APPENDIX A

Java Reserved Words and Keywords

These words have contextual meaning for the Java language and cannot be used as identifiers.

abstract	default	goto	package	synchronized
assert	do	if	private	this
boolean	double	implements	protected	throw
break	else	import	public	throws
byte	enum	instanceof	return	transient
case	extends	int	short	true
catch	false	interface	static	try
char	final	long	strictfp	void
class	finally	native	super	volatile
const	float	new	switch	while
continue	for	null		

The words *true*, *false*, and *null* are literals. The remainder of the words are Java keywords, although *const* and *goto* are not currently used in the Java language.

Java Reserved Words and Keywords

APPENDIX B

Operator Precedence

These rules of operator precedence are followed when expressions are evaluated. Operators in a higher level in the hierarchy—defined by their row position in the table—are evaluated before operators in a lower level. Thus, an expression in parentheses is evaluated before a shortcut postincrement is performed, and so on with the operators in each level. When two or more operators on the same level appear in an expression, the evaluation of the expression follows the corresponding rule for same-statement evaluation shown in the second column.

Operators	Order of Same-Statement Evaluation	Operation
( )	left to right	parentheses for explicit grouping
++ --	right to left	shortcut postincrement and postdecrement
++ -- !	right to left	shortcut preincrement and predecrement, logical unary NOT
* / %	left to right	multiplication, division, modulus
+ -	left to right	addition or *String* concatenation, subtraction
< <= > >= instanceof	left to right	relational operators: less than, less than or equal to, greater than, greater than or equal to; instanceof
== !=	left to right	equality operators: equal to and not equal to
&&	left to right	logical AND
||	left to right	logical OR
?:	left to right	conditional operator
= += -= *= /= %=	right to left	assignment operator and shortcut assignment operators

APPENDIX C

The Unicode Character Set

Java characters are encoded using the Unicode Character Set, which is designed to support international alphabets, punctuation, and mathematical and technical symbols. Each character is stored as 16 bits, so as many as 65,536 characters are supported.

The American Standard Code for Information Interchange (ASCII) character set is supported by the first 128 Unicode characters from 0000 to 007F, which are called the controls and Basic Latin characters, as shown on the next page.

Any character from the Unicode set can be specified as a *char* literal in a Java program by using the following syntax: '\uNNNN' where NNNN are the four hexadecimal digits that specify the Unicode encoding for the character.

For more information on the Unicode character set, visit the Unicode Consortium's website: *www.unicode.org*.

Controls and Basic Latin Characters

	000	001	002	003	004	005	006	007
0	NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	` 0060	p 0070
1	SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
2	STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
3	ETX 0003	DC3 0013	# 0023	3 0033	C 0043	S 0053	c 0063	s 0073
4	EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
5	ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
6	ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
7	BEL 0007	ETB 0017	' 0027	7 0037	G 0047	W 0057	g 0067	w 0077
8	BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
9	HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
A	LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
B	VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
C	FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
D	CR 000D	GS 001D	- 002D	= 003D	M 004D	] 005D	m 006D	} 007D
E	SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
F	SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

APPENDIX D

Representing Negative Integers

The industry standard method for representing negative integers is called **two's complement**. Here is how it works:

For an integer represented using 16 bits, the leftmost bit is reserved for the sign bit. If the sign bit is 0, then the integer is positive; if the sign bit is 1, then the integer is negative.

For example, let's consider two numbers, one positive and one negative.

0000 0101 0111 1001 is a positive integer, which we call *a*.

1111 1111 1101 1010 is a negative integer, which we will call *b*.

Using the methodology presented in Chapter 1 for converting a binary number to a decimal number, we can convert the binary number, *a*, to its decimal equivalent. Hence, the value of *a* is calculated as follows:

$$a = 2^{10} + 2^8 + 2^6 + 2^5 + 2^4 + 2^3 + 2^0$$
$$= 1{,}024 + 256 + 64 + 32 + 16 + 8 + 1$$
$$= 1{,}401$$

In contrast, *b*, the negative number, is represented in binary using the two's complement method. The leftmost bit, which is the sign bit, is a 1, indicating that *b* is negative. To calculate the value of a negative number, we first calculate its two's complement. The two's complement of any binary number is another binary number, which, when added to the original number, will yield a sum consisting of all 0s and a carry bit of 1 at the end.

To calculate the two's complement of a binary number, n, subtract n from 2^d, where d is the number of binary digits in n. The following formula summarizes that rule:

```
Two's complement of n = 2^d - n
```

Knowing that $2^d - 1$ is always a binary number containing all 1s, we can simplify our calculations by first subtracting 1 from 2^d, then adding a 1 at the end.

```
Two's complement of n = 2^d - 1 - n + 1
```

So to calculate the two's complement of b, which has 16 digits, we subtract b from a binary number consisting of 16 1s, then add 1, as shown here.

```
          2^d - 1   1111 1111 1111 1111
             - b    1111 1111 1101 1010
                    0000 0000 0010 0101
             + 1                      1
                    ---------------------
two's complement of b  0000 0000 0010 0110
```

Thus, the two's complement of b, which we will call c, is 0000 0000 0010 0110.

Another, simpler, way to calculate a two's complement is to invert each bit, then add 1. Inverting bits means to change all 0s to 1s and to change all 1s to 0s. Using this method, we get

```
        b   1111 1111 1101 1010

b inverted  0000 0000 0010 0101
      + 1                    1
        c   0000 0000 0010 0110
```

We can verify that the two's complement of b is correct by calculating the sum of b and c.

```
    b     1111 1111 1101 1010
    c     0000 0000 0010 0110
b + c   1 0000 0000 0000 0000
```

Converting c to decimal will give us the value of our original number b, which, as we remember, is negative. We have

$$
\begin{aligned}
b &= -(2^5 + 2^2 + 2^1) \\
&= -(32 + 4 + 2) \\
&= -38
\end{aligned}
$$

Because a leftmost bit of 0 indicates that the number is positive, using 16 bits, the largest positive number (we will call it *max*) that we can represent is

```
0111 1111 1111 1111
```

$max = (2^{14} + 2^{13} + 2^{12} + 2^{11} + 2^{10} + 2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0)$

This is equivalent to $2^{15} - 1$, which is 32,768 − 1, or 32,767.

Using 16 bits, then, the smallest negative number (we will call it *min*) that we can represent is

```
1000 0000 0000 0000
```

The two's complement of *min* is *min* itself. If we invert the bits and add 1, we get the same value we started with:

```
        min        1000 0000 0000 0000

  min inverted     0111 1111 1111 1111
          + 1      ____________________1
two's complement   1000 0000 0000 0000
```

and therefore *min* is -2^{15} or −32,768.

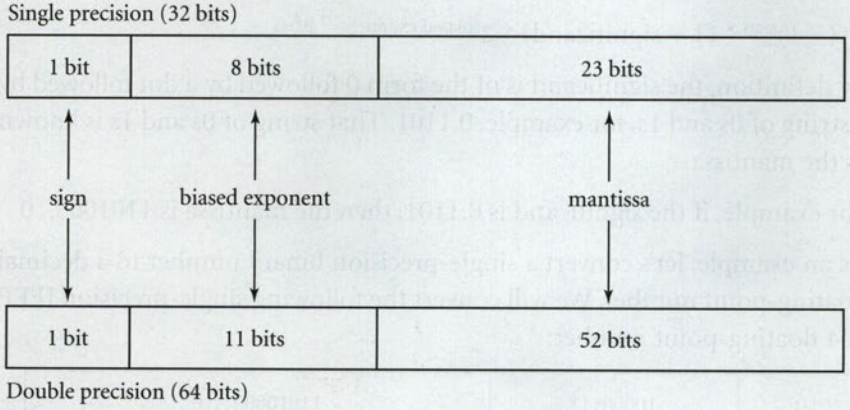
Thus, using 16 bits, we can represent integers between −32,768 and 32,767.

APPENDIX E

Representing Floating-Point Numbers

IEEE 754, a specification accepted worldwide and used by the Java language, defines how to represent floating-point numbers in binary numbers. Single-precision floating-point numbers use 32 bits of memory, and double-precision floating-point numbers use 64 bits.

Here is how single- and double-precision floating-point numbers are represented:



The leftmost bit stores the sign of the floating-point number; a 0 indicates a positive number, while a 1 indicates a negative number.

To represent the exponent of the number, which can be positive or negative, each representation stores a positive, biased exponent, calculated by adding a fixed bias, or scaling factor, to the real exponent of the number.

