APPENDIXES

Appendix A
Java Keywords and Reserved Words

Appendix B
The ASCII Character Set

Appendix C
Operator Precedence Chart

Appendix D
Java Modifiers

Appendix E
Special Floating-Point Values

Appendix F
Number Systems

Appendix G
Bitwise Operations

Appendix H
Regular Expressions

Appendix I
Enumerated Types

Appendix J
The Big-O, Big-Omega, and Big-Theta Notations

Appendix A

Java Keywords and Reserved Words

Keywords have special meaning in Java and are part of the syntax. Reserved words are the words that cannot be used as identifiers. Keywords are reserved words. The following 50 keywords are reserved for use by the Java language:

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

The keywords **goto** and **const** are C++ keywords reserved, but not currently used in Java. This enables Java compilers to identify them and to produce better error messages if they appear in Java programs.

The literal values **true**, **false**, and **null** are reserved words, but not keywords. You cannot use them as identifiers.

In the code listing, we use the keyword color for **true**, **false**, and **null** to be consistent with their coloring in Java IDEs.

*The **strictfp** keyword is a modifier for a method or class that enables it to use strict floating-point calculations. Floating-point arithmetic can be executed in one of two modes: *strict* or *nonstrict*. The strict mode guarantees that the evaluation result is the same on all Java Virtual Machine implementations. The nonstrict mode allows intermediate results from calculations to be stored in an extended format different from the standard IEEE floating-point number format. The extended format is machine dependent and enables code to be executed faster. However, when you execute the code using the nonstrict mode on different JVMs, you may not always get precisely the same results. By default, the nonstrict mode is used for floating-point calculations. To use the strict mode in a method or a class, add the **strictfp** keyword in the method or the class declaration. Strict floating-point may give you slightly better precision than nonstrict floating-point, but the distinction will only affect some applications. Strictness is not inherited; that is, the presence of **strictfp** on a class or interface declaration does not cause extended classes or interfaces to be strict.

Appendix B

The ASCII Character Set

Tables B.1 and B.2 show ASCII characters and their respective decimal and hexadecimal codes. The decimal or hexadecimal code of a character is a combination of its row index and column index. For example, in Table B.1, the letter A is at row 6 and column 5, so its decimal equivalent is 65; in Table B.2, letter A is at row 4 and column 1, so its hexadecimal equivalent is 41.

Table B.1 ASCII Character Set in the Decimal Index

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dcl	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	”	#	$	%	&	’
4	(	)	*	+	,	–	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	∧	–	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Table B.2 ASCII Character Set in the Hexadecimal Index

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	nl	vt	ff	cr	so	si
1	dle	dcl	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
2	sp	!	”	#	$	%	&	’	(	)	*	+	,	–	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	∧	–
6	’	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	del

APPENDIX C

Operator Precedence Chart

The operators are shown in decreasing order of precedence from top to bottom. Operators in the same group have the same precedence, and their associativity is shown in the table.

Operator	*Name*	*Associativity*
`()`	Parentheses	Left to right
`()`	Function call	Left to right
`[]`	Array subscript	Left to right
`.`	Object member access	Left to right
`++`	Postincrement	Left to right
`--`	Postdecrement	Left to right
`++`	Preincrement	Right to left
`--`	Predecrement	Right to left
`+`	Unary plus	Right to left
`-`	Unary minus	Right to left
`!`	Unary logical negation	Right to left
`(type)`	Unary casting	Right to left
`new`	Creating object	Right to left
`*`	Multiplication	Left to right
`/`	Division	Left to right
`%`	Remainder	Left to right
`+`	Addition	Left to right
`-`	Subtraction	Left to right
`<<`	Left shift	Left to right
`>>`	Right shift with sign extension	Left to right
`>>>`	Right shift with zero extension	Left to right
`<`	Less than	Left to right
`<=`	Less than or equal to	Left to right
`>`	Greater than	Left to right
`>=`	Greater than or equal to	Left to right
`instanceof`	Checking object type	Left to right

Operator	*Name*	*Associativity*
==	Equal comparison	Left to right
!=	Not equal	Left to right
&	(Unconditional AND)	Left to right
^	(Exclusive OR)	Left to right
|	(Unconditional OR)	Left to right
&&	Conditional AND	Left to right
||	Conditional OR	Left to right
?:	Ternary condition	Right to left
=	Assignment	Right to left
+=	Addition assignment	Right to left
-=	Subtraction assignment	Right to left
*=	Multiplication assignment	Right to left
/=	Division assignment	Right to left
%=	Remainder assignment	Right to left

APPENDIX D

Java Modifiers

Modifiers are used on classes and class members (constructors, methods, data, and class-level blocks), but the `final` modifier can also be used on local variables in a method. A modifier that can be applied to a class is called a *class modifier*. A modifier that can be applied to a method is called a *method modifier*. A modifier that can be applied to a data field is called a *data modifier*. A modifier that can be applied to a class-level block is called a *block modifier*. The following table gives a summary of the Java modifiers.

Modifier	*Class*	*Constructor*	*Method*	*Data*	*Block*	*Explanation*
`(blank)`*	√	√	√	√	√	A class, constructor, method, or data field is visible in this package.
`public`	√	√	√	√		A class, constructor, method, or data field is visible to all the programs in any package.
`private`		√	√	√		A constructor, method, or data field is only visible in this class.
`protected`		√	√	√		A constructor, method, or data field is visible in this package and in subclasses of this class in any package.
`static`			√	√	√	Define a class method, a class data field, or a static initialization block.
`final`	√		√	√		A final class cannot be extended. A final method cannot be modified in a subclass. A final data field is a constant.
`abstract`	√		√			An abstract class must be extended. An abstract method must be implemented in a concrete subclass.
`native`			√			A native method indicates that the method is implemented using a language other than Java.

* (blank) means no modifiers are used. For example: `class Test {}`

Modifier	*Class*	*Constructor*	*Method*	*Data*	*Block*	*Explanation*
`synchronized`			√		√	Only one thread at a time can execute this method.
`strictfp`	√		√			Use strict floating-point calculations to guarantee that the evaluation result is the same on all JVMs.
`transient`				√		Mark a nonserializable instance data field.

The modifiers `public`, `private`, and `protected` are known as *visibility* or *accessibility modifiers* because they specify how classes and class members are accessed.

The modifiers `public`, `private`, `protected`, `static`, `final`, and `abstract` can also be applied to inner classes.

Java 8 introduced the `default` modifier for declaring a default method in an interface. A default method provides a default implementation for the method in the interface.

Appendix E

Special Floating-Point Values

Dividing an integer by zero is invalid and throws `ArithmeticException`, but dividing a floating-point value by zero does not cause an exception. Floating-point arithmetic can overflow to infinity if the result of the operation is too large for a `double` or a `float`, or underflow to zero if the result is too small for a `double` or a `float`. Java provides the special floating-point values `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, and `NaN` (Not a Number) to denote these results. These values are defined as special constants in the `Float` class and the `Double` class.

If a positive floating-point number is divided by zero, the result is `POSITIVE_INFINITY`. If a negative floating-point number is divided by zero, the result is `NEGATIVE_INFINITY`. If a floating-point zero is divided by zero, the result is `NaN`, which means that the result is undefined mathematically. The string representations of these three values are `Infinity`, `-Infinity`, and `NaN`. For example,

```
System.out.print(1.0 / 0); // Print Infinity
System.out.print(–1.0 / 0); // Print –Infinity
System.out.print(0.0 / 0); // Print NaN
```

These special values can also be used as operands in computations. For example, a number divided by `POSITIVE_INFINITY` yields a positive zero. Table E.1 summarizes various combinations of the `/`, `*`, `%`, `+`, and `–` operators.

Table E.1 Special Floating-Point Values

x	*y*	*x/y*	*x*y*	*x%y*	*x + y*	*x − y*
Finite	±0.0	±infinity	±0.0	NaN	Finite	Finite
Finite	±infinity	±0.0	±0.0	x	±infinity	infinity
±0.0	±0.0	NaN	±0.0	NaN	±0.0	±0.0
±infinity	Finite	±infinity	±0.0	NaN	±infinity	±infinity
±infinity	±infinity	NaN	±0.0	NaN	±infinity	infinity
±0.0	±infinity	±0.0	NaN	±0.0	±infinity	±0.0
NaN	Any	NaN	NaN	NaN	NaN	NaN
Any	NaN	NaN	NaN	NaN	NaN	NaN

Note
If one of the operands is NaN, the result is NaN.

APPENDIX F

Number Systems

F.1 Introduction

Computers use binary numbers internally, because computers are made naturally to store and process 0s and 1s. The binary number system has two digits, 0 and 1. A number or character is stored as a sequence of 0s and 1s. Each 0 or 1 is called a *bit* (binary digit). base radix

In our daily life, we use decimal numbers. When we write a number such as 20 in a program, it is assumed to be a decimal number. Internally, computer software is used to convert decimal numbers into binary numbers, and vice versa. decimal numbers

We write computer programs using decimal numbers. However, to deal with an operating system, we need to reach down to the "machine level" by using binary numbers. Binary numbers tend to be very long and cumbersome. Often hexadecimal numbers are used to abbreviate them, with each hexadecimal digit representing four binary digits. The hexadecimal number system has 16 digits: 0–9 and A–F. The letters A, B, C, D, E, and F correspond to the decimal numbers 10, 11, 12, 13, 14, and 15. hexadecimal number

The digits in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. A decimal number is represented by a sequence of one or more of these digits. The value that each digit represents depends on its position, which denotes an integral power of 10. For example, the digits 7, 4, 2, and 3 in decimal number 7423 represent 7000, 400, 20, and 3, respectively, as shown below:

$$\boxed{7}\boxed{4}\boxed{2}\boxed{3} = 7 \times 10^3 + 4 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

$$10^3\ 10^2\ 10^1\ 10^0 = 7000 + 400 + 20 + 3 = 7423$$

The decimal number system has 10 digits, and the position values are integral powers of 10. We say that 10 is the *base* or *radix* of the decimal number system. Similarly, since the binary number system has two digits, its base is 2, and since the hex number system has 16 digits, its base is 16.

If 1101 is a binary number, the digits 1, 1, 0, and 1 represent 1×2^3, 1×2^2, 0×2^1, and 1×2^0, respectively: binary numbers

$$\boxed{1}\boxed{1}\boxed{0}\boxed{1} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$2^3\ 2^2\ 2^1\ 2^0 = 8 + 4 + 0 + 1 = 13$$

If 7423 is a hex number, the digits 7, 4, 2, and 3 represent 7×16^3, 4×16^2, 2×16^1, and 3×16^0, respectively:

$$\boxed{7}\boxed{4}\boxed{3}\boxed{2} = 7 \times 16^3 + 4 \times 16^2 + 2 \times 16^1 + 3 \times 16^0$$

$$16^3\ 16^2\ 16^1\ 16^0 = 28672 + 1024 + 32 + 3 = 29731$$

F.2 Conversions between Binary and Decimal Numbers

binary to decimal

Given a binary number $b_n b_{n-1} b_{n-2} \ldots b_2 b_1 b_0$, the equivalent decimal value is

$$b_n \times 2^n + b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \ldots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

Here are some examples of converting binary numbers to decimals:

Binary	*Conversion Formula*	*Decimal*
10	$1 \times 2^1 + 0 \times 2^0$	2
1000	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	8
10101011	$1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	171

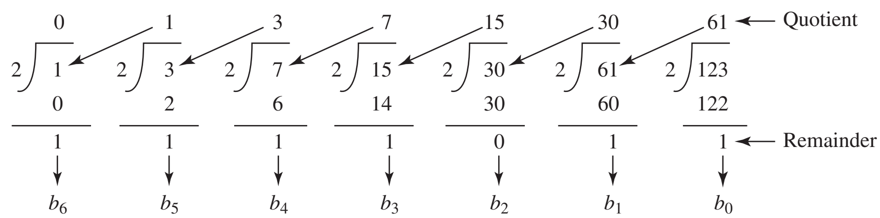
decimal to binary

To convert a decimal number d to a binary number is to find the bits $b_n, b_{n-1}, b_{n-2}, \ldots, b_2, b$ and b_0 such that

$$d = b_n \times 2^n + b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \ldots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

These bits can be found by successively dividing d by 2 until the quotient is 0. The remainders are $b_0, b_1, b_2, \ldots, b_{n-2}, b_{n-1}$, and b_n.

For example, the decimal number 123 is 1111011 in binary. The conversion is done as follows:



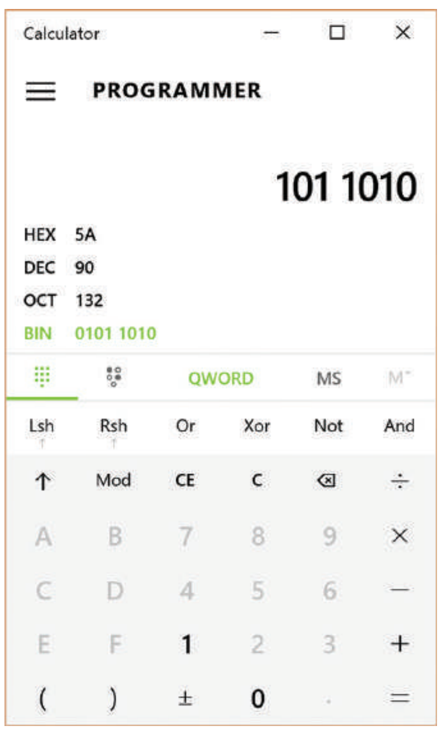


FIGURE F.1 You can perform number conversions using the Windows Calculator.

Tip

The Windows Calculator, as shown in Figure F.1, is a useful tool for performing number conversions. To run it, search for *Calculator* from the *Start* button and launch Calculator, then under *View* select *Scientific*.

F.3 Conversions between Hexadecimal and Decimal Numbers

hex to decimal

Given a hexadecimal number $h_n h_{n-1} h_{n-2} \ldots h_2 h_1 h_0$, the equivalent decimal value is

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

Here are some examples of converting hexadecimal numbers to decimals:

Hexadecimal	*Conversion Formula*	*Decimal*
7F	$7 \times 16^1 + 15 \times 16^0$	127
FFFF	$15 \times 16^3 + 15 \times 16^2 + 15 \times 16^1 + 15 \times 16^0$	65535
431	$4 \times 16^2 + 3 \times 16^1 + 1 \times 16^0$	1073

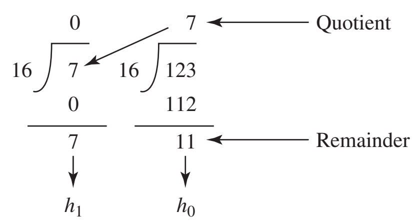
decimal to hex

To convert a decimal number d to a hexadecimal number is to find the hexadecimal digits $h_n, h_{n-1}, h_{n-2}, \ldots, h_2, h_1$, and h_0 such that

$$d = h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

These numbers can be found by successively dividing d by 16 until the quotient is 0. The remainders are $h_0, h_1, h_2, \ldots, h_{n-2}, h_{n-1}$, and h_n.

