new root.

root The node in a tree that has no parent and is at the top level.

rubber banding Continuously erasing and redrawing a shape so that it follows the mouse position.

run-time error An error that is detected when the program executes. In Java, run-time errors are detected by the Java Virtual Machine.

starter method See *wrapper method*.

seed The initial value in a pseudorandom number sequence. Changing the seed causes a different sequence to be generated by the pseudorandom number generator.

selection sort A sort algorithm in which the smallest item is selected from the unsorted portion of the sequence and placed into the next position in the sorted portion.

self-balancing search tree A search tree with insertion and removal algorithms that maintain the tree in balance. See *2-3 tree*, *2-3-4 tree*, *AVL tree, balanced binary search tree,* and *Red-Black tree*.

sequence diagram A UML diagram that shows the sequence of messages between objects that are required to perform a given function or realize a use case.

set difference For sets A and B, A–B is the subset of a set, A, that does not contain elements of some other set, B.

set intersection A set of the elements that are common to two sets.

set union A set of the elements that are in one set or the other.

shallow copy A copy of an object that copies only the values of the data fields. If a data field is a reference, the original and the copy reference the same target object.

Shell sort A variation on insertion sort in which elements separated by a value known as the gap are sorted using the insertion sort algorithm. This process repeats using a decreasing sequence of values for the gap.

sibling One of two or more nodes in a tree that have a common parent.

signature A method's name and the types of its parameters. The return type is not part of the signature because it is illegal to have two methods with the same signature and different return types.

simple path A path that contains no cycles.

simulation The process of modeling a physical system using a computer program.

single-step execution In debugging, the process of executing one statement at a time so that the user may examine the values of variables after each statement is executed.

skip-list A randomized variant of an ordered linked list with additional parallel lists. Parallel lists at higher levels skip geometrically more items. Searching begins at the highest level to quickly get to the correct part of the list, then uses progressively lower level lists. A new item is added by randomly selecting a level, then inserting it in order in the lists for that and all lower levels. With enough levels, searching is $\mathbf{O}(\log n)$.

software life cycle The sequence of phases that a software product goes through as it is developed.

spanning tree A minimum subset of the vertices of a connected graph that still results in a connected graph.

stack trace A listing of the sequence of method calls that starts where an error is detected and ends at the program invocation.

state The current value of all of the data fields in an object.

statement coverage A measure of testing thoroughness. If a test exercises a statement, then the statement is considered

covered. The ratio of the covered statements to the total number of statements is the statement coverage. See also *branch coverage* and *path coverage*.

static method A method defined within a class but not associated with any particular object of that class.

static variable A variable defined in a class that is not a member of any particular object but is shared by all objects of the class.

step into When debugging in single-step mode, setting the next statement to be executed to be the first statement of the method. Each individual statement in the method is executed in sequence.

step over When debugging in single-step mode, setting the method call to be treated as a single statement.

stepwise refinement The process of breaking a complicated problem into simpler problems. This process is repeated with the smaller problems until a problem of solvable size is reached.

strongly typed language A programming language in which the type of objects is verified when arguments are bound to parameters and when values are assigned to variables. A syntax error occurs if the types are not compatible.

structure chart A diagram that represents the relationship between problems and their subproblems.

structured walkthrough A design or code review following a defined process in which the author of a program leads the review team through the design and implementation, and the reviewers follow a checklist of common defects to verify that these defects are not present.

stub A dummy method that is used to test another method. A stub takes the place of a method that the method being tested calls. A stub typically will return a known result.

subclass A class that is an extension of another class. A subclass inherits the members of its superclass.

subset A set that contains only elements that are in some other set. A subset may contain any or all of the elements of the other set, or it may be the empty set.

subtree of a node The tree that consists of this node as its root.

superclass A class that has a subclass. See *subclass*.

syntax error An error that violates the syntax rules of the language. Syntax errors are generally the result of a mistake in entering the program into the computer (typographical error) or a misunderstanding of the language syntax. Syntax errors are detected by the compiler.

system analyst A person who analyzes a problem to determine the requirements for a software program.

system testing Testing of a complete program or solution to a problem.

tail recursion See *last-line recursion*.

test case An individual test.

test framework A set of classes and procedures used to design and conduct tests.

test harness A method that executes the individual test cases of a test suite and records the results.

test suite A collection of test cases.

throw an exception Indicate that the situation that causes an exception has been detected.