The purpose of the bias is to be able to represent both extremely large and extremely small numbers. The bias is equal to

$$2^{(\#\text{ of bits of the biased exponent} - 1)} - 1$$

Thus, for single precision, the bias is

$$2^{(8-1)} - 1 = 2^7 - 1 = 127$$

In single-precision, the 8-bit biased exponent can store 256 positive values (0 to 255). Thus, with a bias of 127, we can represent floating-point numbers with real exponents from −127 to 128, as shown here:

Real exponent	−127	−126	...	0	...	127	128
+ Bias	127	127	...	127	...	127	127
Biased exponent	0	1	...	127	...	254	255

Conversely, to find the real exponent from the biased exponent, we subtract the bias. For example, if the biased exponent is 150, then the real exponent is 150 − 127, which is 23. Similarly, if the biased exponent is 3, the actual exponent is 3 − 127, which is −124.

For double precision, the bias is

$$2^{(11-1)} - 1 = 2^{10} - 1 = 1023$$

A floating-point number is considered to be in the form

$$(-1)^{\text{sign}} * (1 + \text{significand}) * 2^{(\text{biased exponent} - \text{bias})}$$

By definition, the significand is of the form 0 followed by a dot followed by a string of 0s and 1s, for example, 0.1101. That string of 0s and 1s is known as the mantissa.

For example, if the significand is 0.1101, then the mantissa is 110100...0

As an example, let's convert a single-precision binary number to a decimal floating-point number. We will convert the following single-precision IEEE 754 floating-point number:

0	10000111	11010000...0

The leftmost digit, 0, tells us that the number is positive. The biased exponent is 10000111, which converted to decimal, is

$$= 2^7 + 2^2 + 2^1 + 2^0$$
$$= 128 + 4 + 2 + 1$$
$$= 135$$

The bias for single-precision floating-point numbers is 127, so the number is

$$= (-1)^0 * (1 + .1101) * 2^{(135 - 127)}$$
$$= 1.1101 * 2^8$$
$$= 1\ 1101\ 0000$$

In decimal, the number is

$$= 2^8 + 2^7 + 2^6 + 2^4$$
$$= 256 + 128 + 64 + 16$$
$$= 464$$

Given that .1 is $½^1$ or ½ in decimal, and .01 is $½^2$ or ¼, and .0001 is $½^4$ or 1/16 in decimal, we also could have calculated the number using this method:

$$= 1.1101 * 2^8$$
$$= (1 + 1 * ½^1 + 1 * ½^2 + 0 * ½^3 + 1 * ½^4) * 2^8$$
$$= (1 + ½ + ¼ + 1/16) * 2^8$$
$$= (1 + ½ + ¼ + 1/16) * 256$$
$$= 464$$

Now, let's convert a decimal floating-point number into single-precision, binary format. Here, we will convert the number −5.375, which we'll call *y*. First we convert the whole number portion (5) to binary, getting 101:

$$5 = 101$$

Then we convert the fractional part to binary:

$$.375 = .25 + .125$$
$$= ¼ + ⅛$$
$$= ½^2 + ½^3$$
$$= 0 * ½^1 + 1 * ½^2 + 1 * ½^3$$

Thus, .375 as represented in binary is .011.

Therefore, *y* can be represented in binary as

$$y = -\ 101.011$$
$$= -1.01011 * 2^2$$

We now can deduce the sign, the biased exponent, and the mantissa. The sign is 1 because the number is negative. The significand is 1.01011, and therefore the mantissa is 01011000…00. The exponent is 2, so the biased

exponent is 129 (2 plus the bias for single-precision numbers, which is 127):

$$\begin{aligned}\text{Biased exponent} &= 2 + 127 \\ &= 129\end{aligned}$$

Converting 129 to binary, we get

$$129 = 1000\ 0001$$

Therefore, the IEEE 754 single-precision value of the number y is

1	10000001	010110000...0

APPENDIX F

Java Classes APIs

In this appendix, we have compiled the APIs for the Java classes and interfaces used in this book. There are more methods and constructors for the classes presented here, and there are many more classes in the Java class library. We invite you to explore the Java APIs at *www.oracle.com/technetwork/java.*

ActionEvent

Package: java.awt.event

Description: contains information relating to the action event fired by a component. This event is passed to any *ActionListener* registered on the component.

A Useful Method of the *ActionEvent* Class

Return value	Method name and argument list
`Object`	`getSource( )`
	returns the object reference of the component that fired the event. This method is inherited from the *EventObject* class.

ActionListener Interface

Package: java.awt.event

Description: interface implemented by a class that will handle *ActionEvents* fired by a user interface component.

Interface Method to be Implemented

```
public void actionPerformed( ActionEvent event )
```

An event handler that implements the *ActionListener* interface provides code in this method to respond to the *ActionEvent* fired by any registered components.

ArrayList

Package: java.util

Description: implements a dynamically resizable array of object references.

Constructors

```
ArrayList<E>( )
```

constructs an *ArrayList* object of data type *E* with an initial capacity of 10.

```
ArrayList<E>( int initialCapacity )
```

constructs an *ArrayList* object of data type *E* with the specified initial capacity. Throws an *IllegalArgumentException*.

Useful Methods of the *ArrayList* Class (*E* represents the data type of the *ArrayList*.)

Return value	Method name and argument list
`boolean`	`add( E element )`
	appends the specified *element* to the end of the list. Returns *true*.
`void`	`clear( )`
	removes all the elements from the list.
`E`	`get( int index )`
	returns the element at the specified *index* position; the element is not removed from the list.

`E` `remove( int index )`

returns and removes the element at the specified *index* position.

`E` `set( int index, E newElement )`

returns the element at the specified *index* position and replaces that element with *newElement*.

`int` `size( )`

returns the number of elements in the list.

`void` `trimToSize( )`

sets the capacity to the list's current size.

BigDecimal

Package: java.math

Description: provides methods that perform addition, subtraction, multiplication, and division so that the results are exact, without the rounding errors caused by floating-point operations.

Constructor

`BigDecimal( String num )`

creates a *BigDecimal* object equivalent to the decimal number *num* expressed as a *String*. Throws a *NumberFormatException*.

Useful Methods of the *BigDecimal* Class

Return value	Method name and argument list
`BigDecimal`	`add( BigDecimal num )`

returns a *BigDecimal* object equal to the current *BigDecimal* object plus *num*.

`BigDecimal` `subtract( BigDecimal num )`

returns a *BigDecimal* object equal to the current *BigDecimal* object minus *num*.

`BigDecimal` `multiply( BigDecimal num )`

returns a *BigDecimal* object equal to the current *BigDecimal* object times *num*.

`BigDecimal divide( BigDecimal num )`

returns a *BigDecimal* object equal to the current *BigDecimal* object divided by *num*. Throws an *ArithmeticException*.

`int compareTo( BigDecimal num )`

returns 0 if the current *BigDecimal* object is equal to *num*; –1 if the current *BigDecimal* object is less than *num*; and 1 if the current *BigDecimal* object is greater than *num*.

BorderLayout

Package: java.awt

Description: a layout manager that arranges user interface components into five areas: NORTH, SOUTH, EAST, WEST, and CENTER. Each area can hold, at most, one component.

Constructors

`BorderLayout( )`

creates a border layout with no gaps between components.

`BorderLayout( int hGap, int vGap )`

creates a border layout with a horizontal gap of *hGap* pixels between components and a vertical gap of *vGap* pixels between components. Unlike *GridLayout*, horizontal gaps are not placed at the left and right edges, nor are vertical gaps placed at the top and bottom edges.

ButtonGroup

Package: javax.swing

Description: creates a mutually exclusive group of buttons.

Constructor

`ButtonGroup( )`

constructs a button group. Adding buttons to a *ButtonGroup* makes the buttons in the group mutually exclusive.

A Useful Method of the *ButtonGroup* Class

Return value	Method name and argument list
`void`	`add( AbstractButton button )`
	adds the *button* to the button group. The argument can be an object of the *JButton*, *JRadioButton*, or *JCheckBox* class because these are subclasses of the *AbstractButton* class.

Color

Package: java.awt

Description: creates colors to be used in producing graphical output.

Constructor

`Color( int red, int green, int blue )`

instantiates a *Color* object with the combined color intensities of *red*, *green*, and *blue*. Each color intensity can range from 0 to 255.