For example, the decimal number 123 is 7B in hexadecimal. The conversion is done as follows:



F.4 Conversions between Binary and Hexadecimal Numbers

hex to binary

To convert a hexadecimal number to a binary number, simply convert each digit in the hexadecimal number into a four-digit binary number, using Table F.1.

For example, the hexadecimal number 7B is 01111011, where 7 is 0111 in binary and B is 1011 in binary.

binary to hex

To convert a binary number to a hexadecimal number, convert every four binary digits from right to left in the binary number into a hexadecimal number.

For example, the binary number 001110001101 is 38D, since 1101 is D, 1000 is 8, and 0011 is 3, as shown below.

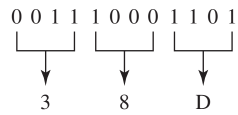


TABLE F.1 Converting Hexadecimal to Binary

Hexadecimal	*Binary*	*Decimal*
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Note

Octal numbers are also useful. The octal number system has eight digits, 0 to 7. A decimal number 8 is represented in the octal system as 10.

Check Point

F.1 Convert the following decimal numbers into hexadecimal and binary numbers:

`100; 4340; 2000`

F.2 Convert the following binary numbers into hexadecimal and decimal numbers:

`1000011001; 100000000; 100111`

F.3 Convert the following hexadecimal numbers into binary and decimal numbers:

`FEFA9; 93; 2000`

Appendix G

Bitwise Operations

To write programs at the machine-level, often you need to deal with binary numbers directly and perform operations at the bit level. Java provides the bitwise operators and shift operators defined in Table G.1.

The bit operators apply only to integer types (`byte`, `short`, `int`, and `long`). A character involved in a bit operation is converted to an integer. All bitwise operators can form bitwise assignment operators, such as `^=`, `|=`, `<<=`, `>>=`, and `>>>=`.

Table G.1

Operator	*Name*	*Example (using bytes in the example)*	*Description*
`&`	Bitwise AND	10101110 `&` 10010010 yields 10000010	The AND of two corresponding bits yields a 1 if both bits are 1.
`|`	Bitwise inclusive OR	10101110 `|` 10010010 yields 10111110	The OR of two corresponding bits yields a 1 if either bit is 1.
`^`	Bitwise exclusive OR	10101110 `^` 10010010 yields 00111100	The XOR of two corresponding bits yields a 1 only if two bits are different.
`~`	One's complement	~10101110 yields 01010001	The operator toggles each bit from 0 to 1 and from 1 to 0.
`<<`	Left shift	10101110 `<<` 2 yields 10111000	The operator shifts bits in the first operand left by the number of bits specified in the second operand, filling with 0s on the right.
`>>`	Right shift with sign extension	10101110 `>>` 2 yields 11101011 00101110 `>>` 2 yields 00001011	The operator shifts bit in the first operand right by the number of bits specified in the second operand, filling with the highest (sign) bit on the left.
`>>>`	Unsigned right shift with zero extension	10101110 `>>>` 2 yields 00101011 00101110 `>>>` 2 yields 00001011	The operator shifts bit in the first operand right by the number of bits specified in the second operand, filling with 0s on the left.

Note

Programs using the bitwise operators are more efficient than the arithmetic operators. For example, to multiply an `int` value `x` by `2`, you can write `x << 1` rather than `x * 2`.

Appendix H

Regular Expressions

Often, you need to write the code to validate user input such as to check whether the input is a number, a string with all lowercase letters, or a Social Security number. How do you write this type of code? A simple and effective way to accomplish this task is to use the regular expression.

regular expression

A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings. Regular expression is a powerful tool for string manipulations. You can use regular expressions for matching, replacing, and splitting strings.

H.1 Matching Strings

matches

Let us begin with the `matches` method in the `String` class. At first glance, the `matches` method is very similar to the `equals` method. For example, the following two statements both evaluate to `true`:

```
"Java".matches("Java");
"Java".equals("Java");
```

However, the `matches` method is more powerful. It can match not only a fixed string, but also a set of strings that follow a pattern. For example, the following statements all evaluate to `true`:

```
"Java is fun".matches("Java.*")
"Java is cool".matches("Java.*")
"Java is powerful".matches("Java.*")
```

`"Java.*"` in the preceding statements is a regular expression. It describes a string pattern that begins with Java followed by any zero or more characters. Here, the substring `.*` matches any zero or more characters.

H.2 Regular Expression Syntax

A regular expression consists of literal characters and special symbols. Table H.1 lists some frequently used syntax for regular expressions.

Table H.1 Frequently Used Regular Expressions

Regular Expression	*Matches*	*Example*
x	a specified character x	Java matches Java
.	any single character	Java matches J..a
(ab|cd)	ab or cd	ten matches t(en|im)
[abc]	a, b, or c	Java matches Ja[uvwx]a

(*continued*)

Regular Expression	*Matches*	*Example*
[^abc]	any character except a, b, or c	Java matches Ja[^ars]a
[a-z]	a through z	Java matches [A-M]av[a-d]
[^a-z]	any character except a through z	Java matches Jav[^b-d]
[a-e[m-p]]	a through e or m through p	Java matches [A-G[I-M]]av[a-d]
[a-e&&[c-p]]	intersection of a-e with c-p	Java matches [A-P&&[I-M]]av[a-d]
\d	a digit, same as [0-9]	Java2 matches "Java[\\d]"
\D	a non-digit	$Java matches "[\\D][\\D]ava"
\w	a word character	Java1 matches "[\\w]ava[\\d]"
\W	a non-word character	$Java matches "[\\W][\\w]ava"
\s	a whitespace character	"Java 2" matches "Java\\s2"
\S	a non-whitespace char	Java matches "[\\S]ava"
*p**	zero or more occurrences of pattern *p*	aaaa matches "a*" abab matches "(ab)*"
p+	one or more occurrences of pattern *p*	a matches "a+b*" able matches "(ab)+.*"
p?	zero or one occurrence of pattern *p*	Java matches "J?Java" ava matches "J?ava"
p{n}	exactly n occurrences of pattern *p*	Java matches "Ja{1}.*" Java does not match ".{2}"
p{n,}	at least n occurrences of pattern *p*	aaaa matches "a{1,}" a does not match "a{2,}"
p{n,m}	between n and m occurrences (inclusive)	aaaa matches "a{1,9}" abb does not match "a{2,9}bb"
\p{P}	a punctuation character !"#$%&'()*+, -./:;<=>?@ [\]^_'{|}~	J?a matches "J\\p{P}a" J?a. does not match "J\\p{P}a"

Note

Backslash is a special character that starts an escape sequence in a string. So you need to use \\ to represent a literal character \.

Note

Recall that a *whitespace character* is ' ', '\t', '\n', '\r', or '\f'. So \s is the same as [\t\n\r\f], and \S is the same as [^ \t\n\r\f].

Note

A word character is any letter, digit, or the underscore character. So \w is the same as [a-z[A-Z][0-9]_] or simply [a-zA-Z0-9_], and \W is the same as [^a-zA-Z0-9_].

Note

The entries *, +, ?, {n}, {n,}, and {n, m} in Table H.1 are called *quantifiers* that specify how many times the pattern before a quantifier may repeat. For example, A* quantifier matches zero or more **A**'s, **A+** matches one or more **A**'s, **A?** matches zero or one **A**, **A{3}** matches exactly **AAA**, **A{3,}** matches at least three **A**'s, and **A{3,6}** matches between 3 and 6 **A**'s. ***** is the same as **{0,}**, **+** is the same as **{1,}**, and **?** is the same as **{0,1}**.

Caution

Do not use spaces in the repeat quantifiers. For example, **A{3,6}** cannot be written as **A{3, 6}** with a space after the comma.

Note

You may use parentheses to group patterns. For example, **(ab){3}** matches **ababab**, but **ab{3}** matches **abbb**.

Let us use several examples to demonstrate how to construct regular expressions.

Example 1

The pattern for Social Security numbers is **xxx-xx-xxxx**, where **x** is a digit. A regular expression for Social Security numbers can be described as

```
[\\d]{3}-[\\d]{2}-[\\d]{4}
```

For example,

```
"111-22-3333".matches("[\\d]{3}-[\\d]{2}-[\\d]{4}") returns true.
"11-22-3333".matches("[\\d]{3}-[\\d]{2}-[\\d]{4}") returns false.
```

Example 2

An even number ends with digits **0**, **2**, **4**, **6**, or **8**. The pattern for even numbers can be described as

```
[\\d]*[02468]
```

For example,

```
"123".matches("[\\d]*[02468]") returns false.
"122".matches("[\\d]*[02468]") returns true.
```

Example 3

The pattern for telephone numbers is **(xxx) xxx-xxxx**, where **x** is a digit and the first digit cannot be zero. A regular expression for telephone numbers can be described as

```
\\([1-9][\\d]{2}\\) [\\d]{3}-[\\d]{4}
```

Note the parentheses symbols **(** and **)** are special characters in a regular expression for grouping patterns. To represent a literal **(** or **)** in a regular expression, you have to use **\\(** and **\\)**. For example,

```
"(912) 921-2728".matches("\\([1-9][\\d]{2}\\) [\\d]{3}-[\\d]{4}")
returns true.
"921-2728".matches("\\([1-9][\\d]{2}\\) [\\d]{3}-[\\d]{4}") returns
false.
```

Example 4

Suppose the last name consists of at most 25 letters, and the first letter is in uppercase. The pattern for a last name can be described as

```
[A-Z][a-zA-Z]{1,24}
```

Note you cannot have arbitrary whitespace in a regular expression. For example, `[A-Z][a-zA-Z]{1, 24}` would be wrong.
For example,

```
"Smith".matches("[A-Z][a-zA-Z]{1,24}") returns true.
"Jones123".matches("[A-Z][a-zA-Z]{1,24}") returns false.
```

Example 5

Java identifiers are defined in Section 2.3, Identifiers.

- An identifier must start with a letter, an underscore (_), or a dollar sign ($). It cannot start with a digit.
- An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs ($).

The pattern for identifiers can be described as

```
[a-zA-Z_$][\\w$]*
```

Example 6

What strings are matched by the regular expression `"Welcome to (Java|HTML)"`? The answer is `Welcome to Java` or `Welcome to HTML`.

Example 7

What strings are matched by the regular expression `".*"`? The answer is any string.

H.3 Replacing and Splitting Strings

The `matches` method in the `String` class returns `true` if the string matches the regular expression. The `String` class also contains the `replaceAll`, `replaceFirst`, and `split` methods for replacing and splitting strings, as shown in Figure H.1.

java.lang.String	
+matches(regex: String): boolean	Returns true if this string matches the pattern.
+replaceAll(regex: String, replacement: String): String	Returns a new string that replaces all matching substrings with the replacement.
+replaceFirst(regex: String, replacement: String): String	Returns a new string that replaces the first matching substring with the replacement.
+split(regex: String): String[]	Returns an array of strings consisting of the substrings split by the matches.
+split(regex: String, limit: int): String[]	Same as the preceding split method except that the limit parameter controls the number of times the pattern is applied.

FIGURE H.1 The `String` class contains the methods for matching, replacing, and splitting strings using regular expressions.

The `replaceAll` method replaces all matching substring, and the `replaceFirst` method replaces the first matching substring. For example, the code

```
System.out.println("Java Java Java".replaceAll("v\\w", "wi"));
```

displays

```
Jawi Jawi Jawi
```

and this code

```
System.out.println("Java Java Java".replaceFirst("v\\w", "wi"));
```

displays

```
Jawi Java Java
```

There are two overloaded `split` methods. The `split(regex)` method splits a string into substrings delimited by the matches. For example, the statement

```
String[] tokens = "Java1HTML2Perl".split("\\d");
```

splits string `"Java1HTML2Perl"` into `Java`, `HTML`, and `Perl` and saves in `tokens[0]`, `tokens[1]`, and `tokens[2]`.

In the `split(regex, limit)` method, the `limit` parameter determines how many times the pattern is matched. If `limit <= 0`, `split(regex, limit)` is same as `split(regex)`. If `limit > 0`, the pattern is matched at most `limit – 1` times. Here are some examples:

```
"Java1HTML2Perl".split("\\d", 0); splits into Java, HTML, Perl
"Java1HTML2Perl".split("\\d", 1); splits into Java1HTML2Perl
"Java1HTML2Perl".split("\\d", 2); splits into Java, HTML2Perl
"Java1HTML2Perl".split("\\d", 3); splits into Java, HTML, Perl
"Java1HTML2Perl".split("\\d", 4); splits into Java, HTML, Perl
"Java1HTML2Perl".split("\\d", 5); splits into Java, HTML, Perl
```

Note

By default, all the quantifiers are *greedy*. This means that they will match as many occurrences as possible. For example, the following statement displays `JRvaa`, since the first match is `aaa`:

```
System.out.println("Jaaavaa".replaceFirst("a+", "R"));
```

You can change a qualifier's default behavior by appending a question mark (`?`) after it. The quantifier becomes *reluctant* or *lazy*, which means that it will match as few occurrences as possible. For example, the following statement displays `JRaavaa`, since the first match is `a`:

```
System.out.println("Jaaavaa".replaceFirst("a+?", "R"));
```

H.4 Replacing Partial Content in a Matched Substring

Sometimes, you need to make a replacement for partial content in a matched string. For example, suppose we have a text as follows:

```
String text = "3 * (x – y) is in lines 12-56.";
```

We would like to replace the text to

```
"3 * (x – y) is in lines 12 to 56.";
```

Note the `–` symbol is replaced to the word `"to"` if it is preceded by the word `"lines"` and between two numbers. We would like to replace all the occurrences of the `–` symbol by the word `"to"` for such cases in a text. To accomplish this, we will find a matching substring with the pattern `[lines \\d+-\\d+"]` and then replace the `–` symbol in the pattern by the word `"to"`. This can be done using the `Pattern` and `Matcher` classes.