Timsort A modification of the merge sort algorithm that takes advantage of sorted subsets in the data being sorted. Used as the library sort algorithm in Python and in Java for sorting lists of objects.

token A character or string extracted from a larger string. Tokens are separated by *delimiter characters*.

top-down design A design process that represents the solution to a higher module in terms of the solution to one or more lower-level modules.

topological sort An ordering of a sequence of items for which a partial order is defined that does not violate the partial order. For example, if a is defined to be before b (a is a prerequisite of b) by the partial order, then a will not appear later in the sequence than b. A partial order is defined by a directed acyclic graph.

transition phase In the Unified Model of the software life cycle, the phase in which the software product is turned over to the end users.

tree traversal The process of systematically visiting each node in a tree.

`try` block A block preceded by the reserved word **`try`**. Part of the **`try-catch-finally`** sequence.

`try-catch-finally` sequence A sequence consisting of a `try` block followed by one or more **`catch`** clauses and optionally followed by a **`finally`** block. Or a **`try`** block followed by a **`finally`** block. Exceptions that are thrown by the **`try`** block are handled by the **`catch`** clauses that follow it. Statements in the **`finally`** block are executed either after the **`try`** block exits normally or when a **`catch`** block that handles an exception exits.

`try`-with-resources A try block that begins with the declaration of a resource such as a data file that is to be opened and processed in the **`try`** block. The resource is automatically closed when exiting from the **`try`** block or any **`catch`** block that may be executed. This eliminates the need for the programmer to close the resource.

type cast The process of converting from one type to another.

unchecked exception An exception that does not have to be declared in a **`throws`** statement or have the statements that might throw it enclosed within a **`try`** block.

undirected edge An edge in an *undirected graph*.

undirected graph A graph in which no edge has a direction. If u and v are vertices in a graph, then the presence of the edge $\{u, v\}$ indicates that v is adjacent to u and u is adjacent to v. Contrast with *directed graph*.

Unified Model A software development life cycle model that is defined in terms of a sequence of phases and workflows. The workflows are exercised during each iteration of each phase, but the distribution of the amount of effort for each workflow varies from iteration to iteration.

Unified Modeling Language (UML) A language to describe the modeling of an object-oriented design that is the unification of several previous modeling systems. Specifically, the modeling techniques developed by Booch, Jacobson, and Rumbaugh were combined to form the initial version. UML has since evolved and is defined by a standard issued by the Object Modeling Group.

unit testing Testing of an individual unit of a software program. In Java, a unit is generally a method or class.
unnamed reference See *anonymous reference*.
unwinding the recursion The process of returning from a sequence of method calls and forming the result.
upcast Casting a reference to a superclass or interface type.
user interface (UI) The way in which the user and a program interact, or the class that provides this interaction.

version control The process of keeping track of the various changes that are made to a program as it is developed or maintained.
vertices The set of items that are part of a graph. The vertices are related to one another by edges.

waterfall model A software development model in which all of the activities of one workflow are completed before the next one is started.
weight A value associated with an edge in a weighted graph.
weighted graph A graph in which each edge is assigned a value.
widening conversion A conversion from a type that has a smaller set of values to one that has a larger set of values.
window A top-level container in a GUI application. Generally a window is a rectangular area on the display surface. See also *frame*.
wrapper class A class that encapsulates a primitive data type.
wrapper method A method whose only purpose is to call a recursive method, perhaps providing initial values for some parameters and returning the result. Also called a starter method.

Index

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.