Predefined *Color* Constants

Color Constant	Red	Green	Blue
`Color.BLACK`	0	0	0
`Color.BLUE`	0	0	255
`Color.CYAN`	0	255	255
`Color.DARK_GRAY`	64	64	64
`Color.GRAY`	128	128	128
`Color.GREEN`	0	255	0
`Color.LIGHT_GRAY`	192	192	192
`Color.MAGENTA`	255	0	255
`Color.ORANGE`	255	200	0
`Color.PINK`	255	175	175
`Color.RED`	255	0	0
`Color.WHITE`	255	255	255
`Color.YELLOW`	255	255	0

Container

Package: java.awt

Description: a user interface component that can contain other components. *JComponent* is a subclass of *Container*; thus, all *JComponents* inherit these methods.

Useful Methods of the *Container* Class

Return value	Method name and argument list
`Component`	`add( Component component )` adds the *component* to the container, using the rules of the layout manager. Returns *component*.
`void`	`removeAll( )` removes all components from the container.
`void`	`setLayout( LayoutManager mgr )` sets the layout manager of the container to *mgr*.

DecimalFormat

Package: java.text

Description: provides methods for formatting numbers for output.

Constructor

`DecimalFormat( String pattern )`

instantiates a *DecimalFormat* object with the output *pattern* specified in the argument.

A Useful Method of the *DecimalFormat* Class

Return value	Method name and argument list
`String`	`format( double number )` returns a *String* representation of *number* formatted according to the *DecimalFormat* pattern used to instantiate the object. This method is inherited from the *NumberFormat* class.

Commonly Used Pattern Symbols for a *DecimalFormat* Object

Symbol	Meaning
0	Required digit. If the value for the digit in this position is 0, insert a zero.
#	Digit. Don't insert a character if the digit is 0.
.	Decimal point
,	Comma separator
%	Multiply by 100 and display a percentage sign

Double

Package: java.lang

Description: wrapper class that creates an equivalent object from a *double* variable and provides methods for converting a *String* to a *double* primitive type and a *Double* object.

Constructor

`Double( double d )`

instantiates a *Double* object with a *double* instance variable having the same value as *d*.

Useful Methods of the *Double* Wrapper Class

Return value	Method name and argument list
`double`	`parseDouble( String s )` *static* method that converts the *String s* to a *double* and returns that value. Throws a *NumberFormatException*.
`Double`	`valueOf( String s )` *static* method that converts the *String s* to a *Double* object and returns that object. Throws a *NumberFormatException*.

Enum

Package: java.lang

Description: provides for creation of enumerated types.

Useful Methods for *enum* Objects

Return value	Method name and argument list
`int`	`compareTo( Enum eObj )` compares two *enum* objects and returns a negative number if *this* object is less than the argument, a positive number if *this* object is greater than the argument, and 0 if the two objects are equal.
`boolean`	`equals( Object eObj )` returns *true* if *this* object is equal to the argument *eObj*; returns *false* otherwise.
`int`	`ordinal( )` returns the numeric value of the *enum* object. By default, the value of the first object in the list is 0, the value of the second object is 1, and so on.
`String`	`toString( )` returns the name of the *enum* constant.
`Enum`	`valueOf( String enumName )` *static* method that returns the *enum* object whose name is the same as the *String* argument *enumName*.

Exception

Package: java.lang

Description: the superclass for all predefined Java exceptions. All subclasses of the *Exception* class inherit these *public* methods.

Useful Methods of *Exception* Classes

Return value	Method name and argument list
`String`	`getMessage( )` returns a message indicating the cause of the exception. This method is inherited from the *Throwable* class.
`void`	`printStackTrace( )` prints the line number of the code that caused the exception, along with the sequence of method calls leading up to the exception.

`String toString( )`

returns a *String* containing the exception class name and a message indicating the cause of the exception.

File

Package: java.io

Description: represents platform-independent file names.

Constructor

`File( String pathname )`

constructs a *File* object with the *pathname* file name so that the file name is platform-independent.

FileOutputStream

Package: java.io

Description: writes bytes to a file.

Constructor

`FileOutputStream( String filename, boolean mode )`

constructs a *FileOutputStream* object from a *String* representing the name of a file; if mode is *false*, we will write to the file; if *mode* is *true*, we will append to the file. Throws a *FileNotFoundException*.

FlowLayout

Package: java.awt

Description: layout manager that arranges components left to right, starting a new row when a newly added component does not fit on the current row.

Constructor

`FlowLayout( )`

creates a flow layout with components centered.

Graphics

Package: java.awt

Description: represents the current graphical context, including the component on which drawing will take place and the current color.

Useful Methods of the *Graphics* Class

Return value	Method name and argument list
`void`	`clearRect( int x, int y, int width, int height )` draws a solid rectangle in the current background color with its top-left corner at (*x*, *y*), with the specified *width* and *height* in pixels.
`void`	`drawLine( int xStart, int yStart, int xEnd, int yEnd )` draws a line starting at (*xStart*, *yStart*) and ending at (*xEnd*, *yEnd*).
`void`	`drawOval( int x, int y, int width, int height )` draws the outline of an oval inside an invisible rectangle with the specified *width* and *height* in pixels. The top-left corner of the rectangle is (*x*, *y*).
`void`	`drawRect( int x, int y, int width, int height )` draws the outline of a rectangle with its top-left corner at (*x*, *y*), with the specified *width* and *height* in pixels.
`void`	`drawPolygon( Polygon p )` draws the outline of *Polygon p*.
`void`	`drawString( String s, int x, int y )` displays the *String s*. If you were to draw an invisible rectangle around the first letter of the *String*, (*x*, *y*) would be the lower-left corner of that rectangle.
`void`	`fillOval( int x, int y, int width, int height )` draws a solid oval inside an invisible rectangle with the specified *width* and *height* in pixels. The top-left corner of the rectangle is (*x*, *y*).
`void`	`fillRect( int x, int y, int width, int height )` draws a solid rectangle with its top-left corner at (*x*, *y*), with the specified *width* and *height* in pixels.

```
void        fillPolygon( Polygon p )
```

draws the *Polygon p* and fills its area with the current color.

```
void        setColor( Color c )
```

sets the current foreground color to the *Color* specified by *c*.

GridLayout

Package: java.awt

Description: a layout manager that arranges components in a grid with a fixed number of rows and columns.

Constructors

```
GridLayout( int numberOfRows, int numberOfColumns )
```

creates a grid layout with the number of rows and columns specified by the arguments.

```
GridLayout( int numberOfRows, int numberOfColumns,
            int hGap, int vGap )
```

creates a grid layout with the specified number of rows and columns and with a horizontal gap of *hGap* pixels between columns and a vertical gap of *vGap* pixels between rows. Horizontal gaps are also placed at the left and right edges, and vertical gaps are placed at the top and bottom edges.

Integer

Package: java.lang

Description: wrapper class that creates an equivalent object for an *int* variable and provides methods for converting a *String* to an *int* primitive type and an *Integer* object.

Constructor

```
Integer( int i )
```

instantiates an *Integer* object with an *int* instance variable having the same value as *i*.

Useful Methods of the *Integer* Wrapper Class

Return value	Method name and argument list
`int`	`parseInt( String s )`
	static method that converts the *String s* to an *int* and returns that value. Throws a *NumberFormatException*.
`Integer`	`valueOf( String s )`
	static method that converts the *String s* to an *Integer* object and returns that object. Throws a *NumberFormatException*.

ItemEvent

Package: java.awt.event

Description: contains information relating to the item event fired by a component. This event is passed to any *ItemListener* registered on the component.

Useful Methods of the *ItemEvent* Class

Return value	Method name and argument list
`Object`	`getSource( )`
	returns the object reference of the component that fired the event. This method is inherited from the *EventObject* class.
`int`	`getStateChange( )`
	If the item is selected, the value *SELECTED* is returned; if the item is deselected, the value *DESELECTED* is returned, where *SELECTED* and *DESELECTED* are *static int* constants of the *ItemEvent* class.

ItemListener Interface

Package: java.awt.event

Description: interface implemented by a class that will handle *ItemEvents* fired by a user interface component.

Interface Method to be Implemented

`public void itemStateChanged( ItemEvent event )`	
	An event handler that implements the *ItemListener* interface writes code in this method to respond to the *ItemEvent* fired by any registered components.

JButton

Package: javax.swing

Description: a command button user interface component. When a user presses the button, an *ActionEvent* is fired.