The `Pattern` class is a compiled representation of a regular expression. You can create an instance of `Pattern` using `Pattern.compile(regex)`. The resulting pattern can then be used to create a `Matcher` object. For example, the following code creates a `Pattern` object `p` and creates a `Matcher` object `m` for the text using the pattern `p`:

```
String regex = "lines \\d+-\\d+";
Pattern p = Pattern.compile(regex);
Matcher m = p.matcher(text);
```

You can now use the `find()` method in the `Matcher` class to find a matched substring for the pattern, use the `group()` method to return the matched substring, and replace the `–` symbol in the matched string, and then use the `addReplacement` and `addTail` methods to add the text and its replacement into a `StringBuilder`.

The complete code is given in Listing H.1

LISTING H.1 PatternMatcherDemo.java

```
1  import java.util.regex.Matcher;
2  import java.util.regex.Pattern;
3
4  public class PatternMatcherDemo {
5    public static void main(String args[]) {
6     String text = "3 * (x – y) is in lines 12–56.";
7      String regex = "lines \\d+-\\d+";
8      Pattern p = Pattern.compile(regex);
9      Matcher m = p.matcher(text);
10
11     StringBuffer sb = new StringBuffer();
12     while (m.find()) {
13        String replacement = m.group();
14        replacement = replacement.replace("-", " to ");
15        m.appendReplacement(sb, replacement);
16    }
17
18    m.appendTail(sb);
19    System.out.println(sb.toString());
20  }
26 }
```

This is an elaborated process. Invoking `m.find()` (line 12) scans the text to find the next match for the pattern in the text from the starting position. Initially, the starting position is at index `0`. Invoking `m.group()` (line 13) returns the matched substring to `String replacement`. The `String`'s `replace` method replaces `"-"` with `"to"` (line 14). Invoking `m.appendReplace(sb, replacement)` appends the current unmatched content in the text to `sb` and then appends `replacement` to `sb`, where `sb` is a `StringBuilder`. Note that the current unmatched content are the substring that has been scanned by `m.find()`, but not part of the matched string. The loop (lines 12-16) continues to find the next matching, obtain the matched substring, replace the partial content in the substring, and append the unmatched content and the replacement to `sb`. The loop ends when no more matches can be found. The program then invokes the `m.addTail(sb)` method to append the remaining unmatched content in the text to `sb` (line 18).

Note that all these methods `find()`, `group()`, `addReplacement`, and `addTail` are used together in a find-replace-append loop. When `m.find()` is invoked for the first time, the starting position is at index `0`. When `m.find()` is invoked again, it first resets the starting position to pass the end of the matched substring.

APPENDIX I

Enumerated Types

I.1 Simple Enumerated Types

An enumerated type defines a list of enumerated values. Each value is an identifier. For example, the following statement declares a type, named **MyFavoriteColor**, with values **RED**, **BLUE**, **GREEN**, and **YELLOW** in this order:

```
enum MyFavoriteColor {RED, BLUE, GREEN, YELLOW};
```

A value of an enumerated type is like a constant and so, by convention, is spelled with all uppercase letters. So, the preceding declaration uses **RED**, not **red**. By convention, an enumerated type is named like a class with first letter of each word capitalized.

Once a type is defined, you can declare a variable of that type:

```
MyFavoriteColor color;
```

The variable **color** can hold one of the values defined in the enumerated type **MyFavoriteColor** or **null**, but nothing else. Java enumerated type is *type-safe*, meaning that an attempt to assign a value other than one of the enumerated values or **null** will result in a compile error.

The enumerated values can be accessed using the syntax

```
EnumeratedTypeName.valueName
```

For example, the following statement assigns enumerated value **BLUE** to variable **color**:

```
color = MyFavoriteColor.BLUE;
```

Note you have to use the enumerated type name as a qualifier to reference a value such as **BLUE**.

As with any other type, you can declare and initialize a variable in one statement:

```
MyFavoriteColor color = MyFavoriteColor.BLUE;
```

An enumerated type is treated as a special class. An enumerated type variable is therefore a reference variable. An enumerated type is a subtype of the **Object** class and the **Comparable** interface. Therefore, an enumerated type inherits all the methods in the **Object** class and the **compareTo** method in the **Comparable** interface. Additionally, you can use the following methods on an enumerated object:

- **public String name();**
  Returns a name of the value for the object.
- **public int ordinal();**
  Returns the ordinal value associated with the enumerated value. The first value in an enumerated type has an ordinal value of 0, the second has an ordinal value of 1, the third one 3, and so on.

Listing I.1 gives a program that demonstrates the use of enumerated types.

LISTING I.1 EnumeratedTypeDemo.java

```
 1 public class EnumeratedTypeDemo {
 2   static enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
 3     FRIDAY, SATURDAY};
 4
 5   public static void main(String[] args) {
 6     Day day1 = Day.FRIDAY;
 7     Day day2 = Day.THURSDAY;
 8
 9     System.out.println("day1's name is " + day1.name());
10     System.out.println("day2's name is " + day2.name());
11     System.out.println("day1's ordinal is " + day1.ordinal());
12     System.out.println("day2's ordinal is " + day2.ordinal());
13
14     System.out.println("day1.equals(day2) returns " +
15       day1.equals(day2));
16     System.out.println("day1.toString() returns " +
17       day1.toString());
18     System.out.println("day1.compareTo(day2) returns " +
19       day1.compareTo(day2));
20   }
21 }
```

define an enum type

declare an enum variable

get enum name

get enum ordinal

compare enum values

```
day1's name is FRIDAY
day2's name is THURSDAY
day1's ordinal is 5
day2's ordinal is 4
day1.equals(day2) returns false
day1.toString() returns FRIDAY
day1.compareTo(day2) returns 1
```

An enumerated type **Day** is defined in lines 2 and 3. Variables **day1** and **day2** are declared as the **Day** type and assigned enumerated values in lines 6 and 7. Since **day1**'s value is **FRIDAY**, its ordinal value is 5 (line 11). Since **day2**'s value is **THURSDAY**, its ordinal value is 4 (line 12).

Since an enumerated type is a subclass of the **Object** class and the **Comparable** interface, you can invoke the methods **equals**, **toString**, and **compareTo** from an enumerated object reference variable (lines 14–19). **day1.equals(day2)** returns true if **day1** and **day2** have the same ordinal value. **day1.compareTo(day2)** returns the difference between **day1**'s ordinal value and **day2**'s.

Alternatively, you can rewrite the code in Listing I.1 into Listing I.2.

LISTING I.2 StandaloneEnumTypeDemo.java

```
public class StandaloneEnumTypeDemo {
  public static void main(String[] args) {
    Day day1 = Day.FRIDAY;
    Day day2 = Day.THURSDAY;

    System.out.println("day1's name is " + day1.name());
    System.out.println("day2's name is " + day2.name());
    System.out.println("day1's ordinal is " + day1.ordinal());
```

```
9        System.out.println("day2's ordinal is " + day2.ordinal());
10
11       System.out.println("day1.equals(day2) returns " +
12         day1.equals(day2));
13       System.out.println("day1.toString() returns " +
14         day1.toString());
15       System.out.println("day1.compareTo(day2) returns " +
16         day1.compareTo(day2));
17     }
18   }
19
20   enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
21     FRIDAY, SATURDAY}
```

An enumerated type can be defined inside a class, as shown in lines 2 and 3 in Listing I.1, or standalone as shown in lines 20 and 21 in Listing I.2. In the former case, the type is treated as an inner class. After the program is compiled, a class named EnumeratedTypeDemo$Day.class is created. In the latter case, the type is treated as a stand-alone class. After the program is compiled, a class named Day.class is created.

Note

When an enumerated type is declared inside a class, the type must be declared as a member of the class and cannot be declared inside a method. Furthermore, the type is always **static**. For this reason, the **static** keyword in line 2 in Listing I.1 may be omitted. The visibility modifiers on inner class can be also be applied to enumerated types defined inside a class.

Tip

Using enumerated values (e.g., **Day.MONDAY**, **Day.TUESDAY**, and so on) rather than literal integer values (e.g., 0, 1, and so on) can make the program easier to read and maintain.

I.2 Using if or switch Statements with an Enumerated Variable

An enumerated variable holds a value. Often, your program needs to perform a specific action depending on the value. For example, if the value is **Day.MONDAY**, play soccer; if the value is **Day.TUESDAY**, take piano lesson, and so on. You can use an **if** statement or a **switch** statement to test the value in the variable, as shown in (a) and (b).

```
if (day.equals(Day.MONDAY)) {
  // process Monday
}
else if (day.equals(Day.TUESDAY)) {
  // process Tuesday
}
else
  ...
```

(a)

Equivalent

```
switch (day) {
  case MONDAY:
    // process Monday
    break;
  case TUESDAY:
    // process Tuesday
    break;
  ...
}
```

(b)

In the **switch** statement in (b), the case label is an unqualified enumerated value (e.g., **MONDAY**, but not **Day.MONDAY**).

I.3 Processing Enumerated Values Using a Foreach Loop

Each enumerated type has a static method `values()` that returns all enumerated values for the type in an array. For example,

```
Day[] days = Day.values();
```

You can use a regular for loop in (a) or foreach loop in (b) to process all the values in the array.

```
for (int i = 0; i < days.length; i++)
  System.out.println(days[i]);
```

(a)

Equivalent

```
for (Day day: days)
  System.out.println(day);
```

(b)

I.4 Enumerated Types with Data Fields, Constructors, and Methods

The simple enumerated types introduced in the preceding section define a type with a list of enumerated values. You can also define an enumerate type with data fields, constructors, and methods, as shown in Listing I.3.

Listing I.3 TrafficLight.java

```
1  public enum TrafficLight {
2    RED ("Please stop"), GREEN ("Please go"),
3    YELLOW ("Please caution");
4
5    private String description;
6
7    private TrafficLight(String description) {
8      this.description = description;
9    }
10
11   public String getDescription() {
12     return description;
13   }
14 }
```

The enumerated values are defined in lines 2 and 3. The value declaration must be the first statement in the type declaration. A data field named `description` is declared in line 5 to describe an enumerated value. The constructor `TrafficLight` is declared in lines 7–9. The constructor is invoked whenever an enumerated value is accessed. The enumerated value's argument is passed to the constructor, which is then assigned to `description`.

Listing I.4 gives a test program to use `TrafficLight`.

Listing I.4 TestTrafficLight.java

```
public class TestTrafficLight {
  public static void main(String[] args) {
    TrafficLight light = TrafficLight.RED;
    System.out.println(light.getDescription());
  }
}
```

An enumerated value `TrafficLight.red` is assigned to variable `light` (line 3). Accessing `TrafficLight.RED` causes the JVM to invoke the constructor with argument “please stop”. The methods in enumerated type are invoked in the same way as the methods in a class. `light.getDescription()` returns the description for the enumerated value (line 4).

Note

The Java syntax requires that the constructor for enumerated types be private to prevent it from being invoked directly. The private modifier may be omitted. In this case, it is considered private by default.

APPENDIX J

The Big-O, Big-Omega, and Big-Theta Notations

Chapter 22 presented the Big-O notation in laymen's term. In this appendix, we give a precise mathematic definition for the Big-O notation. We will also present the Big-Omega and Big-Theta notations.

J.1 The Big-O Notation

The Big-O notation is an asymptotic notation that describes the behavior of a function when its argument approaches a particular value or infinity. Let $f(n)$ and $g(n)$ be two functions, we say that $f(n)$ is $O(g(n))$, pronounced "big-O of $g(n)$", if there is a constant c $(c > 0)$ and value m such that $f(n) \leq c \times g(n)$, for $n \geq m$.

For example, $f(n) = 5n^3 + 8n^2$ is $O(n^3)$, because you can find $c = 13$ and $m = 1$ such that $f(n) \leq cn^3$ for $n \geq m$. $f(n) = 6n\log n + n^2$ is $O(n^2)$, because you can find $c = 7$ and $m = 2$ such that $f(n) \leq cn^2$ for $n \geq m$. $f(n) = 6n\log n + 400n$ is $O(n\log n)$, because you can find $c = 406$ and $m = 2$ such that $f(n) \leq cn\log n$ for $n \geq m$. $f(n) = n^2$ is $O(n^3)$, because you can find $c = 1$ and $m = 1$ such that $f(n) \leq cn^3$ for $n \geq m$. Note that there are infinite number of choices of c and m such that $f(n) \leq c \times g(n)$ for $n \geq m$.

The Big-O notation denotes that a function $f(n)$ is asymptotically less than or equal to another function $g(n)$. This allows you to simplify the function by ignoring multiplicative constants and discarding the non-dominating terms in the function.

J.2 The Big-Omega Notation

The Big-Omega notation is the opposite of the Big-O notation. It is an asymptotic notation that denotes that a function $f(n)$ is greater than or equal to another function $g(n)$. Let $f(n)$ and $g(n)$ be two functions, we say that $f(n)$ is $\Omega(g(n))$, pronounced "big-Omega of $g(n)$", if there is a constant $c(c > 0)$ and value m such that $f(n) \geq c \times g(n)$, for $n \geq m$.

For example, $f(n) = 5\mathrm{n}^3 + 8\mathrm{n}^2$ is $\Omega(\mathrm{n}^3)$, because you can find $c = 5$ and $m = 1$ such that $f(n) \geq cn^3$ for $n \geq m$. $f(n) = 6n\log n + n^2$ is $\Omega(n^2)$, because you can find $c = 1$ and $m = 1$ such that $f(n) \geq cn^2$ for $n \geq m$. $f(n) = 6n\log n + 400n$ is $\Omega(n\log n)$, because you can find $c = 6$ and $m = 1$ such that $f(n) \geq cn\log n$ for $n \geq m$. $f(n) = n^2$ is $\Omega(n)$, because you can find $c = 1$ and $m = 1$ such that $f(n) \geq cn$ for $n \geq m$. Note that there are infinite number of choices of c and m such that $f(n) \geq c \times g(n)$ for $n \geq m$.

J.3 The Big-Theta Notation

The Big-Theta notation denotes that two functions are the same asymptotically. Let $f(n)$ and $g(n)$ be two functions, we say that $f(n)$ is $\Theta(g(n))$, pronounced "big-Theta of $g(n)$", if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

For example, $f(n) = 5n^3 + 8n^2$ is $\Theta(n^3)$, because you $f(n)$ is $O(n^3)$ and $f(n)$ is $\Omega(n^3)$. $f(n) = 6n\log n + 400n$ is $\Theta(n\log n)$, because $f(n)$ is $O(n\log n)$ and $f(n)$ is $\Theta(n\log n)$.