Constructor

`JButton( String buttonLabel )`

constructs a command button labeled *buttonLabel*.

A Useful Method of the *JButton* Class

Return value	Method name and argument list
`void`	`addActionListener( ActionListener handler )`

registers an event handler for *ActionEvents* on this button. This method is inherited from the *AbstractButton* class.

JCheckBox

Package: javax.swing

Description: a checkbox user interface component. When a user clicks the checkbox, its state alternates between selected and not selected. Each click fires an *ItemEvent*.

Constructors

`JCheckBox( String checkBoxLabel )`

constructs a checkbox labeled *checkBoxLabel*. By default, the checkbox is initially deselected.

`JCheckBox( String checkBoxLabel, boolean selected )`

constructs a checkbox labeled *checkBoxLabel*. If *selected* is *true*, the checkbox is initially selected; if *selected* is *false*, the checkbox is initially deselected.

Useful Methods of the *JCheckBox* Class

Return value	Method name and argument list
`void`	`addItemListener( ItemListener handler )`

registers an event handler for *ItemEvents* on this checkbox. This method is inherited from the *AbstractButton* class.

`boolean` `isSelected( )`

returns *true* if the checkbox is selected; *false* otherwise. This method is inherited from the *AbstractButton* class.

`void` `setSelected( boolean state )`

selects the checkbox if *state* is *true*; deselects the checkbox if *state* is *false*. This method is inherited from the *AbstractButton* class.

JComboBox

Package: javax.swing

Description: a drop-down list user interface component. When a user selects an item from the list, an *ItemEvent* is fired.

Constructor

`JComboBox( Object [ ] arrayName )`

constructs a new *JComboBox* component initially filled with the objects in *arrayName*. Often, the objects are *Strings*.

Useful Methods of the *JComboBox* Class

Return value	Method name and argument list
`void`	`addItemListener( ItemListener handler )`
	registers an event handler for *ItemEvents* on this combo box.
`int`	`getSelectedIndex( )`
	returns the index of the selected item. The index of the first item in the list is 0.
`void`	`setMaximumRowCount( int size )`
	sets the number of rows that will be visible at one time. If the list has more items than the maximum number visible at one time, scrollbars are added.
`void`	`setSelectedIndex( int index )`
	sets the item at *index* as selected. The index of the first item in the list is 0.

JComponent

Package: javax.swing

Description: the superclass for Swing user interface components, except for top-level components, such as *JFrame* and *JApplet*. Subclasses include *JButton*, *JCheckBox*, *JRadioButton*, *JList*, *JTextField*, *JTextArea*, *JPasswordField*, *JComboBox*, *JPanel*, and others, which inherit these *public* methods.

Useful Methods of the *JComponent* Class

Return value	Method name and argument list
`void`	`addMouseListener( MouseListener handler )` registers a *MouseListener* object on the component. This method is inherited from the *Component* class.
`void`	`addMouseMotionListener( MouseMotionListener handler )` registers a *MouseMotionListener* object on the component. This method is inherited from the *Component* class.
`void`	`repaint( )` automatically forces a call to the *paint* method. This method is inherited from the *Component* class.
`void`	`setBackground( Color backColor )` sets the background color of the component to *backColor*.
`void`	`setEnabled( boolean mode )` enables the component if *mode* is *true*, disables the component if *mode* is *false*. An enabled component can respond to user interaction.
`void`	`setForeground( Color foreColor )` sets the foreground color of the component to *foreColor*.
`void`	`setOpaque( boolean mode )` sets the component's background to opaque if *mode* is *true*; sets the component's background to transparent if *mode* is *false*. If opaque, the component's background is filled with the component's background color; if transparent, the component's background is filled with the background color of the container on which it is placed. The default is transparent.

`void` `setToolTipText( String toolTip )`

sets the tool tip text to *toolTip*. When the mouse lingers over the component, the tool tip text will be displayed.

`void` `setVisible( boolean mode )`

makes the component visible if *mode* is *true*; hides the component if mode is *false*. The default is visible.

JFrame

Package: javax.swing

Description: a window user interface component.

Constructors

`JFrame( )`

constructs a *JFrame* object, initially invisible, with no text in the title bar.

`JFrame( String titleBarText )`

constructs a *JFrame* object, initially invisible, with *titleBarText* displayed on the window's title bar.

Useful Methods of the *JFrame* Class

Return value	Method name and argument list
`Container`	`getContentPane( )`

returns the content pane object for this window.

`void` `setDefaultCloseOperation( int operation )`

sets the default operation when the user closes this window, that is, when the user clicks on the X icon in the top-right corner of the window.

`void` `setSize( int width, int height )`

sizes the window to the specified *width* and *height* in pixels. This method is inherited from the *Component* class.

`void` `setVisible( boolean mode )`

displays this window if *mode* is *true*; hides the window if *mode* is *false*. This method is inherited from the *Component* class.

JLabel

Package: javax.swing

Description: a user interface component that displays text or an image. A *JLabel* does not fire any events.

Constructors

`JLabel( String text )`

creates a *JLabel* object that displays the specified *text*.

`JLabel( String text, int alignment )`

creates a *JLabel* object that displays the specified *text*. The *alignment* argument specifies the alignment of the text within the label component. The *alignment* value can be any of the following *static int* constants of the *SwingConstants* interface: LEFT, CENTER, RIGHT, LEADING, or TRAILING. By default, the label text is left-adjusted.

`JLabel( Icon image )`

creates a *JLabel* object that displays the *image*.

Useful Methods of the *JLabel* Class

Return value	Method name and argument list
`void`	`setIcon( Icon newIcon )`
	sets the *Icon* to be displayed in the label as *newIcon*.
`void`	`setText( String newText )`
	sets the text to be displayed in the label as *newText*.

JList

Package: javax.swing

Description: a list user interface component. When a user selects one or more items from the list, an *ItemEvent* is fired.

Constructor

`JList( Object [ ] arrayName )`

constructs a new *JList* component initially filled with the objects in *arrayName*. Often, the objects are *Strings*.

Useful Methods of the *JList* Class

Return value	Method name and argument list
`void`	`addListSelectionListener( ListSelectionListener handler )` registers an event handler for *ItemEvents* fired by this list.
`int`	`getSelectedIndex( )` returns the index of the selected item. The index of the first item in the list is 0.
`void`	`setSelectedIndex( int index )` selects the item at *index*. The index of the first item in the list is 0.
`void`	`setSelectionMode( int selectionMode )` sets the number of selections that can be made at one time. The following *static int* constants of the *ListSelectionModel* interface can be used to set the selection mode: SINGLE_SELECTION—one selection allowed SINGLE_INTERVAL_SELECTION—multiple contiguous items can be selected MULTIPLE_INTERVAL_SELECTION—multiple contiguous intervals can be selected (This is the default.)

JOptionPane

Package: javax.swing

Description: pops up an input or output dialog box.

Useful Methods of the *JOptionPane* Class

Return value	Method name and argument list
`String`	`showInputDialog( Component parent, Object prompt )` *static* method that pops up an input dialog box, where *prompt* asks the user for input. Returns the characters typed by the user as a *String*.
`void`	`showMessageDialog( Component parent, Object message )` *static* method that pops up an output dialog box with *message* displayed. The *message* argument is usually a *String*.

JPasswordField

Package: javax.swing

Description: A single-line text field user interface component that allows users to enter text, such as a password, without the text being displayed. When a user presses the *Enter* key with the cursor in the field, an *ActionEvent* is fired.

Constructor

`JPasswordField( int numberColumns )`

constructs an empty password field with the specified number of columns.

Useful Methods of the *JPasswordField* Class

Return value	Method name and argument list
`void`	`addActionListener( ActionListener handler )`
	registers an event handler for this password field. This method is inherited from the *JTextField* class.
`char [ ]`	`getPassword( )`
	returns the text entered in this password field as an array of *chars*.
`void`	`setEchoChar( char c )`
	sets the echo character of the password field to *c*.
`void`	`setEditable( boolean mode )`
	sets the properties of the password field as editable or non-editable, depending on whether *mode* is *true* or *false*. The default is editable. This method is inherited from the *JTextComponent* class.
`void`	`setText( String newText )`
	sets the text of the password field to *newText*. This method is inherited from the *JTextComponent* class.

JRadioButton

Package: javax.swing

Description: a radio button user interface component. When a user presses the button, other radio buttons in the *ButtonGroup* are deselected and an *ItemEvent* is fired.

Constructors

`JRadioButton( String buttonLabel )`

constructs a radio button labeled *buttonLabel*. By default, the radio button is initially deselected.

`JRadioButton( String buttonLabel, boolean selected )`

constructs a radio button labeled *buttonLabel*. If *selected* is *true*, the button is initially selected; if *selected* is *false*, the button is deselected.

Useful Methods of the *JRadioButton* Class

Return value	Method name and argument list
`void`	`addItemListener( ItemListener handler )`
	registers an event handler for *ItemEvents* for this radio button. This method is inherited from the *AbstractButton* class.
`boolean`	`isSelected( )`
	returns *true* if the radio button is selected; *false* otherwise. This method is inherited from the *AbstractButton* class.
`void`	`setSelected( boolean state )`
	selects the radio button if *state* is *true*; deselects the radio button if *state* is *false*. This method is inherited from the *AbstractButton* class.

JTextArea

Package: javax.swing

Description: a multi-line text field user interface component. When a user presses the *Enter* key with the cursor in the field, an *ActionEvent* is fired.

Constructors

`JTextArea( String text )`

constructs a text area initially filled with *text*.

`JTextArea( int numRows, int numColumns )`

constructs an empty text area with the number of rows and columns specified by *numRows* and *numColumns*.

`JTextArea( String text, int numRows, int numColumns )`

constructs a text area initially filled with *text*, and with the number of rows and columns specified by *numRows* and *numColumns*.

Useful Methods of the *JTextArea* Class

Return value	Method name and argument list
`String`	`getText( )`
	returns the text contained in the text area. This method is inherited from the *JTextComponent* class.
`void`	`setEditable( boolean mode )`
	sets the properties of the text area as editable or non-editable, depending on whether *mode* is *true* or *false*. The default is editable. This method is inherited from the *JTextComponent* class.
`void`	`setText( String newText )`
	sets the text of the text area to *newText*. This method is inherited from the *JTextComponent* class.

JTextField

Package: javax.swing

Description: a single-line text field user interface component. When a user presses the *Enter* key with the cursor in the field, an *ActionEvent* is fired.

Constructors

`JTextField( String text, int numColumns )`

constructs a new text field initially filled with *text*, with the specified number of columns.

`JTextField( int numberColumns )`

constructs an empty text field with the specified number of columns.

Useful Methods of the *JTextField* Class

Return value	Method name and argument list
`void`	`addActionListener( ActionListener handler )`
	registers an event handler for *ActionEvents* fired by this text field.

`String` `getText( )`

returns the text contained in the text field. This method is inherited from the *JTextComponent* class.

`void` `setEditable( boolean mode )`

sets the properties of the text field as editable or non-editable, depending on whether *mode* is *true* or *false*. The default is editable. This method is inherited from the *JTextComponent* class.

`void` `setText( String newText )`

sets the text of the text field to *newText*. This method is inherited from the *JTextComponent* class.

ListSelectionListener Interface

Package: javax.swing.event

Description: interface implemented by a class that will handle *ListSelectionEvents* fired by a user interface component, such as a *JList*.

Interface Method to be Implemented

`public void valueChanged( ListSelectionEvent e )`

An event handler that implements the *ListSelectionListener* interface writes code in this method to respond to the *ListSelectionEvent* fired by any registered components.

Math

Package: java.lang

Description: provides methods for performing common mathematical computations. All methods are *static*.

Predefined *static* Constants

	Data type	Description
`E`	`double`	the base of the natural logarithm. Approximate value is 2.78.
`PI`	`double`	pi, the ratio of the circumference of a circle to its diameter. Approximate value is 3.14.

Math Class Method Summary Note: All methods are *static*.	
Return value	**Method name and argument list**
`dataTypeOfArg`	`abs( arg )` returns the absolute value of the argument *arg*, which can be a *double*, *float*, *int*, or *long*.
`double`	`log( double a )` returns the natural logarithm (in base e) of its argument. For example, log(1) returns 0 and log(*Math.E*) returns 1.
`dataTypeOfArgs`	`max( argA, argB )` returns the larger of the two arguments. The arguments can be *doubles*, *floats*, *ints*, or *longs*.
`dataTypeOfArgs`	`min( argA, argB )` returns the smaller of the two arguments. The arguments can be *doubles*, *floats*, *ints*, or *longs*.
`double`	`pow( double base, double exp )` returns the value of *base* raised to the *exp* power.
`int`	`round( float a )` returns the closest integer to its argument, *a*.
`double`	`sqrt( double a )` returns the positive square root of *a*.

MouseEvent

Package: java.awt.event

Description: object containing information relating to a mouse event generated by the user moving or dragging the mouse or clicking its buttons on a component. This event is passed to any *MouseListener* or *MouseMotionListener* registered on the component.

Useful Methods of the *MouseEvent* Class

Return value	Method name and argument list
`int`	`getX( )`
	returns the *x* value of the (*x*,*y*) coordinate of the mouse activity.
`int`	`getY( )`
	returns the *y* value of the (*x*,*y*) coordinate of the mouse activity.

MouseListener Interface

Package: java.awt.event

Description: interface implemented by a class that will handle *MouseEvents* (press, release, click, enter, exit).

Interface Methods to be Implemented

`public void mouseClicked( MouseEvent e )`

called when the mouse button is pressed and released on a registered component.

`public void mouseEntered( MouseEvent e )`

called when the mouse enters a registered component.

`public void mouseExited( MouseEvent e )`

called when the mouse exits a registered component.

`public void mousePressed( MouseEvent e )`

called when the mouse button is pressed on a registered component.

`public void mouseReleased( MouseEvent e )`

called when the mouse is released after being pressed on a registered component.

MouseMotionListener Interface

Package: java.awt.event

Description: interface implemented by a class that will handle *MouseEvents* (move, drag).

Interface Methods to be Implemented

`public void mouseDragged( MouseEvent e )`

called when the mouse is dragged after its button is pressed on a registered component.

`public void mouseMoved( MouseEvent e )`

called when the mouse is moved onto a registered component.

NumberFormat

Package: java.text

Description: provides methods for formatting numbers in currency, percent, and other formats. There are no constructors for this class.

Useful Methods of the *NumberFormat* Class

Return value	Method name and argument list
`String`	`format( double number )`
	returns a *String* representation of *number* formatted according to the *NumberFormat* object reference used to call the method.
`NumberFormat`	`getCurrencyInstance( )`
	static method that creates a format for printing money.
`NumberFormat`	`getPercentInstance( )`
	static method that creates a format for printing a percentage.

ObjectInputStream

Package: java.io

Description: reads serialized objects from a file.

Constructor

`ObjectInputStream( InputStream in )`

constructs an *ObjectInputStream* from the *InputStream in*. Throws an *IOException*.

A Useful Method of the *ObjectInputStream* Class

Return value	Method name and argument list
`Object`	`readObject( )`

reads the next object and returns it. The object read must be an instance of a class that implements the *Serializable* interface. When the end of the file is reached, an *EOFException* is thrown. Also throws an *IOException* and *ClassNotFoundException*.

ObjectOutputStream

Package: java.io

Description: writes objects in a serialized format to a file

Constructor

`ObjectOutputStream( OutputStream out )`

creates an *ObjectOutputStream* that writes to the *OutputStream out*. Throws an *IOException*.

A Useful Method of the *ObjectOutputStream* Class

Return value	Method name and argument list
`void`	`writeObject( Object obj )`

writes the object *obj* to a file. That object must be an instance of a class that implements the *Serializable* interface. Throws an *InvalidClassException*, *NotSerializableException*, and *IOException*.

Polygon

Package: java.awt

Description: encapsulates a polygon represented by a set of (x, y) coordinates that are the vertices of a polygon.

Constructor

`Polygon( )`

creates an empty *Polygon*.

A Useful Method of the *Polygon* Class

Return value	Method name and argument list
`void`	`addPoint( int x, int y )`

appends the coordinate to the polygon.

PrintWriter

Package: java.io

Description: writes primitive data types and *Strings* to a text file.

Constructor

`PrintWriter( OutputStream os )`

constructs a *PrintWriter* object from the *OutputStream* object.