Note

The Big-O notation gives an upper bound of a function. The Big-Omega notation gives a lower bound of a function. The Big-Theta notation gives a tight bound of a function. For simplicity, the Big-O notation is often used, even though the Big-Theta notation may be more factually appropriate.

Java Quick Reference

Console Input

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
long longValue = input.nextLong();
double doubleValue = input.nextDouble();
float floatValue = input.nextFloat();
String string = input.next();
String line = input.nextLine();
```

Console Output

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

Conditional Expression

```
boolean-expression ? expression1 :
  expression2

y = (x > 0) ? 1 : -1

System.out.println(number % 2 == 0 ?
  "number is even" : "number is odd");
```

Primitive Data Types

byte	8 bits
short	16 bits
int	32 bits
long	64 bits
float	32 bits
double	64 bits
char	16 bits
boolean	true/false

Arithmetic Operators

+	addition
-	subtraction
*****	multiplication
/	division
%	remainder
++var	preincrement
--var	predecrement
var++	postincrement
var--	postdecrement

Assignment Operators

=	assignment
+=	addition assignment
-=	subtraction assignment
***=**	multiplication assignment
/=	division assignment
%=	remainder assignment

Relational Operators

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal

Logical Operators

&&	short circuit AND
||	short circuit OR
!	NOT
^	exclusive OR

if Statements

```
if (condition) {
  statements;
}

if (condition) {
  statements;
}
else {
  statements;
}

if (condition1) {
  statements;
}
else if (condition2) {
  statements;
}
else {
  statements;
}
```

switch Statements

```
switch (intExpression) {
  case value1:
    statements;
    break;
  ...
  case valuen:
    statements;
    break;
  default:
    statements;
}
```

loop Statements

```
while (condition) {
  statements;
}

do {
  statements;
} while (condition);

for (init; condition;
  adjustment) {
  statements;
}
```

Companion Website: www.pearsonhighered.com/liang

Java Quick Reference

Frequently Used Static Constants/Methods

```
Math.PI
Math.random()
Math.pow(a, b)
Math.abs(a)
Math.max(a, b)
Math.min(a, b)
Math.sqrt(a)
Math.sin(radians)
Math.asin(a)
Math.toRadians(degrees)
Math.toDegress(radians)
System.currentTimeMillis()
Integer.parseInt(string)
Integer.parseInt(string, radix)
Double.parseDouble(string)
Arrays.sort(type[] list)
Arrays.binarySearch(type[] list, type key)
```

Array/Length/Initializer

```
int[] list = new int[10];
list.length;
int[] list = {1, 2, 3, 4};
```

Multidimensional Array/Length/Initializer

```
int[][] list = new int[10][10];
list.length;
list[0].length;
int[][] list = {{1, 2}, {3, 4}};
```

Ragged Array

```
int[][] m = {{1, 2, 3, 4},
             {1, 2, 3},
             {1, 2},
             {1}};
```

Text File Output

```
PrintWriter output =
  new PrintWriter(filename);
output.print(...);
output.println(...);
output.printf(...);
```

Text File Input

```
Scanner input = new Scanner(
  new File(filename));
```

File Class

```
File file =
  new File(filename);
file.exists()
file.renameTo(File)
file.delete()
```

Object Class

```
Object o = new Object();
o.toString();
o.equals(o1);
```

Comparable Interface

```
c.compareTo(Comparable)
c is a Comparable object
```

String Class

```
String s = "Welcome";
String s = new String(char[]);
int length = s.length();
char ch = s.charAt(index);
int d = s.compareTo(s1);
boolean b = s.equals(s1);
boolean b = s.startsWith(s1);
boolean b = s.endsWith(s1);
boolean b = s.contains(s1);
String s1 = s.trim();
String s1 = s.toUpperCase();
String s1 = s.toLowerCase();
int index = s.indexOf(ch);
int index = s.lastIndexOf(ch);
String s1 = s.substring(ch);
String s1 = s.substring(i,j);
char[] chs = s.toCharArray();
boolean b = s.matches(regex);
String s1 = s.replaceAll(regex,repl);
String[] tokens = s.split(regex);
```

ArrayList Class

```
ArrayList<E> list = new ArrayList<>();
list.add(object);
list.add(index, object);
list.clear();
Object o = list.get(index);
boolean b = list.isEmpty();
boolean b = list.contains(object);
int i = list.size();
list.remove(index);
list.set(index, object);
int i = list.indexOf(object);
int i = list.lastIndexOf(object);
```

printf Method

```
System.out.printf("%b %c %d %f %e %s",
  true, 'A', 45, 45.5, 45.5, "Welcome");
System.out.printf("%-5d %10.2f %10.2e %8s",
  45, 45.5, 45.5, "Welcome");
```

Companion Website: www.pearsonhighered.com/liang

INDEX

Symbols
`--` (decrement operator), 57–58
`-` (subtraction operator), 46, 53
`.` (dot operator), 333
`.` (object member access operator), 333, 431
`/` (division operator), 46, 53
`//`, in line comment syntax, 18
`/*`, in block comment syntax, 18
`/**.*/` (Javadoc comment syntax), 18
`/=` (division assignment operator), 56–57
`;` (semicolons), common errors, 86
`\` (backslash character), as directory separator, 478
`\` (escape characters), 128
`||` (or logical operator), 95–99
`+` (addition operator), 46, 53
`+` (string concatenation operator), 36, 133
`++` (increment operator), 57–58
`+=` (addition assignment operator), augmented, 56–57
`=` (assignment operator), 42–43, 56–57
`=` (equals operator), 78
`-=` (subtraction assignment operator), 56–57
`==` (comparison operator), 78, 434
`=` (equal to operator), 78
`!` (not logical operator), 95–99
`!=` (not equal to comparison operator), 78
`$` (dollar sign character), use in source code, 40
`%` (remainder or modulo operator), 46, 53
`%=` (remainder assignment operator), 56–57
`&&` (and logical operator), 95–99
`( )` (parentheses), 13, 227
`*` (multiplication operator), 14, 46, 53
`*=` (multiplication assignment operator), 56
`^` (exclusive or logical operator), 95–99
`{}` (curly braces), 13, 81, 85
`<` (less than comparison operator), 78
`<=` (less than or equal to comparison operator), 78
`>` (greater than comparison operator), 78
`>=` (greater than or equal to comparison operator), 78

Numbers
24-point game, 811–812

A

`abs` method, `Math` class, 124, 530
Absolute file name, 477
Abstract classes
  `AbstractCollection` class, 776, 777
  `AbstractMap` class, 830
  `AbstractSet` class, 816
  case study: abstract number class, 505–507
  case study: `Calendar` and `GregorianCalendar` classes, 507–510
  characteristics of, 504–505
  `Circle.java` and `Rectangle.java` examples, 502
  compared with interfaces, 523–526
  `GeometricObject.java` example, 500–502
  `InputStream` and `OutputStream` classes, 694–695
  interfaces compared to, 510
  key terms, 534
  overview of, 368–369, 500–501
  questions and excercises, 535–540
  `Rational.java` example, 528–531
  reasons for using abstract methods, 502
  summary, 534–535
  `TestCalendar.java` example, 508–510
  `TestGeometricObject.java` example, 502–503
  `TestRationalClass.java` example, 527–528
  using as interface, 924
Abstract data type (ADT), 368
Abstract methods
  characteristics of, 504
  `GenericMatrix.java` example, 766–769
  `GeometricObject` class, 501–502
  implementing in subclasses, 501–502
  in interfaces, 510
  key terms, 534
  in `Number` class, 530
  overview of, 227–228
  questions and exercises, 535–540
  reasons for using, 502
  summary, 534–535
`abstract` modifier, for denoting abstract methods, 500
Abstract number class
  `LargestNumbers.java`, 506–507
  overview of, 505–507
Abstract Windows Toolkit. *see* AWT (Abstract Windows Toolkit)
`AbstractCollection` class, 777
`AbstractMap` class, 830
`AbstractSet` class, 816
Accessibility modifiers, 1169
Accessor methods. *see* Getter (accessor) methods
`acos` method, trigonometry, 122–123
`ActionEvent`, 595–596
Actions (behaviors), object, 324
Activation records, invoking methods and, 210
Actual concrete types, 752
Actual parameters, defining methods and, 207
Ada, high-level languages, 8
`add` method
  implementing linked lists, 935
  `List` interface, 783
Addition (`+=`) assignment operator, 56–57
Addition (`+`) operator, 46, 53
Adelson-Velsky, G. M., 996
Adjacency lists, representing edges, 1052–1054
Adjacency matrices
  representing edges, 1052–1054
  weighted, 1054
Adjacent edges, overview of, 1048
ADT (Abstract data type), 368
Aggregate operations, for collection streams
  `AnalyzeNumbersUsingStream.java` example, 1150–1151
  case study: analyzing numbers, 1150–1151
  case study: counting keywords, 1155–1156
  case study: counting occurrences of each letter, 1151–1152
  case study: counting occurrences of each letter in string, 1152–1153
  case study: finding directory size, 1154–1155
  case study: occurrences of words, 1157–1158
  case study: processing all elements in two-dimensional array, 1153–1154

Aggregate operations, for collection streams (*Continued*)
  CollectDemo.java example, 1145–1147
  CollectGroupDemo.java example, 1148–1150
  CountKeywordStream.java example, 1155–1156
  CountLettersUsingStream.java example, 1151–1152
  CountOccurrenceOfLettersInAString.java example, 1152–1153
  CountOccurrenceOfWordsStream.java example, 1157–1158
  DirectorySizeStream.java example, 1154–1155
  DoubleStream, 1136–1139
  grouping elements using groupingby collector, 1147–1150
  IntStream, 1136–1139
  IntStreamDemo.java example, 1136–1139
  LongStream, 1136–1139
  overview of, 1130
  parallel streams, 1139–1141
  ParallelStreamDemo.java example, 1139–1141
  quiz and exercises, 1158–1159
  Stream class, 1131
  stream pipelines, 1130–1136
  stream reduction using collect method, 1144–1147
  stream reduction using reduce method, 1141–1144
  StreamDemo.java example, 1132–1133
  StreamReductionDemo.java example, 1142–1144
  summary, 1158
  TwoDimensionalArrayStream.java example, 1153–1154
Aggregating classes, 376
Aggregating objects, 376
Aggregation relationships, objects, 376–377
AIFF audio files, 676
Algorithms, 34
  analyzing Towers of Hanoi problem, 846–847
  Big *O* notation for measuring efficiency of, 840–841
  binary search, 846
  Boyer-Moore algorithm, 870–873
  bubble sort, 890–892
  comparing growth functions, 847–848
  comparing prime numbers, 861
  determining Big *O* for repetition, sequence, and selection statements, 842–845
  EfficientPrimeNumbers.java example, 857–859
  external sorts. *see* External sorts
  finding closest pair of points, 861–864
  finding convex hull for a set of points, 867–869
  finding Fibonacci numbers, 849–851
  finding greatest common denominator, 851–855
  finding prime numbers, 855–861
  GCDEuclid.java example, 853–855
  GCD.java example, 852–853
  gift-wrapping algorithm, 867–868
  Graham's algorithm, 868–869
  graph algorithms, 1046–1047
  greedy, 986
  heap sort. *see* Heap sorts
  insertion sorts, 888–890
  key terms, 876
  Knuth-Morris-Pratt algorithm, 873–876
  merge sorts, 892–896
  overview of, 840
  PrimeNumbers.java example, 856–857
  quick sort, 896–900
  quiz and exercises, 878–886
  recurrence relations and, 847
  selection sort and insertion sort, 846
  SieveOfEratosthenes.java example, 860–861
  solving Eight Queens problem, 864–867
  for sort method, 759
  summary, 876–877
Algorithms, spanning tree
  Dijkstra's single-source shortest-path algorithm, 1106–1111
  MST algorithm, 1103–1105
  Prim's minimum spanning tree algorithm, 1103–1105
allMatch method, 1132, 1134
Ambiguous invocation, of methods, 223
American Standard Code for Information Interchange (ASCII). *see* ASCII (American Standard Code for Information Interchange)
And (&&) logical operator, 95–99
Animation
  case study: bouncing ball, 626–629
  case study: US map, 630–633
  ClockAnimation.java, 625–626
  FadeTransition, 622–623
  key terms, 633
  PathTransition, 619–622
  programming exercises, 634–641
  quiz, 634
  summary, 633–634
  Timeline, 624–625
Anonymous arrays, 262
Anonymous objects, 333
AnonymousHandlerDemo.java, 603–605
anyMatch method, 1132, 1134
Application Program Interfaces (APIs), 11
Apps, developing on Web servers, 11
Arc
  overview, 575
  ShowArc.java, 575–577
Arguments
  defining methods and, 207
  passing by values, 214–217
  receiving string arguments from command line, 276–279
  variable-length argument lists, 268–269
ArithmeticException class, 457
Arithmetic/logic units, CPU components, 3
Array elements, 252
Array initializers, 252–253
arraycopy method, System class, 260
ArrayIndexOutOfBoundsException, 255
ArrayList class
  animation of array lists, 925
  case study: custom stack class, 441–442
  cloning arrays, 519
  compared with LinkedList, 784–787
  creating and adding numbers to array lists, 434–440
  creating array lists, 778–780
  defined under List interface, 783
  DistinctNumbers.java example, 438–440
  as example of generic class, 752–753
  heap sorts, 901
  implementing array lists, 928–935
  implementing bucket sorts, 908
  implementing buckets, 1023
  implementing stacks using array lists. *see* Stacks
  MyArrayList, 924–925, 947
  MyArrayList compared with MyLinkedList, 947
  MyArrayList.java example, 929–933
  MyList.java example, 926–928
  representing edges in graphs, 1053
  SetListPerformanceTest.java example, 825–826
  storing edge objects in, 1051
  for storing elements in a list, 776
  storing heaps in, 901
  storing list of objects in, 434–435
  TestArrayAndLinkedList.java, 785–787
  TestArrayList.java example, 436–438

TestMyArrayList.java example, 933–935
Vector class compared with, 799
Arrays class, 274–276
Arrays, in general
edge arrays, 1051
as fixed-size data structure, 928
implementing binary heaps using, 901
ragged arrays, 1052
sorting using Heap class, 906
storing lists in. *see* ArrayList class
storing vertices in, 1050
Arrays, multi-dimensional
case study: daily temperature and humidity, 304–306
case study: guessing birthdays, 306–307
overview of, 303–304
questions and exercises, 308–321
summary, 307
Arrays, single-dimensional
accessing elements, 252
ArrayList class, 437–438
Arrays class, 274–276
case study: analyzing numbers, 257–258
case study: counting occurrences of letters, 265–268
case study: deck of cards, 258–260
case study: generic method for sorting, 758–759
constructing strings from, 388
converting strings to/from, 392
copying, 260–261
creating, 251–252, 514–516
declaring, 250
foreach loops, 255–257
initializers, 252–253
key terms, 279
of objects, 353–355
overview of, 250
passing to methods, 261–264
processing, 253–255
questions and exercises, 280–288
returning from methods, 264–265
searching, 269–273
serializing, 709–711
size and default values, 252
sorting, 273–274, 514–516
summary, 279–280
treating as objects in Java, 333
variable-length argument lists, 268–269
Arrays, two-dimensional
case study: finding closest pair of points, 298–300
case study: grading multiple-choice test, 296–298
case study: processing all elements in a two-dimensional array, 1153–1154
case study: Sudoku, 300–303
declaring variables and creating two-dimensional arrays, 290–291
defined, 290
obtaining length of two-dimensional arrays, 291–292
passing to methods to two-dimensional arrays, 295–296
processing two-dimensional arrays, 293–295
questions and exercises, 308–321
ragged arrays, 292–293
representing graph edges with, 1051
representing weighted graphs, 1093–1095
summary, 307
Arrows keys, on keyboards, 6
ASCII (American Standard Code for Information Interchange)
character (char data type), 126–127
decimal and hexadecimal equivalents, 1165
encoding scheme, 3–4
text encoding, 692
text I/O *vs.* binary I/O, 693–694
asin method, trigonometry, 123
asList method, 786
Assemblers, 7
Assembly language, 7
Assignment operator (=), 1175
augmented, 56–57
overview of, 42–43
Assignment statements (assignment expressions)
assigning value to variables, 36
overview of, 42–43
Associative array, 1016
Associative arrays. *see* Maps
Associativity, of operators, 107, 1166–1167
atan method, trigonometry, 122–123
Attributes, object, 324
Audio files
case study: national flags and anthems, 679–681
MediaDemo.java, 677–679
Autoboxing/Autounboxing, 386, 753–754
Average-case analysis
measuring algorithm efficiency, 840, 854
quick sort and, 899–900
AVL trees
AVLTree.java example, 1002–1007
balancing nodes on a path, 1000–1001
deleting elements, 1002
designing classes for, 999–1000
key terms, 1012
overriding the insert method, 1000–1001
overview of, 996
questions and exercises, 1012–1013
rebalancing, 996–998
rotations for balancing, 1001
summary, 1012
TestAVLTree.java example, 1008–1011
time complexity of, 1011
AVLTree class
delete method, 1007, 1011
overview of, 1002–1007
as subclass of BST class, 999
testing, 1008–1011
AWT (Abstract Windows Toolkit)
Color class, 553–554
Date class, 336–337, 507–508
Error class, 460, 462
event classes in, 596
EventObject class, 596–597
exceptions. *see* Exception class
File class, 477–479, 692
Font class, 554–555
GeometricObject class, 501–502
GuessDate class, 306–307
IllegalArgumentException class, 463
InputMismatchException class, 458, 484
KeyEvent class, 613
MalformedURLException class, 487–488
MouseEvent class, 611–612
Polygon class, 577–578
String class, 388
Swing *vs.*, 542

B

Babylonian method, 242
Backslash character (\), as directory separator, 478
Backtracking algorithm, 864–867

Backward pointer, in doubly linked lists, 948
Balance factor, for AVL nodes, 996, 1004
Balanced nodes
  in AVL trees, 996
  AVLTree class, 1002–1003, 1006–1007
Base cases, in recursion, 726
BaseStream interface, 1130
BASIC, high-level languages, 8
Bean machine game, 288, 640
beginIndex method, for obtaining substrings from strings, 137
Behaviors (actions), object, 324
Behind the scene evaluation, expressions, 107
Best-case input
  measuring algorithm efficiency, 840, 854
  quick sort and, 899–900
BFS (breadth-first searches). *see* Breadth-first searches (BFS)
Big *O*
  determining for repetition, sequence, and selection statements, 842–845
  for measuring algorithm efficiency, 840–842
BigDecimal class, 387–388, 505
Binary
  files, 692
  machine language as binary code, 7
  operator, 48
  searches, 270–273, 730
Binary digits (Bits), 3
Binary heaps (binary trees), 900. *see also* Heap sorts
  complete, 900, 906
Binary I/O
  BufferedInputStream and BufferedOutputStream classes, 701–704
  characters and strings in, 698–699
  classes, 694–704
  DataInputStream and DataOutputStream classes, 698–701
  DetectEndOfFile.java, 701
  FileInputStream and FileOutputStream classes, 695–698
  FilterInputStream and FilterOutputStream classes, 698
  overview of, 692
  TestDataStream.java, 699–700
  TestFileStream.java, 696–698
  vs. text I/O, 693–694
Binary numbers
  converting to/from decimal, 745, 1172
  converting to/from hexadecimal, 1173–1174
  overview of, 1171
Binary search algorithm, 883
  analyzing, 846
  recurrence relations and, 847
Binary search trees (BST)
  BST class, 965–974
  BSTAnimation.java example, 980–981
  BST.java example, 968–973
  BTView.java example, 981–983
  case study: data compression, 985–990
  deleting elements, 974–980
  displaying/visualizing binary trees, 980–983
  HuffmanCode.java example, 987–990
  implementing using linked structure, 960–961
  inserting elements, 962–963
  iterators, 983–985
  key terms, 990
  overview of, 960
  quiz and exercises, 990–994
  representation of, 961–962
  searching for elements, 962
  summary, 990
  TestBSTDelete.java example, 977–980
  TestBST.java example, 973–974
  TestBSTWithIterator.java example, 984–985
  tree traversal, 963–964
  tree visualization and MVC, 980–983
  Tree.java example, 965–967
Binary trees, 960
binarySearch method
  applying to lists, 792–793
  Arrays class, 275
Binding properties
  BindingDemo.java, 550–551
  ShowCircleCentered.java, 548–550
Bit operators, 1175
Bits (binary digits), 3
Bitwise operators, 1175
Block comments, in Welcome.java, 13
Block modifiers, 1168–1169
Block style, programming style, 19
Blocks, in Welcome.java, 13
BMI (Body Mass Index), 91–92, 372–375
Boolean accessor method, 347
boolean data type
  java.util.Random, 337
  overview of, 78–80
Boolean expressions
  case study: determining leap year, 99–100
  conditional operators, 105–106
  defined, 78
  if statements and, 80–81
  if-else statements, 82–83
  writing, 88–89
Boolean literals, 79
Boolean values
  defined, 78
  as format specifier, 147
  logical operators and, 95
  redundancy in testing, 86
Boolean variables
  assigning, 88
  overview of, 78–79
  redundancy in testing, 86
BorderPane
  overview of, 563
  ShowBorderPane.java, 563–564
Bottom-up implementation, 229–231
Bounded generic types
  erasing, 764–765
  GenericMatrix.java example, 767–769
  MaxUsingGenericType.java example, 760–761
  overview of, 757