Useful Methods of the *PrintWriter* class

Return value	Method name and argument list
`void`	`close( )` releases the resources associated with the *PrintWriter* object.
`void`	`print( boolean b )` prints the *boolean b* to the *OutputStream*.
`void`	`print( char c )` prints the character *c* to the *OutputStream*.
`void`	`print( double d )` prints the *double d* to the *OutputStream*.
`void`	`print( int i )` prints the *int i* to the *OutputStream*.
`void`	`print( String s )` prints the *String s* to the *OutputStream*.
`void`	`println( boolean b )` prints the *boolean b* to the *OutputStream* and appends a newline.
`void`	`println( char c )` prints the character *c* to the *OutputStream* and appends a newline.
`void`	`println( double d )` prints the *double d* to the *OutputStream* and appends a newline.
`void`	`println( int i )` prints the *int i* to the *OutputStream* and appends a newline.

`void println( String s )`

prints the *String s* to the *OutputStream* and appends a newline.

Random

Package: java.util

Description: generates random numbers.

Constructor

`Random( )`

creates a random number generator.

A Useful Method of the *Random* Class

Return value	Method name and argument list
`int`	`nextInt ( int number )`

returns a random number ranging from 0 up to, but not including, *number* in uniform distribution.

Scanner

Package: java.util

Description: provides support for reading from an input stream or file.

Constructors

`Scanner( InputStream source )`

creates a *Scanner* object for reading from *source*. If *source* is *System.in*, this instantiates a *Scanner* object for reading from the Java console.

`Scanner( File source )`

creates a *Scanner* object for reading from a file. (See the *File* class.) Throws a *FileNotFoundException*.

`Scanner( String source )`

creates a *Scanner* object for reading from and parsing a *String*.

Selected Methods of the *Scanner* Class

Return value	Method name and argument list
`void`	`close( )`
	releases resources associated with an open input stream.
`boolean`	`hasNext( )`
	returns *true* if there is another token in the input stream; *false*, otherwise.
`boolean`	`hasNextBoolean( )`
	returns *true* if the next token in the input stream can be read as a *boolean*; *false*, otherwise.
`boolean`	`hasNextByte( )`
	returns *true* if the next token in the input stream can be read as a *byte*; *false*, otherwise.
`boolean`	`hasNextDouble( )`
	returns *true* if the next token in the input stream can be read as a *double*; *false*, otherwise.
`boolean`	`hasNextFloat( )`
	returns *true* if the next token in the input stream can be read as a *float*; *false*, otherwise.
`boolean`	`hasNextInt( )`
	returns *true* if the next token in the input stream can be read as an *int*; *false*, otherwise.
`boolean`	`hasNextLong( )`
	returns *true* if the next token in the input stream can be read as a *long*; *false*, otherwise.
`boolean`	`hasNextShort( )`
	returns *true* if the next token can be read as a *short*; *false*, otherwise.
`String`	`next( )`
	returns the next token in the input stream as a *String*.

`boolean`	`nextBoolean( )` returns the next input token as a *boolean*. Throws an *InputMismatchException*.
`byte`	`nextByte( )` returns the next input token as a *byte*. Throws an *InputMismatchException*.
`double`	`nextDouble( )` returns the next input token as a *double*. Throws an *InputMismatchException*.
`float`	`nextFloat( )` returns the next input token as a *float*. Throws an *InputMismatchException*.
`int`	`nextInt( )` returns the next input token as an *int*. Throws an *InputMismatchException*.
`String`	`nextLine( )` returns the remainder of the input line as a *String*.
`long`	`nextLong( )` returns the next input token as a *long*. Throws an *InputMismatchException*.
`short`	`nextShort( )` returns the next input token as a *short*. Throws an *InputMismatchException*.
`Scanner`	`useDelimiter( String pattern )` sets the delimiter to *pattern* and returns this *Scanner*.

String

Package: java.lang

Description: provides support for storing, searching, and manipulating sequences of characters.

Constructors

`String( String str )`

creates a *String* object with the value of *str*, which can be a *String* object or a *String* literal.

`String( )`

creates an empty *String* object.

`String( char [ ] charArray )`

creates a *String* object containing the characters in the *char* array *charArray*.

Methods

Return value	Method name and argument list
`char`	`charAt( int index )`
	returns the character at the position specified by *index*. The first index is 0.
`int`	`compareTo( String str )`
	compares the value of the two *Strings*. If the *String* object is less than the argument, a negative integer is returned. If the *String* object is greater than the *String* argument, a positive number is returned; if the two *Strings* are equal, a 0 is returned.
`boolean`	`equals( Object str )`
	compares the value of two *Strings*. Returns *true* if *str* is a *String*, is not *null*, and is equal to the *String* object; *false* otherwise.
`boolean`	`equalsIgnoreCase( String str )`
	compares the value of two *Strings*, treating upper- and lowercase characters as equal. Returns *true* if the *Strings* are equal; *false* otherwise.
`int`	`indexOf( char searchChar )`
	returns the index of the first occurrence of *searchChar* in the *String*. Returns –1 if not found.
`int`	`indexOf( String substring )`
	returns the index of the first occurrence of *substring* in the *String*. Returns –1 if not found.

`int`	`length( )` returns the number of characters in the *String*.
`String`	`substring( int startIndex, int endIndex )` returns a substring of the *String* object beginning at the character at index *startIndex* and ending at the character at index (*endIndex* – *1*).
`String`	`toLowerCase( )` converts all letters in the *String* to lowercase.
`String`	`toUpperCase( )` converts all letters in the *String* to uppercase.

System

Package: java.lang

System.out

The *out* class constant of the *System* class is a *PrintStream* object, which represents the standard system output device. The following *PrintStream* methods can be called using the object reference **System.out** in order to print to the Java console.

Methods

Return value	Method name and argument list
`void`	`print( argument )` prints *argument* to the standard output device. The argument is usually any primitive data type or a *String* object.
`void`	`println( argument )` prints *argument* to the standard output device, then prints a new-line character. The argument is usually any primitive data type or a *String* object.

APPENDIX G

Solutions to Selected Exercises

1.7 Exercises, Problems, and Projects

1.7.1 Multiple Choice Exercises:

1. Java
4. servers.
7. is a multiple of 4.
10. C
13. *javac Hello.java*

1.7.2 Converting Numbers

16. 11000011100
19. 0x15

1.7.3 General Questions

22. 1.5 billion
25. red = 51; green = 171; blue = 18
28. *javac*

2.7 Exercises, Problems, and Projects

2.7.1 Multiple Choice Exercises

1. `int a;`

2.7.2 Reading and Understanding Code

4. 12.5
7. 2.0
10. 4
13. 5
16. 2.4
19. 5
22. 0

2.7.3 Fill In the Code

25.
```
boolean a;
a = false;
```

28.
```
float avg = (float) ( a + b ) / 2;
System.out.println( "The average is " + avg );
```

31. `a *= 3;`

2.7.4 Identifying Errors in Code

34. Cannot assign a *double* to a *float* variable (possible loss of precision).
37. There should not be a space between – and =.

2.7.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

40. Cannot assign a *double* to an *int* variable (possible loss of precision). Change to:

```
int a = 26;
```

43. =+ is different from += (shortcut operator). Here, *a* is assigned the value + 3. To add 3 to *a*, change the second statement to:

```
a += 3;
```

3.19 Exercises, Problems, and Projects

3.19.1 Multiple Choice Exercises

1. `import`

4. `new`

7. It is a class method.

10. `double`

13. `Math.E;`

3.19.2 Reading and Understanding Code

16. hello

19. 3.141592653589793

22. 8

3.19.3 Fill In the Code

25.
```
System.out.println( s.length( ) );
```

28.
```
System.out.print( "Welcome" );
System.out.print( "to" );
System.out.print( "Java" );
System.out.print( "Illuminated\n" );
```

31.
```
// code below assumes we have imported Scanner
Scanner scan = new Scanner( System.in );
System.out.print( "Enter two integers > " );
int i = scan.nextInt( );
int j = scan.nextInt( );
int min = Math.min( i, j );
System.out.println( "min of " + i + " and " + j + " is " + min );
```

34.
```
// code below assumes we have imported Scanner
Scanner scan = new Scanner( System.in );
System.out.print( "Enter a double > " );
double number = scan.nextDouble( );
double square = Math.pow( number, 2 );
System.out.println( number + " square = " + square );
```

3.19.4 Identifying Errors in Code

37. The Java compiler does not recognize system. It should be *System*, not system.

40. The *round* method of the *Math* class returns a *long*; a *long* cannot be assigned to a *short* variable due to a potential loss of precision.