Bounded wildcards, 762
Boxing, converting wrapper object to primitive value, 386
Boyer-Moore algorithm, 870–873
  StringMatchBoyerMoore.java, 872
Braces. *see* Curly braces ({})
Breadth-first searches (BFS)
  applications of, 1077
  finding BFS trees, 1048
  implementing, 1075–1076
  overview of, 1074
  TestBFS.java, 1076
  traversing graphs, 1067
Breadth-first traversal, tree traversal, 964
break statements
  controlling loops, 186–189
  using with switch statements, 102, 103

Breakpoints, setting for debugging, 108
Brute-force algorithm, 851
BST (binary search trees). *see* Binary search trees (BST)
BST class
  AVLTree class as subclass of, 999
  BST.java example, 968–973
  overview of, 965–966
  TestBSTDelete.java example, 977–980
  TestBST.java example, 973–974
  time complexity of, 978
  Tree.java example, 966–967
Bubble sorts, 283
  algorithms, 891
  BubbleSort.java example, 891–892
  overview of, 890–891
  time complexity of, 892
Buckets
  bucket sorts, 907–909
  separate chaining and, 1023–1024, 1042
BufferedInputStream and BufferedOutputStream classes, 701–704
Bugs (logic errors), 21, 108
Bus, function of, 2–3
Button, 646–648
Button, ButtonDemo.java, 647–648
ButtonBase, 646–647
ButtonDemo.java, 647–648
byte type, numeric types
  hash codes for primitive types, 1017
  overview of, 45
Bytecode
  translating Java source file into, 15
  verifier, 17–18
Bytes
  defined, 3
  measuring storage capacity in, 4

C

C++, high-level languages, 8
C, high-level languages, 8
Cable modems, 6
Calendar class, 507–508
Call stacks
  displaying in debugging, 108
  invoking methods and, 210
Calling
  methods, 208–210
  objects, 333
canRead method, File class, 478–479
canWrite method, File class, 478–479
capacity() method, StringBuilder class, 397
Case sensitivity
  identifiers and, 40
  in Welcome.java, 12
Casting. *see* Type casting
Casting objects
  CastingDemo.java example, 430–431
  overview of, 429–430
Catching exceptions. *see also* try-catch blocks
  catch block omitted when finally clause is used, 471
  CircleWithException.java example, 467
  InputMismatchExceptionDemo.java example, 458–459
  overview of, 463–465
  QuotientWithException.java example, 456–458
CDs (compact discs), as storage device, 5
Cells
  in Sudoku grid, 300
  in tic-tac-toe case study, 671–676
Celsius, converting Fahrenheit to/from, 237–238
Chained exceptions, 473–474
char data type. *see* Characters (char data type)
Characters (char data type)
  applying numeric operators to, 226
  in binary I/O, 698–699
  case study: ignoring nonalphanumeric characters when checking palindromes, 398–400
  casting to/from numeric types, 128–129
  comparing, 78
  comparing and testing, 129–131
  constructing strings from arrays of, 388–389
  converting to strings, 392
  decimal and hexadecimal equivalents of ASCII character set, 1165
  escape characters, 128
  finding, 137–138
  generic method for sorting array of Comparable objects, 758
  hash codes for primitive types, 1017
  overview of, 126
  RandomCharacter.java, 226
  retrieving in strings, 132
  TestRandomCharacter.java, 226–227
  Unicode and ASCII and, 127
charAt (index) method
  retrieving characters in strings, 132
  StringBuilder class, 397–398
CheckBox, 648–651
CheckBoxDemo.java, 649–651
Checked exceptions, 461
checkIndex method, 935
Checkpoint Questions, recurrence relations and, 847
Child, in BST, 961–962
Choice lists. *see* ComboBox
Circle and Ellipse
  overview, 572–573
  ShowEllipse.java, 573–574
Circle class, 324, 325
Circular
  doubly linked lists, 948
  singly linked lists, 948
Clarity, class design guidelines, 532
Class diagrams, UML, 325
Class loaders, 17
Class modifiers, Java modifiers, 1168–1169
ClassCastException, 430
Classes
  abstract. *see* Abstract classes
  abstraction and encapsulation in, 368–372
  benefits of generics, 752–754
  case study: designing class for matrix using generic types, 766–771
  case study: designing class for stacks, 380–382
  case study: designing Course class, 378–380
  in Circle.java (for CircleWithPrivateDataFields), 347–348
  in Circle.java (for CircleWithStaticMembers) example, 340–341
  clients of, 327
  commenting, 18
  in ComputeExpression.java, 14–15
  data field encapsulation for maintaining, 346–347
  defining custom exception classes, 474–477
  defining for objects, 324–326
  defining generic, 754–756
  design guidelines, 531–534
  identifiers, 40
  inner (nested) classes. *see* Inner (nested) classes
  from Java Library, 336–339
  names/naming conventions, 12, 44

Classes (*Continued*)
Point2D, 338–339
preventing extension of, 445
raw types and backward compatibility, 760–761
static variables, constants, and methods, 339–344
in TestCircleWithPrivateDataFields.java example, 348–349
in TestCircleWithStaticMembers.java example, 341–344
in UML diagram, 325, 326
variable scope and, 339–340
visibility modifiers, 344–346
in Welcome.java, 12
in WelcomeWithThreeMessages.java, 14
Classes, binary I/O
BufferedInputStream and BufferedOutputStream classes, 701–704
DataInputStream and DataOutputStream classes, 698–701
DetectEndOfFile.java, 701
FileInputStream and FileOutputStream classes, 695–698
FilterInputStream and FilterOutputStream classes, 698
overview of, 694–695
TestDataStream.java, 699–700
TestFileStream.java, 696–698
Classifier, 1147
Class's contract, 368
Clock speed, CPUs, 3
ClockPane Class
ClockPane.java, 582–584
DisplayClock.java, 581–582
paintClock method, 583–584
clone method, shallow and deep copies, 520–521
Cloneable interface
House.java example, 519–523
overview, 518–519
Closest pair problem, two-dimensional array applied to, 298–300
Closest-pair animation, 883
Cloud storage, 5
Cluster, 1020
COBOL, high-level languages, 8
Code
arrays for simplifying, 254–255
comments and, 103
incremental development, 164
programming. *see* Programs/programming
reuse. *see* Reusable code
sharing. *see* Sharing code
in software development process, 61
Coding trees, 985–990. *see also* Huffman coding trees
Coherent purpose, class design guidelines, 531–532
collect method, stream reduction using, 1144–1147
Collections
Collection interface, 776–780
forEach method, 782
iterators for traversing collections, 780–781
singleton and unmodifiable, 835
static methods for, 792–795
TestCollection.java example, 778–780
Collections class
singleton and unmodifiable collections, 835
static methods, 792
Collections Framework hierarchy
ArrayList and LinkedList class, 784–787
case study: displaying bouncing balls, 795–798
case study: stacks used to evaluate expressions, 803–807
Collection interface, 776–778
Comparator interface, 787–791
Dequeue interface, 800–803
designing complex data structures, 1054
forEach method, 782
iterators for traversing collections, 780–781
key terms, 807
List interface, 783–787
Map interface, 1016
methods of List interface, 783–787
overview of, 776
PriorityQueue class, 802–803
Queue interface, 800
queues and priority queues, 800–803
quiz and exercises, 808–814
static methods for lists and collections, 792–795
summary, 808
TestCollection.java example, 778–780
TestIterator.java example, 780–781
Vector and Stack classes, 798–800
Collisions, in hashing
double hashing, 1022–1023
handling using open addressing, 1019–1023
handling using separate chaining, 1023–1025
linear probing, 1019–1020
overview of, 1017
quadratic probing, 1020–1021
Column index, 291
ComboBox
ComboBoxDemo.java, 660–662
overview of, 659–660
Command-line arguments, 276–279
Comments
code maintainability and, 103
programming style and, 18
in Welcome.java, 13
Common denominator, finding greatest common denominator. *see* Gcd (greatest common denominator)
Communication devices, computers and, 6
Compact discs (CDs), as storage device, 5
Comparable interface
ComparableRectangle.java example, 515–516
Comparator interface *vs.*, 788–789
enumerated types, 1181
as example of generic interface, 752–753
generic method for sorting array of Comparable objects, 758
overview of, 514–515
PriorityQueue class and, 802
Rational class implementing, 528
SortComparableObjects.java example, 515
SortRectangles.java example, 516–517
TreeMap class and, 831
Comparator interface
Comparable *vs.*, 789
GeometricObjectComparator.java, 787–788
methods of, 787–788
PriorityQueue class and, 802–803
sorted method, 1134
SortStringByLength.java, 789
SortStringIgnoreCase.java, 790–791
TestComparator.java, 788
TestTreeSetWithComparator.java example, 821–823
TreeMap class and, 831
compare method, 787–788
compareTo method
Cloneable interface and, 518
Comparable interface defining, 514
ComparableRectangle.java example, 515–516
comparing strings, generic method for sorting array of Comparable objects, 759
implementing in Rational class, 528

wrapper classes and, 383–384
compareToIgnoreCase method, 136, 790
Comparison operators (==), 78, 434
Compatibility, raw types and backward compatibility, 760–761
Compile errors (Syntax errors)
common errors, 13–14
debugging, 108
programming errors, 19–20
Compile time
error detection at, 752
restrictions on generic types, 765
Xlint:unchecked error, 760
Compilers
ambiguous invocation and, 223
reporting syntax errors, 19–20
translating Java source file into bytecode file, 15–16
translating source program into machine code, 8, 9
Complete binary tree, 900, 906
Complete graphs, 1048
Completeness, class design guidelines, 533
Complex numbers, Math class, 538
Components
ListView, 662–664
ListViewDemo.java, 664–665
quiz and exercises, 682–690
ScrollBar, 665–667
ScrollBarDemo.java, 666–667
SliderDemo.java, 669–670
sliders, 669
summary, 681–682
TextArea, 655–658
TextAreaDemo.java, 658
TextFieldDemo.java, 654–655
Composition, in designing stacks and queues, 950–951
Composition relationships
aggregation and, 376–377
between ArrayList and MyStack, 441–442
Compound expressions
case study: stacks used to evaluate, 803–804
EvaluateExpression.java example, 805–807
Compression
data compression using Huffman coding, 985–990
of hash codes, 1018–1019
HuffmanCode.java example, 987–990
Compute expression, 14
Computers
communication devices, 6–7
CPUs, 3
input/output devices, 5–6
memory, 4
OSs (operating systems), 9–10
overview of, 2–3
programming languages, 7–9
storage devices, 4–5
concat method, 133
Concatenate strings, 36, 133
Conditional operators, 105–106
Connect four game, 315
Connected circles problem
ConnectedCircles.java, 1073
overview of, 1072–1073
Connected graphs, 1048
Consistency, class design guidelines, 532
Consoles
defined, 12
formatting output, 146–150
input, 12
output, 12
reading input, 37–39
Constant time, comparing growth functions, 848
Constants
class, 339–340
declaring, 340
identifiers, 40
KeyCode constants, 613
named constants, 43–44
naming conventions, 44
wrapper classes and, 383
Constructor chaining, 419–420
Constructor modifiers, 1168–1169
Constructors, 360
in abstract classes, 502
for AVLTree class, 1002–1003
for BMI class, 373–374
calling subclass constructors, 418
creating Date class, 337
creating objects with, 331
creating Random objects, 337
for DataInputStream and DataOutputStream classes, 699
generic classes and, 756
interfaces *vs.* abstract classes, 523
invoking with this reference, 360
for Loan class, 370–372
object methods and, 324
private, 346
for String class, 388
for StringBuilder class, 395–396
in TestCircle.java example, 327, 328
in TV.java example, 329
UML diagram of, 326
for UnweightedGraph class, 1058–1059
for WeightedGraph class, 1092–1093
wrapper classes and, 383
Containers
creating data structures, 776
maps as, 828
removing elements from, 826
storing objects in, 777
types supported by Java Collections Framework, 776
contains method, 826, 827
continue statements, for controlling loops, 186–189
Contract, object class as, 324
Control, 545–548
Control units, CPUs, 3
Control variables, in for loops, 173–174
ControlCircle.java, 600–601
Conversion methods, for wrapper classes, 383
Convex hull
finding for set of points, 867–869
gift-wrapping algorithm applied to, 867–868
Graham's algorithm applied to, 868–869
string matching, 869–876
Copying
arrays, 260–261
files, 704–706
Core, of CPU, 3
cos method, trigonometry, 122–123
Cosine function, 589
count method, 1134–1135
Counter-controlled loops, 161
Coupon collector's problem, 284–285
Course class, 378–379
CPUs (central processing units), 3

Cubic time, comparing growth functions, 848
Curly braces {}
  in block syntax, 13
  dangers of omitting, 174–175
  forgetting to use, 85–86
Cursor, mouse, 6
Cycle, connected graphs, 1048

D

.dat files (binary), 694
Data, arrays for referencing, 250
Data compression
  Huffman coding for, 985–990
  `HuffmanCode.java` example, 987–990
Data fields
  accessing object data, 333
  encapsulating, 346–349, 532
  in interfaces, 512
  object state represented by, 324
  referencing, 334, 358–360
  in `TestCircle.java` example, 327
  in `TV.java` example, 329
  UML diagram of, 326
Data modifiers, 1168–1169
Data streams. *see* `DataInputStream`/`DataOutputStream` classes
Data structures. *see also* Collections Framework hierarchy
  array lists. *see* `ArrayList` class
  choosing, 776
  collections. *see* Collections
  first-in, first-out, 800
  linked lists. *see* `LinkedList` class
  lists. *see* Lists
  queues. *see* Queues
  stacks. *see* Stacks
Data structures, implementing
  array lists, 928–935
  `GenericQueue.java` example, 951
  implementing `MyLinkedList` class, 939–947
  linked lists, 935–949
  lists, 924–928
  `MyArrayList` compared with `MyLinkedList`, 947
  `MyArrayList.java` example, 929–933
  `MyLinkedList` class, 924–925, 937, 947
  `MyPriorityQueue.java` example, 953–954
  overview of, 924
  priority queues, 953–954
  quiz and exercises, 955–957
  stacks and queues, 949–953
  summary, 955
  `TestMyArrayList.java` example, 933–935
  `TestMyLinkedList.java` example, 938–939
  `TestPriorityQueue.java` example, 954
  `TestStackQueue.java` example, 952–953
  variations on linked lists, 948–949
Data types
  ADT (Abstract data type), 368
  `boolean`, 78–80, 337
  `char`. *see* Characters (`char` data type)
  `double`. *see* double (double precision), numeric types
  `float`. *see* Floating-point numbers (`float` data type)
  fundamental. *see* Primitive types (fundamental types)
  generic. *see* Generics
  `int`. *see* Integers (`int` data type)
  `long`. *see* Long, numeric types
  numeric, 45–48
  reference types. *see* Reference types
  specifying, 35
  strings, 131
  types of, 41
  using abstract class as, 504
`DataInputStream`/`DataOutputStream` classes
  `DetectEndOfFile.java`, 701
  external sorts and, 909
  overview of, 698
  `TestDataStream.java`, 699–700
`Date` class
  case study: `Calendar` and `GregorianCalendar` classes, 507–508
  `java.util`, 336
De Morgan's law, 97
Debugging
  benefits of stepwise refinement, 234
  code modularization and, 217
  selections, 108
Decimal numbers
  `BigDecimal` class, 387–388
  converting to hexadecimals, 184–186, 219–221, 745
  converting to/from binary, 745, 1172
  converting to/from hexadecimal, 1173
  equivalents of ASCII character set, 1165
  overview of, 1171
Declaring constants, 43, 340
Declaring exceptions
  `CircleWithException.java` example, 467
  `ReadData.java` example, 483–484
  `TestCircleWithCustomException.java` example, 475–476
  `throws` keyword for, 462, 467
Declaring methods
  generic methods, 757
  static methods, 339
Declaring variables
  array variables, 250
  overview of, 40–41
  specifying data types and, 35–36
  two-dimensional array variables, 290–291
Decrement (`--`) operator, 57–58
Deep copies, 521
Default field values, for data fields, 334
Degree of vertex, 1048
Delete key, on keyboards, 6
`delete` method, `AVLTree` class, 1007, 1011
Delimiters, token reading methods and, 484
Denominator. *see* Gcd (greatest common denominator)
Denominators, in rational numbers, 526
Depth-first searches (DFS)
  applications, 1071–1072
  case study: connected circles problem, 1072–1073
  finding DFS trees, 1048
  implementing, 1069–1071
  traversing graphs, 1067–1068
Depth-first traversal, tree traversal, 964
`Deque` interface, `LinkedList` class, 800–803
`dequeue` method, 953
`DescriptionPane` class, 657–658
Descriptive names
  benefits of, 40
  for variables, 35
Deserialization, of objects, 709