43. The *char 'H'* cannot be assigned to the *String s*. The two data types are not compatible.

3.19.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

46. Java is case sensitive. The *Math* class needs to be spelled with an uppercase M.

49. In the output statement, we are just printing the value of *grade* without any formatting. To format *grade* as a percentage, the output statement should be:

```
System.out.println( "Your grade is " + percent.format( grade ) );
```

4.7 Exercises, Problems, and Projects

4.7.1 Multiple Choice Exercises

1. *java.awt*

4. true

7. the (x, y) coordinate of the upper-left corner of the rectangle we are drawing

10. 256

4.7.2 Reading and Understanding Code

13. 250 pixels

4.7.3 Fill In the Code

16. `g.setColor( Color.RED );`

19. `g.fillRect( 50, 30, 50, 270 );`

4.7.4 Identifying Errors in Code

22. There should be double quotes around the literal *Find a bug*, not single quotes. Single quotes are used for a *char*, not a *String*.

25. There is no *public color* instance variable in the *Graphics* class. The *setColor* mutator method should be used to set the color of the *Graphics* object.

4.7.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

28. We are trying to override the *paint* method, which is an instance method. The header of *paint* should therefore not include the keyword *static*.

5.14 Exercises, Problems, and Projects

5.14.1 Multiple Choice Exercises

1.

- ❑ `a < b` — `true`
- ❑ `a != b` — `true`
- ❑ `a == 4` — `false`
- ❑ `( b – a ) <= 1` — `false`
- ❑ `Math.abs( a – b ) >= 2` — `true`
- ❑ `( b % 2 == 1 )` — `true`
- ❑ `b <= 5` — `true`

4. yes

7.

- ❑ `a < b || b < 10` — `no`
- ❑ `a != b && b < 10` — `yes`
- ❑ `a == 4 || b < 10` — `yes`
- ❑ `a > b && b < 10` — `no`

5.14.2 Reading and Understanding Code

10. *true*

13. 27 is divisible by 3
End of sequence

16. Hello 3
    Hello 4
    Done

19. Number 3
    Number 4
    Other number

5.14.3 Fill In the Code

22.
```
if ( a )
  a = false;
else
  a = true;
```

25.
```
if ( b % c == 0 )
  a = true;
else
  a = false;
```

28.
```
if ( a && b > 10 )
  c++;
```

5.14.4 Identifying Errors in Code

31. The *&&* operator cannot be applied to two *int* operands (*a1* and *a2*).

34. We need a set of parentheses around *b1*.

37. There is no error.

5.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

40. The expression `a = 31` evaluates to an *int*, 31. The *if* condition requires a *boolean* expression. To fix the problem, replace `a = 31` with `a == 31`.

6.14 Exercises, Problems, and Projects

6.14.1 Multiple Choice Exercises

1. The code runs forever.

4. true

6.14.2 Reading and Understanding Code

7. Enter an int > 3
   Enter an int > 5
   Hello
   Enter an int > –1
   Hello

10. 8 and 42

13. 3

16. 40 and 60

19. 3
    3
    3
    3
    4

6.14.3 Fill In the Code

22.
```
System.out.print( "Enter an integer > " );
int value = scan.nextInt( );
while ( value != 20 )
{
  if ( value >= start )
    System.out.println( value );
  System.out.print( "Enter an integer > " );
  value = scan.nextInt( );
}
```

25.
```
Scanner scan = new Scanner( System.in );
word = scan.next( );
while ( ! word.equals( "end" ) )
{
  // and your code goes here
  sentence += word;
  word = scan.next( );
}
```

28.
```
Scanner scan = new Scanner( System.in );
int sum = 0;
System.out.println( "Enter an integer > " );
int value = scan.nextInt( );
while ( value != 0 && value != 100 )
```

```
{
  sum += value;
  System.out.println( "Enter an integer > " );
  value = scan.nextInt( );
}
System.out.println( "sum is " + sum );
```

6.14.4 Identifying Errors in Code

31. The variable *num* needs to be initialized after it is declared, and a priming read is needed.

34. The loop is infinite. *Number* is always different from 5 or different from 7. The logical OR (||) should be changed to a logical AND (&&).

6.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

37. It is an infinite loop; *i* should be incremented, not decremented, inside the body of the *while* loop so that the loop eventually terminates.

40. In the *for* loop header, the loop initialization statement, the loop condition, and the loop update statement should be separated by semicolons (;), not commas(,).

7.18 Exercises, Problems, and Projects

7.18.1 Multiple Choice Exercises

1. The convention is to start with an uppercase letter.

4. true

7. can be basic data types, existing Java types, or user-defined types (from user-defined classes).

10. one parameter, of the same type as the corresponding field.

13. these fields do not need to be passed as parameters to the methods because the class methods have direct access to them.

16. All of the above.

7.18.2 Reading and Understanding Code

19. *double*

22. an instance method (keyword *static* not used)

25.
```
public static void foo3( double d );
```

7.18.3 Fill In the Code

28.
```
private int grade;
private char letterGrade;
```

31.
```
public TelevisionChannel( String newName, int newNumber,
                          boolean newCable )
{
  name = newName;
  number  = newNumber;
  cable = newCable;
}
```

34.
```
public String toString( )
{
  return ( "name: " + name + "\tnumber: "
           + number + "\tcable: " + cable );
}
```

37.
```
public String typeOfChannel( )
{
  if ( cable )
    return "cable";
  else
    return "network";
}
```

7.18.4 Identifying Errors in Code

40. The *toString* method needs to return a *String*, not output data.

43. The method header is incorrect; it should be

```
public double calcTax( )
```

46. There are two errors: The assignment operator = should not be used when declaring an *enum* set. And the *enum* constant objects should not be *String* literals but identifiers.

7.18.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

49. The compiler understands that *Grade* is a method since its header says it returns a *char*. It looks as if it is intended to be a constructor so the keyword *char* should be deleted from the constructor header.

52. The constructor assigns the parameter *letterGrade* to itself, therefore not changing the value of the instance variable *letterGrade*, which by default is the space character. The constructor could be recoded as follows:

```
public Grade( char newLetterGrade )
{
  letterGrade = newLetterGrade;
}
```

8.10 Exercises, Problems, and Projects

8.10.1 Multiple Choice Exercises:

1. `int [ ] a;` and `int a[ ];`

4. 0

7. `a.length`

10. false

8.10.2 Reading and Understanding Code

13. 48.3

16. 12

    48

    65

19. 14

22. It counts how many elements in the argument array have the value 5.

25. It returns an array of *Strings* identical to the argument array except that the *Strings* are all in lowercase.

8.10.3 Fill In the Code

28.
```
if ( a[i] > 20 )
  System.out.println( a[i] );
```

31.
```
System.out.println( "a[" + i + "] = " + a[i] );
```

34.
```
if ( a.length < 2 )
  return false;
else if ( a[0].equals( a[1] ) )
  return true;
else
  return false;
```

8.10.4 Identifying Errors in Code

37. Index –1 is out of bounds; the statement `System.out.println( a[-1] );` will generate a run-time exception.

40. When declaring an array, the square brackets should be empty. Replace `a[3]` by `a[ ]`.

43. Although the code compiles, it outputs the hash code of the array *a*. To output the elements of the array, we need to loop through the array elements and output them one by one.

8.10.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

46. Index `a.length` is out of bounds; when *i* is equal to `a.length`, the expression `a[i]` will generate a run-time exception. Replace `<=` with `<` in the loop condition.

9.10 Exercises, Problems, and Projects

9.10.1 Multiple Choice Exercises

1. `int[ ][ ] a;` and `int a[ ][ ];`

4. *false*

7. `a[2].length`

10. true

13. *java.util*

9.10.2 Reading and Understanding Code

16. 3

19. Munich
    Stuttgart
    Berlin
    Bonn

22. Munich
    Berlin
    Ottawa

25. It counts and returns the number of elements in the argument array *a*.

28. It returns an *int* array of the same length as the length of the array argument *a*. Each element of the returned array stores the number of columns of the corresponding row in the array argument *a*.