Design guidelines, classes, 531–534
Determining Big *O*

for repetition statements, 842–845
for selection statements, 842–845
for sequence statements, 842–845
DFS (depth-first searches). *see* Depth-first searches (DFS)
Dial-up modems, 6
Dictionaries, 1016. *see also* Maps
Digital subscriber lines (DSLs), 6
Digital versatile disc (DVDs), 5
Digits, matching, 100
Dijkstra's single-source shortest-path algorithm, 1110
Direct recursion, 723
Directed graphs, 1047
Directories
case study: determining directory size, 731–732
`DirectorySize.java`, 731–732
`File` class and, 478
file paths, 477
`disjoint` method, 794
Disks, as storage device, 5
Display message
in `Welcome.java`, 12
in `WelcomeWithThreeMessages.java`, 14
`distinct` method, 1132, 1134–1135
Divide-and-conquer algorithm, 861–864
Divide-and-conquer strategy. *see* Stepwise refinement
Division (`/=`) assignment operator, 42
Division operator (`/`), 46, 53
Documentation, programming and, 18
Dot operator (`.`), 333
Dot pitch, measuring sharpness of displays, 6
`double` (double precision), numeric types
converting characters and numeric values to strings, 392
declaring variables and, 41
generic method for sorting array of `Comparable` objects, 758
hash codes for primitive types, 1017
`java.util.Random`, 337
overview of numeric types, 45
precision of, 181
Double hashing, collision handling, 1022–1023
`Double.parseDouble` method, 138
`DoubleStream`, 1136–1139
Doubly linked lists, 948
deciding when to use, 176–178
`do-while` loops, 171–173
`do-while` loops, 171–173
Downcasting objects, 429
`drawArc` method, 575–577
Drives, 5
Drop-down lists. *see* ComboBox
DSLs (digital subscriber lines), 6
DVDs (Digital versatile disc), 5
Dynamic binding, inheritance and, 425–429
Dynamic programming
computing Fibonacci numbers, 849–851
definition, 850
Dijkstra's algorithm, 1110
`ImprovedFibonacci.java` example, 850–851

E

Eclipse
built in debugging, 108
creating/editing Java source code, 15
`Edge` arrays
representing edges, 1051
weighted edges using, 1093–1094
`Edge` class, 1051
Edges
adjacency lists, 1052–1054
adjacency matrices, 1052
adjacent and incident, 1048
defining as objects, 1051
`Graph.java` example, 1056
on graphs, 1047
Prim's algorithm and, 1103
representing edge arrays, 1051
`TestGraph.java` example, 1056
`TestMinimumSpanningTree.java`, 1107–1108
`TestWeightedGraph.java`, 1100–1101
weighted adjacency matrices, 1094
weighted edges using edge array, 1093–1094
weighted graphs, 1093
`WeightedGraph` class, 1091
Edge-weighted graphs
overview of, 1093
`WeightedGraph` class, 1091
Eight Queens puzzle
`EightQueens.java` example, 865–867
recursion, 748
single-dimensional arrays, 288
solving, 864–867
Element type, specifying for arrays, 250
Emirp, 243
Encapsulation
in `Circle.java` (for `CircleWithPrivateDataFields`)
example, 347–348
class design guidelines, 530
of classes, 368–372
of data fields, 346–349
information hiding with, 227
of `Rational` class, 530
Encoding schemes
defined, 3–4
mapping characters to binary equivalents, 127
End of file exception (`EOFException`), 701
End-of-line style, block styles, 19
`enqueue` method, 953
`entrySet` method, `Map` interface, 830
Enumerated types
with data fields, constructors, and methods, 1184–1185
`EnumeratedTypeDemo.java` example, 1182
`if` statements with, 1183
simple, 1181–1183
`StandaloneEnumTypeDemo.java` example, 1182–1183
`switch` statements with, 1183
`TestTrafficLight.java` example, 1184–1185
`TrafficLight.java` example, 1184
`values` method, 1184
Equal (`=`) operator, for assignment, 78
== (equal to operator), 78
Equal to (`==`) operator, for comparison, 76
`equalArea` method, for comparing areas of geometric objects, 503
Equals method
`Arrays` class, 275
`Comparator` interface, 787
`Object` class, 424
Erasure and restrictions, on generics, 764–766
`Error` class, 460, 462
Errors, programming. *see also* Programming errors
(escape characters), 128

Euclid's algorithm
  finding greatest common divisors using, 851–855
  `GCDEuclid.java` example, 852–855
Euler, 1046–1047
Event delegation, 597
Event handlers/event handling, 594–595, 605–607, 613
`Exception` class
  exceptions in, 460
  extending, 474
  in `java.lang`, 474
  subclasses of, 460–461
Exception handling. *see also* Programming errors
  catching exceptions, 463–465, 467–470
  chained exceptions, 473–474
  `ChainedExceptionDemo.java` example, 473–474
  checked and unchecked, 461
  `CircleWithException.java` example, 467
  `ClassCastException`, 430
  declaring exceptions (`throws`), 462, 467
  defined, 454
  defining custom exception classes, 474–477
  `EOFException`, 701
  in `Exception` class, 460
  exception classes cannot be generic, 766
  `FileNotFoundException`, 695
  files input/output, 480–486
  finally clause in, 470–471
  getting information about exceptions, 465–467
  in `House.java` example, 521
  `InputMismatchExceptionDemo.java` example, 458–459
  `InvalidRadiusException.java` example, 474–475
  `IOException`, 461
  key terms, 491
  `NotSerializableException`, 709
  overview of, 454–459
  quiz and exercises, 492–497
  `Quotient.java` example, 454
  `QuotientWithException.java` example, 456–458
  `QuotientWithIf.java` example, 455
  `QuotientWithMethod.java` example, 455–456
  `ReadData.java` example, 483–484
  `ReadFileFromURL.java` example, 487–488
  `ReplaceText.java` example, 485–486
  rethrowing exceptions, 472–473
  summary, 491–492
  `TestCircleWithCustomException.java` example, 475–477
  `TestCircleWithException.java` example, 468–470
  `TestException.java` example, 466–467
  `TestFileClass.java` example, 479
  throwing exceptions, 462–463, 467–470
  types of exceptions, 459–461
  unsupported operations of `Collection` interface, 778
  `WebCrawler.java` example, 489–491
  when to use exceptions, 472
  `WriteData.java` example, 480–481
  `WriteDataWithAutoClose.java` example, 481–482
Exception propagation, 463
Exclusive or (`^`) logical operator, 95–99
Execution stacks. *see* Call stacks
`exists` method, for checking file instances, 478
Explicit casting, 58, 59, 429
Exponent method, `Math` class, 123
Exponential algorithms, 847, 877
Expressions
  assignment statements and, 42–43
  behind the scene evaluation, 107
  Boolean. *see* Boolean expressions
  case study: stacks used to evaluate, 803–804
  `EvaluateExpression.java` example, 805–807
`extends` keyword, interface inheritance and, 524
External sorts
  complexity, 916
  `CreateLargeFile.java` example, 909–910
  implementation phases, 911–913
  overview of, 909

F

Factorials
  case study: computing factorials, 720–723
  `ComputeFactorial.java`, 721–723
  `ComputeFactorialTailRecusion.java`, 741
  tail recursion and, 740–741
`FadeTransition`, 622–623
Fahrenheit, converting Celsius to/from, 237–238
Fall-through behavior, `switch` statements, 103
Feet, converting to/from meters, 238
`fib` method, 724–726
Fibonacci, Leonardo, 724
Fibonacci numbers
  algorithm for finding, 849–851
  case study: computing, 723–726
  `ComputeFibonacci.java`, 724–726
  computing recursively, 743
  `ImprovedFibonacci.java` example, 850–851
  recurrence relations and, 847
`File` class, 477–479, 692
File I/O. *see* Input; Output
File pointers, random-access files and, 712
`FileInputStream`/`FileOutputStream` classes
  overview of, 695–696
  `TestFileStream.java`, 696–698
Files
  case study: copying files, 704–706
  case study: replacing text in, 485–486
  `File` class, 477–479, 692
  input/output, 480–486
  key terms, 491
  quiz and exercises, 492–497
  reading data from, 482–483
  reading data from Web, 487–488
  summary, 491–492
  `TestFileClass.java` example, 479
  writing data to, 480–481
`fill` method, 794
`filter` method, 1132, 1134
`FilterInputStream`/`FilterOutputStream` classes, 698
`final` keyword, for declaring constants, 43
`final` modifier, for preventing classes from being extended, 445
`finally` clause, in exception handling, 470–471
`findAny` method, 1133, 1135–1136
`findFirst` method, 1133, 1135–1136
First-in, first-out data structures, 800
`float` data type. *see* Floating-point numbers (`float` data type)
Floating-point numbers (`float` data type)
  approximation, 67
  converting to integers, 58
  hash codes for primitive types, 1017
  `java.util.Random`, 337
  minimizing numeric errors related to loops, 180–181
  numeric types for, 45
  overview of numeric types, 45

special values, 1170
specifying data types, 35
specifying precision, 148
Flowcharts
do-while loop, 171
if statements and, 80–81
if-else statements, 82–83
for loops, 173, 174
switch statements, 102–103
while loops, 160, 161
FlowPane
HBox and VBox, 564–565
overview, 559
ShowFlowPane.java, 559–561
Folding, hash codes and, 1017
Font, FontDemo.java, 554–555
for loops
deciding when to use, 176–178
nesting, 178–180, 293
overview of, 173–176
processing arrays with, 253
variable scope and, 224–225
foreach (enhanced) loops
implicit use of iterator by, 786
overview of, 255–257
for traversing collections, 781
forEach method, 1133
Formal generic type, 752
Formal parameters. *see* Parameters
Format specifiers, 147–149
FORTRAN, high-level languages, 8
Forward pointer, in doubly linked lists, 948
Fractals
case study, 736–739
H-tree fractals, 749
Koch snowflake fractal, 747
SierpinskiTriangle.java, 736–739
Frames (windows)
ScrollBarDemo.java, 666–667
SliderDemo.java, 669–670
Free cells, in Sudoku grid, 300
frequency method, 794
Function keys, on keyboards, 6
Functions, 207. *see also* Methods
Fundamental types (Primitive types). *see* Primitive types (fundamental types)

G

Galton box, 288
Garbage collection, JVM and, 335
GBs (gigabytes), of storage, 4
Gcd (greatest common denominator)
algorithm for finding, 851–855
case study: finding greatest common denominator, 182–183
computing recursively, 742
gcd method, 217–218
GCDEuclid.java example, 853–855
GCD.java example, 852–853
Rational class and, 526–527
Generic instantiation, 752
Generics
case study: designing class for matrix using generic types, 766–771
case study: generic method for sorting array, 758–759
defining generic classes and interfaces, 754–756
erasing generic types, 764–766
GenericStack class, 755–756
key terms, 771
methods, 756–758
motivation for using, 752–754
overview of, 752
questions and exercises, 772–773
raw types and backward compatibility and, 760–761
restrictions on generic types, 764–766
summary, 771–772
wildcards for specifying range of generic types, 761–764
GeometricObject class
Circle.java and Rectangle.java, 502
overview of, 501
TestGeometricObject.java example, 502–503
getAbsolutePath(), File class, 479
getArea method, Circle example, 327, 328
getArray method, 295–296
getBMI method, BMI class, 373, 374
getCharacterFrequency method, 988
getChars method, converting strings into arrays, 392
getDateCreated method, Date class, 356
getIndex method, ArrayList class, 437
getMinimumSpanningTree method, WeightedGraph class, 1103–1104
getPerimeter method, Circle example, 327
getRadius method, CircleWithPrivateDataFields.java example, 348
getRandomLowerCaseLetter method, 265, 267
getSize method, finding directory size, 442, 1154–1155
getSource method, events, 596
getStackTrace method, for getting information about exceptions, 466
getStatus method, BMI class, 373, 374
Getter (accessor) methods
ArrayList class and, 438
encapsulation of data fields and, 347–348
implementing linked lists, 935
Gift-wrapping algorithm, 867–868
Gigabytes (GBs), of storage, 4
Gigahertz (GHz), clock speed, 3
GMT (Greenwich Mean Time), 54
Gosling, James, 10
Graham's algorithm, 868–869
Graph interface, 1054
Graph theory, 1046
Graphical user interface (GUI), 644
Graphs
breadth-first searches (BFS), 1074–1075
case study: connected circles problem, 1072–1074
case study: nine tails problem, 1077–1083
ConnectedCircles.java, 1073
DisplayUSMap.java example, 1066–1067
Graph.java example, 1056–1057
GraphView.java example, 1064–1065
key terms, 1083
modeling, 1054–1064
overview of, 1046–1047
questions and exercises, 1083–1089
representing edges, 1051–1054
representing vertices, 1048–1050
summary, 1083
terminology regarding, 1047–1048
TestGraph.java example, 1056–1057
traversing, 1067
UnweightedGraph.java example, 1058–1064
visualization of, 1064–1067
Greater than (>) comparison operator, 78
Greater than or equal to (>=) comparison operator, 78

Greatest common denominator. *see* Gcd (greatest common denominator)
Greedy algorithms
  Dijkstra's algorithm, 1109
  overview of, 986
Greenwich Mean Time (GMT), 54
GregorianCalendar class
  Cloneable interface and, 518–519
  in **java.util** package, 363
  overview of, 507–508
  TestCalendar.java example, 508–510
GridPane
  overview, 561–562
  ShowGridPane.java, 562–563
Grids, representing using two-dimensional array, 300
Group classifier, 1147
Group processor, 1147
groupingby collector, grouping elements using, 1147–1150
Growth rates
  algorithm for comparing, 847–848
  comparing algorithms based on, 840

H

Hamiltonian path/cycle, 1087
HandleEvent.java, 595–596
Hand-traces, for debugging, 108
Hangman game, 287, 495, 589, 809, 810
Hard disks, as storage device, 5
Hardware, 2
Has-a relationships
  in aggregation models, 376–377
  composition and, 442
Hash codes, 1017
  compressing, 1018–1019
  vs. hash functions, 1017–1019
  for primitive types, 1017
  for strings, 1017–1018
Hash functions, 1016
  vs. hash codes, 1017–1019
  as index to hash table, 1016
Hash map, 831, 1034
Hash set, 816
Hash tables, 1016, 1035. *see also* Maps
  measuring fullness using load factor, 1024–1025
  parameters, 1031
hashCode method, 816, 1017
Hashing
  collisions handling using open addressing, 1019–1023
  collisions handling using separate chaining, 1023
  compressing hash codes, 1018–1019
  double hashing open addressing, 1022–1023
  function, 1016–1019
  hash codes for primitive types, 1017
  hash codes for strings, 1017–1018
  hash functions *vs.* hash codes, 1017–1019
  key terms, 1041–1042
  linear probing open addressing, 1019–1020
  load factor and rehashing, 1025
  map implementation with, 1025–1034
  MyHashMap.java example of map implementation, 1027–1033
  MyHashSet.java example of set implementation, 1034–1041
  MyMap.java example of map implementation, 1026–1027
  overview of, 1016
  quadratic probing open addressing, 1020–1021
  quiz and exercises, 1042–1044
  set implementation with, 1034–1041
  summary, 1042
  TestMyHashMap.java example of map implementation, 1033–1034
  TestMyHashSet.java example of set implementation, 1041
  what it is, 1016–1017
HashMap class
  concrete implementation of **Map** class, 828–830
  implementation of **Map** class, 1016
  load factor thresholds, 1025
  overview of, 831
  TestMap.java example, 831–833
  types of maps, 829
HashSet class
  case study: counting keywords, 827–828
  implementation of **Set** class, 1034
  overview of, 816–820
  TestHashSet.java example, 817–818
  TestMethodsInCollection.java example, 818–819
  types of sets, 816
Hashtable, 831
HBox and **VBox**
  definition, 566
  overview, 564
  ShowHBoxVBox.java, 565–566
Heap class
  Heap.java example, 904–905
  operations for manipulating heaps in, 904
  sorting arrays with, 906
Heap sorts, 900–907
  adding nodes to heaps, 901–902
  arrays using heaps, 906
  Heap class, 903–904
  Heap.java example, 904–905
  HeapSort.java example, 906
  overview of, 900–901
  removing root from heap, 902–903
  storing heaps, 901
  time complexity of, 906–907
Heaps
  adding nodes to, 901–902
  arrays using, 906
  binary heaps (binary trees), 900
  dynamic memory allocation and, 263
  height of, 906
  implementing priority queues with, 953–954
  removing root from, 902–903
  storing, 901
Height, 960
Height of a heap, 906
Helper method, recursive
  overview of, 728
  RecursivePalindrome.java, 728–729
Hertz (Hz), clock speed in, 3
Hexadecimal numbers
  converting to/from binary, 1173–1174
  converting to/from decimal, 143–144, 184–186, 219–221, 745, 1173
  equivalents of ASCII character set, 1165
  overview of, 1171
Hidden data fields, 358, 360
High-level languages, 8–9
Hilbert curve, 749
Horizontal scroll bars, 666
Horizontal sliders, 668, 669
H-trees
  fractals, 749
  recursive approach to, 720
Huffman coding, 985–990

Huffman coding trees
  data compression using, 985–990