31. 7 (at index 0) 45 (at index 1) 21 (at index 2)

9.10.3 Fill In the Code

34.
```
System.out.println( geo[0][5] );
```

37.
```
for ( int i = 0; i < geo.length; i++ )
{
  for ( int j = 0; j < geo[i].length; j++ )
      System.out.println( geo[i][j] );
}
```

40.
```
int count = 0;
for ( int j = 0; j < a[1].length; j++ )
{
  if ( a[1][j] == 6 )
      count++;
}
System.out.println( "# of 6s in the 2nd row: " + count );
```

43. This method returns the product of all the elements in an array.

```
public static int foo( int [ ][ ] a )
{
  int product = 1;
  for ( int i = 0; i < a.length; i++ )
  {
    for ( int j = 0; j < a[i].length; j++ )
    {
      product *= a[i][j];
    }
  }
  return product;
}
```

46.
```
System.out.println( languages.size( ) );
```

49.
```
for ( String s : languages )
{
  if ( s.charAt( 0 ) == 'P' )
    System.out.println( s );
}
```

9.10.4 Identifying Errors in Code

52. Array dimension missing in `new double [ ][10]`

    Example of correct code: `double [ ][ ] a = new double [4][10];`

55. Cannot declare an *ArrayList* of a basic data type; the type needs to be a class (for example: *Double*)

58. Correct syntax is `variable = expression`. Because `a.size( )` is not a variable, we cannot assign a value to it.

9.10.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

61. Other than `a[0][0]`, the first row is not taken into account because *i* is initialized to 1 in the outer loop. It should be `int i = 0;` not `int i = 1`.

64. Index 3 is out of bounds. There are only 3 elements in *a*; the last index is 2.

67. Because *ArrayList* elements begin at index 0, the statement

    ```
    a.set( 1, 'J' );
    ```

    sets the value of the second element of the *ArrayList*. To set the value of the first element, use this statement:

    ```
    a.set( 0, 'J' );
    ```

10.10 Exercises, Problems, and Projects

10.10.1 Multiple Choice Exercises

1. a class inheriting from another class.

4. *protected* and *public* instance variables and methods

7. You cannot instantiate an object from an *abstract* class.

10. the class must be declared *abstract*.

10.10.2 Reading and Understanding Code

13. *B* inherits from *A*: *name*, *price* (*foo2* and *foo3* are overridden)

    C inherits from *B*: *name*, *price*, *foo2*, and *foo3* (*foo1* is overridden)

16.
```
A( ) called
B( ) called
B version of foo1( ) called
```

19.
```
A( ) called
B( ) called
C( ) called
```

10.10.3 Fill In the Code

22.
```
private char middle;
public G( String f, String n, char m )
{
  super( f, n );
  middle = m;
}
```

25.
```
public class K extends F implements I
```

10.10.4 Identifying Errors in Code

28. There is no error. `new D( )` returns a *D* object reference. *D* inherits from *C*; therefore a *D* object reference "is a" *C* object reference. Thus, it can be assigned to *c2*.

31. The *foo* method does not have a method body; it must be declared *abstract*. The *K* class must be declared *abstract* as well.

10.10.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

34. The instance variable *n* of class *M* is private, and is not inherited by *P*. Therefore, it is not visible inside class *P*.

11.10 Exercises, Problems, and Projects

11.10.1 Multiple Choice Exercises

1. Exceptions enable programmers to attempt to recover from illegal situations and continue running the program.

4. false

7. the contents of the file, if any, will be deleted.

10. *java.util*

11.10.2 Reading and Understanding Code

13. ABCD

16. result is ABCABA

19. Nice finish

22. CS1
    0
    1
    2
    3
    4

11.10.3 Fill In the Code

25.
```
while ( parse.hasNext( ) )
{
  s = parse.next( );
  if ( s.equals( "C" ) )
    break;
}
```

28.
```
String s = file.nextLine( );
while ( s != null )
{
  result += s + " ";
  s = file.nextLine( );
}
System.out.println( result );
```

31.
```
average += grades[i];
...
average /= grades.length;
...
pw.print( average );
pw.close( );
```

11.10.4 Identifying Errors in Code

34. The *next* method returns a *String*; the return value cannot be assigned to an *int* variable.

11.10.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

37. The *catch* block is missing; you need to add it after the *try* block as follows:

```
catch ( IOException ioe )
{
  ioe.printStackTrace( );
}
```

40. When we try to read 3.5 as an *int*, the method generates an *InputMismatchException* and we execute the *catch* block. If we change the data to 1 3 6, then the output would be

```
1
3
6
```

12.18 Exercises, Problems, and Projects

12.18.1 Multiple Choice Exercises

1. *JTextField.*
4. true
7. All three methods
10. false
13. the horizontal and vertical gaps between the five areas of the component

12.18.2 Reading and Understanding Code

16. 3
19. "Hello" is output to the console
22. 3
25. 5 is output to the console

12.18.3 Fill In the Code

28.
```
b = new JButton( "Button" );
```

31.
```
public void actionPerformed( ActionEvent ae )
{
  if ( ae.getSource( ) == b )
    tf.setText( "Button clicked" );
}
```

34.
```
c.add( label1, BorderLayout.NORTH );
```

37.
```
for ( int i = 0; i < buttons.length; i++ )
  p1.add( buttons[i] );
```

40.
```
for ( int i = 0; i < textfields.length; i++ )
  p2.add( textfields[i] );
```

12.18.4 Identifying Errors in Code

43. *ActionListener* is an interface; it can be implemented but not extended.

12.18.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

46. We are trying to add the label *l* to the content pane before *l* has been instantiated. To fix the problem, instantiate *l* before line 15 as follows:

```
l = new Jlabel( "Hello" );
```

49. *ActionListener* and *ActionEvent* need to be imported. To fix the problem, add the following *import* statement:

```
import java.awt.event.*;
```

13.10 Exercises, Problems, and Projects

13.10.1 Multiple Choice Exercises

1. may or may not be *static.*

4. calls itself.

7. a run-time error.

13.10.2 Reading and Understanding Code

10. 0

13. 3

16. There is no output

19. *foo3* outputs the argument *String* in reverse

22. 64

13.10.3 Fill In the Code

25.
```
return foo( s.substring( 2, s.length( ) ) );
```

28.
```
if ( n >= 1000 )      // base case
  return n;
else                  // general case
  return foo( n * n );
```

13.10.4 Identifying Errors in Code

31. In the *else* clause, the *return* keyword is missing.

13.10.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

34. The base case is not coded properly; it needs to return a value, not make another recursive call. Instead of *return foo(0);*, you can code *return 1*.

37. In the general case, the method makes the recursive call with the original *String* less the last two characters as the argument. Therefore, when a *String* has an odd number of characters, an *exception* will eventually occur when a recursive call is made on a *String* containing one character. Assuming this method counts the number of characters in the *String* argument, we can change the code as follows:

```
else
  return ( 1 +
    foo( s.substring( 0, s.length( ) - 1 ) ) ); // line 15
```

14.14 Exercises, Problems, and Projects

14.14.1 Multiple Choice Exercises

1. Linked lists are easily expanded.

4. (*7, Ajay, NFL*)

7. (*7, Ajay, NFL*) and (*5, Joe, Sonic*)

10. *front* = 3, stores 54; *back* = 0, stores 62

13. *front* = 7; *back* = 6; the list is empty

14.14.2 Reading and Understanding Code

16. If the list is not empty, it resets it to empty and returns *true*. Otherwise, it returns *false*.

19. It outputs all the *Player* objects in the list whose *game* field is *Diablo*.

14.14.3 Fill In the Code

22.
```
previous.getNext( ).setNext(
                    previous.getNext( ).getNext( ).getNext( ) );
```

25.
```
public LLNode( char newGrade )
{
  grade = newGrade;
  next = null;
}
```

28.
```
public void setGrade( char newGrade )
{
  grade = newGrade;
}
public void setNext( LLNode newNext )
{
  next = newNext;
}
```

31.
```
public int numberOfItems( )
{
  int count = 0;
  LLNode current = head;
  while ( current != null )
  {
    count++;
    current = current.getNext( );
  }
  return count;
}
```

14.14.4 Identifying Errors in Code

34. The *getID* method belongs to the *Player* class and cannot be called using *head*, a *PlayerNode* object reference.

37. The number of items in the queue would never increase and we would always be able to insert into that queue, eventually overwriting items that are in the queue. This is a logic error. Furthermore, the queue would always be considered empty since the number of items always has the value 0. We would never be able to delete an item from the queue.

14.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

40. There is no *getHead* method in the *PlayerLinkedList* class. In order to get the *Player* object stored at the first node of the list, we can code a method returning the *Player*.

15.7 Exercises, Problems, and Projects

15.7.1 Multiple Choice Exercises

1. $O(n^2)$
4. $O(n)$
7. $n * (n+1) / 2$
10. $O(n^2)$

15.7.2 Compute the Running Time of a Method

13. $O(n)$
16. $O(\log n)$.

Index