  HuffmanCode.java example, 987–990
Hz (Hertz), clock speed in, 3

I

Identifiers, 40
IDEs (integrated development environments) for creating/editing Java source code, 11, 12, 15–16
IEEE (Institute of Electrical and Electronics Engineers), floating point standard (IEEE 754), 45
if statements
  common errors, 85–89
  in computing body mass index, 91–92
  in computing taxes, 92–95
  conditional operator used with, 105–106
  with enumerated types, 1183
  nesting, 83–85
  overview of, 80–82
  SimpleIfDemo.java example, 81–82
if-else statements
  conditional expressions and, 105
  dangling else ambiguity, 87
  multi-way, 83–85
  overview of, 82–85
  recursion and, 726
IllegalArgumentException class, 463
Image, 556–558
Image class, 556
Image icons, ComboBoxDemo.java, 660
Images, ShowImage.java, 557–558
ImageView, 556–558
Immutable
  BigInteger and BigDecimal classes, 387–388
  class, 355
  objects, 355–356
  Rational class, 530
  String object, 388–389
  wrapper classes, 383
Implementation (coding), in software development process, 63–64
Implementation methods, 231–234
Implicit casting, 129, 429
Importing, types of import statements, 38
Increment method, in Increment.java example, 214
Increment (++) operator, 57–58
Incremental development
  benefits of stepwise refinement, 234
  coding incrementally, 164
Indentation, programming style, 19
Indexed variables elements, 252
Indexes
  accessing elements in arrays, 250, 252
  finding characters/substrings in a string, 137–138
  List interface and, 783, 784
  MyList.java example, 926–928
  string index range, 133
indexOf method, 137–138
  implementing MyLinkedList, 946
  List interface, 783
  MyArrayList.java example, 926, 931, 933
Indirect recursion, 723
Infinite loops, 162
Infinite recursion, 723
Information
  getting information about exceptions, 465–467
  hiding (encapsulation), 227
Inheritance
  ArrayList object, 434–435
  calling subclass constructors, 418
  calling superclass methods, 420–421
  case study: custom stack class, 441–442
  casting objects and, 429–433
  CastingDemo.java example, 430–431
  Circle.java example, 414–416
  constructor chaining and, 419–420
  in designing stacks and queues, 950
  DistinctNumbers.java example, 438–440
  dynamic binding and, 425–429
  equals method of Object class, 433–434
  generic classes, 756
  GeometricObject.java example, 413–414
  interface inheritance, 510–511, 524
  is-a relationships and, 442
  key terms, 445
  Object class and, 424
  overriding methods and, 421–422
  overview of, 412
  preventing classes from being extended or overridden, 445
  protected data and methods, 442–444
  quiz and exercises, 447–451
  Rectangle.java example, 416–417
  summary, 446
  superclasses and subclasses and, 412–418
  TestArrayList.java example, 435–438
  TestCircleRectangle.java example, 417–418
  usingsuper keyword, 418
Initializing variables
  AnalyzeNumbers.java, 257
  arrays, 252
  declaring variables and, 41
  multidimensional arrays, 291
  two-dimensional arrays, 293
Inner (nested) classes
  anonymous, 602–603
  AnonymousHandlerDemo.java, 603–605
  for defining listener classes, 601–602
  KeyEventDemo.java, 614–615
  ShortestPathTree class as inner class of WeightedGraph class, 1127–1128
  TicTacToe.java, 672–676
Inorder traversal
  time complexity of, 978
  tree traversal, 963
Input
  reading from console, 37–39
  redirecting using while loops, 169–170
  runtime errors, 21
  streams. *see* InputStream classes
Input, process, output (IPO), 39
InputMismatchException class, 458, 484
Input/output devices, computers and, 5–6
InputStream classes
  BufferedInputStream, 701–704
  case study: copying files, 705–706
  DataInputStream, 698–701
  deserialization and, 709
  DetectEndOfFile.java, 701
  FileInputStream, 695–696
  FilterInputStream, 698
  ObjectInputStream, 706–707
  overview of, 694–695

InputStream classes (*Continued*)
  TestDataStream.java, 699–700
  TestFileStream.java, 696–698
  TestObjectInputStream.java, 708
Insert key, on keyboards, 6
insert method
  AVLTree class, 1011
  overriding, 1000–1001
Insertion order, LinkedHashMap class, 831
Insertion sort algorithm
  analyzing, 846
  recurrence relations and, 847
Insertion sorts
  algorithms, 882–884
  InsertionSort.java example, 888–889
  time complexity of, 890
Instance methods
  accessing object data and methods, 333
  in Circle.java (for CircleWithStaticMembers), 340–341
  class design guidelines, 326–327
  invoking, 370, 373
  when to use instance methods *vs.* static, 341–344
Instance variables
  accessing object data and methods, 333
  class design guidelines, 343
  static variables compared with, 339–341
  in TestCircleWithStaticMembers.java example, 341
  when to use instance variables *vs.* static, 341–344
Instances. *see also* Objects
  checking file instantiation, 478
  checking object instantiation, 324, 430
  generic instantiation, 752
Institute of Electrical and Electronics Engineers (IEEE), floating point standard (IEEE 754), 45
int data type. *see* Integers (int data type)
Integer.parseInt method, 138
Integers (int data type)
  ArrayList for, 439
  BigInteger class, 387–388
  bit operators and, 1175
  case study: designing class for matrix using generic types, 766, 767
  casting to/from char types, 128–129
  converting characters and numeric values to strings, 392–393
  declaring variables and, 41
  division of, 454–458
  finding larger between two, 207
  floating-point numbers converted to, 56–57
  generic method for sorting array of Comparable objects, 758
  greatest common denominator of, 851
  hash codes for primitive types, 1017
  IntegerMatrix.java example, 769
  java.util.Random, 337
  numeric types for, 45
  sorting, 907
  sortingint values, 913
  specifying data types, 35
  TestIntegerMatrix.java example, 770
Integrated development environments (IDEs), 11, 12
  for creating/editing Java source code, 15–16
  overview of, 11
Intelligent guesses, 164
Interfaces
  abstract classes compared with, 521–523
  benefits of, 516
  benefits of generics, 752
  case study: Rational class, 526–527
  Cloneable interface, 518–519
  Comparable interface, 513–514
  ComparableRectangle.java example, 515–516
  for defining common class behaviors, 566
  defining generic, 754–756
  House.java example, 519–523
  key terms, 534
  overview of, 501
  questions and exercises, 535–540
  raw types and backward compartibility, 760
  SortComparableObjects.java example, 515
  SortRectangles.java example, 516–517
  summary, 534–535
  TestEdible.java example, 510–513
Intermediate method, Stream interface, 1130
Interned strings, 389
Interpreters, translating source program into machine code, 8, 9
IntStream, 1136–1139
Invoking methods, 208, 209, 333, 757
  binary I/O classes, 694–695
  BufferedInputStream and BufferedOutputStream classes, 701–704
  case study: copying files, 704–705
  case study: replacing text, 485–486
  Copy.java, 705–706
  DataInputStream and DataOutputStream classes, 698–701
  DetectEndOfFile.java, 701
  FileInputStream and FileOutputStream classes, 695–698
  FilterInputStream and FilterOutputStream classes, 698
  handling text I/O in Java, 692–693
  key terms, 714
  object I/O, 706–707
  overview of, 481, 692
  questions and exercises, 715–718
  random-access files, 711–714
  reading data from file using Scanner class, 482–484
  reading data from the Web, 487–488
  serializable interface, 708–709
  serializing arrays, 709–711
  summary, 715
  TestDataStream.java, 699–700
  TestFileStream.java, 696–698
  TestObjectInputStream.java, 708
  TestObjectOutputStream.java, 707–708
  TestRandomAccessFile.java, 713–714
  text I/O *vs.* binary I/O, 693–694
  types of I/O devices, 5–6
  writing data to file using PrintWriter class, 480–481
IOException, 695, 696
IPO (input, process, output), 39
Is-a relationships
  design guide for when to use interfaces *vs.* classes, 524
  inheritance and, 442
isAbsolute method, for checking file instances, 478–479
isDigit method, Character class, 144
isDirectory method, for checking file instances, 478–479
isFile method, for checking file instances, 478–479
isHidden method, for checking file instances, 478–479
Is-kind-of relationships, 524
isPalindrome method
  RecursivePalindrome.java, 728–729
  as tail-recursive method, 740
isPrime method, prime numbers, 219
isValid method, applying to Sudoku grid, 302
Iterable interface, 780

Iteration/iterators
  binary search trees and, 983–985
  implementingMyLinkedList, 944, 947
  Iterable interface, 984
  Iterator object, 780
  lists and, 780–781
  loops and, 160
  MyArrayList.java example, 932, 933
  recursion compared with, 740
  TestIterator.java example, 780–781
  TestMyArrayList.java example, 933–934
  traversing collections, 780–781

J

Java Collections Framework. *see* Collections Framework hierarchy
java command, for executing Java program, 11
Java Development Toolkit (JDK)
  jdb debugger in, 108
  overview of, 11, 12
Java EE (Java Enterprise Edition), 11
Java language specification, 11–12
Java Library, 336–339
Java ME (Java Micro Edition), 12
Java programming
  creating, compiling, and executing programs, 15–18
  displaying text in message dialog box, 23
  high-level languages, 8–9
  introduction to, 12
  simple examples, 12–15
  using Eclipse, 25–28
  usingNetBeans, 23–25
Java SE (Java Standard Edition), 11–12
Java Virtual Machine. *see* JVM (Java Virtual Machine)
javac command, for compiling Java program, 17
Javadoc comments (/**.*/), 18
JavaFX
  Arc, 575–577
  binding properties, 548–550
  BorderPane, 563–564
  case study: ClockPane Class, 580–584
  Circle and Ellipse, 572–574
  Color class, 553–554
  FlowPane, 559–561
  Font class, 554–555
  GridPane, 561–563
  HBox and VBox, 564–565
  Image and ImageView Classes, 556–558
  key terms, 585
  Layout panes, 558–559
  Line, 569–570
  nodes, 551–552
  panes, 545–548
  Polygon and Polyline, 577–580
  quiz and exercises, 586–591
  Rectangle, 570–572
  shapes, 567
  structure, 542–545
  summary, 585–586
  vs. Swing and AWT, 542
  Text, 567–569
JavaFX CSS, 551
JavaFX UI controls
  ScrollBar, 665–667
  BounceBallSlider.java, 670–671
  button, 646–648
  ButtonDemo.java, 647–648
  case study: developing tic-tac-toe game, 671–676
  case study: national flags and anthems, 679–681
  CheckBox, 648–651
  CheckBoxDemo.java, 649–651
  ComboBox, 659–662
  ComboBoxDemo.java, 660–662
  DescriptionPane.java, 657–658
  Labeled and Label, 644–646
  LabelWithGraphic.java, 644–646
  ListView, 662–664
  ListViewDemo.java, 664–665
  MediaDemo.java, 677–679
  programming exercises, 682–689
  quiz, 682
  RadioButton, 651–653
  RadioButtonDemo.java, 652–653
  ScrollBar, 665–667
  ScrollBarDemo.java, 666–667
  Slider, 668–671
  SliderDemo.java, 669–670
  TextArea, 655–658
  TextAreaDemo.java, 658
  Textfield, 654–655
  TextFieldDemo.java, 654–655
  TicTacToe.java, 672–676
  video and audio, 676–679
java.io
  File class, 477–479
  PrintWriter class, 480
  RandomAccessFile class, 712
java.lang
  Comparable interface, 514
  Exception class, 474
  Number class, 505
  packages, 64
  Throwable class, 459–461, 465
java.net
  MalformedURLException class, 487
  URL class, 486
java.util
  Arrays class, 272–274
  Calendar class, 507–508
  creating stacks, 806
  Date class, 336
  EventObject class, 596–597
  GregorianCalendar class, 363, 507–508
  Java Collections Framework and, 776
  Random class, 337
  Scanner class, 38, 482–484
jdb debugger, 108
JDK (Java Development Toolkit)
  jdb debugger in, 108
  overview of, 12
JShell, 50–52
JVM (Java Virtual Machine)
  defined, 16
  detecting runtime errors, 454
  garbage collection, 258
  heap as storage area in, 263
  interned string and, 389

K

KBs (kilobytes), 4
Key constants, 613

Keyboards, 5–6
KeyEvents
ControlCircleWithMouseAndKey.java, 615–616
KeyEventDemo.java, 614–615
overview of, 613–614
Keys
hashing functions, 1016
maps and, 1042
keySet method, Map interface, 830
Key/value pairs, in maps, 828–829
Keywords (reserved words)
break and continue, 186–189
case study: counting, 827–828
extends, 524
final, 43
list of Java keywords, 1163
super, 418
throws, 462, 463
transient, 709
in Welcome.java, 13
Kilobytes (KBs), 4
Knight's Tour, 747–748
Knuth-Morris-Pratt algorithm, 873–876
StringMatchKMP.java, 875–876
Koch snowflake fractal, 747
Kruskal's algorithm, 1122

L

Label, 644–646
Labeled, 644–646
Labeling vertices, 1050
Labels, LabelWithGraphic.java, 644–646
Lambda expression
LambdaHandlerDemo.java, 607–609
overview of, 605–607
Landis, E. M., 996
LANs (local area networks), 6
lastIndexOf method
implementing MyLinkedList, 946
List interface, 783
MyArrayList.java example, 931, 933
MyList.java example, 926
strings, 137–138
lastModified method, File class, 479
Latin square, 320–321
Layout panes
BorderPane, 563–564
FlowPane, 559
GridPane, 561–563
HBox and VBox, 564–565
Lazy operator, 98
Leaf, 960
deleting, 975
Left subtree, of binary trees, 960
Left-heavy, balancing AVL nodes, 996, 1003
Length, 960
length method, for checking file instances, 478–479
Length, strings, 131–132, 397
Letters, counting, 265–266
Level, 960
Libraries, APIs as, 11
Line
overview, 569
ShowLine.java, 569–570
Line comments, in Welcome.java, 13
Line numbers, in Welcome.java, 12
Linear probing, collision handling, 1019–1020
Linear search algorithm, 882
comparing growth functions, 848
recurrence relations and, 847
Linear searches, arrays, 269–270
Linked data structures
binary search trees, 960–961
hash maps. *see* LinkedHashMap class
hash sets. *see* LinkedHashSet class
lists. *see* LinkedList class
Linked hash map, 831, 832
Linked hash set, 820, 825
LinkedHashMap class
concrete implementation of Map class, 828–830
implementation of Map class, 1016
overview of, 831
TestMap.java example, 831–833
types of maps, 828–829
LinkedHashSet class
implementation of Set class, 1034
ordering elements in hash sets, 818
overview of, 820
SetListPerformanceTest.java example, 825–826
types of sets, 816
LinkedList class
animation of linked lists, 924, 925
compared with ArrayList class, 784–786
defined under List interface, 783
Deque interface, 800–802
implementing buckets, 1023
implementing linked lists, 935–949
implementing MyLinkedList class, 939–947
implementing queues using linked lists. *see* Queues
MyArrayList compared with MyLinkedList, 947
MyLinkedList, 924–925, 937, 947
representing edges in graphs using linked lists, 1054
SetListPerformanceTest.java example, 825–826
TestArrayAndLinkedList.java, 785–786
TestMyLinkedList.java example, 938–939
variations on linked lists, 948–949
Linux OS, 9
List interface
common features of lists defined in, 924
methods of, 783–784
overview of, 783
Vector class implementing, 799, 800
ListIterator interface, 783
Lists
adjacency lists for representing edges, 1052–1054
array lists. *see* ArrayList class
as collection type, 776
comparing performance with sets, 824–826
implementing, 924–928
linked lists. *see* LinkedList class
List interface, 783–784
ListViewDemo.java, 664–665
methods of List interface, 783–784
MyList.java example, 926–928
singleton and unmodifiable, 835
static methods for, 792–795
ListView, 662–664
Literal values, not using as identifiers, 1163
Literals
Boolean literals, 79
character literals, 126
constructing strings from string literal, 388
defined, 48

floating-point literals, 49
integer literals, 49
LL imbalance, AVL nodes, 996
LL rotation
AVLTree class, 1002
balancing nodes on a path, 1000
implementing, 1001
options for balancing AVL nodes, 996
Load factor
hash sets and, 816
rehashing and, 1025
LoanCalculator.java, 610–611
Loans
Loan calculator case study, in event-driven programming, 609–611
Loan.java object, 370–372
Local area networks (LANs), 6–7
Local variables, 224
Locker puzzle, 284
Logarithmic algorithm, 846
Logic errors (bugs), 21, 108
Logical operators (Boolean operators)
overview of, 95
TestBooleanOperators.java example, 96–99
truth tables, 95–96
Long, numeric types
converting characters and numeric values to strings, 392
hash codes for primitive types, 1017
integer literals and, 49
java.util.Random, 337
overview of numeric types, 45
LongStream, 1136–1139
Loop body, 160
Loop-continuation-condition
do-while loop, 171
loop design and, 166
in multiple subtraction quiz, 166
overview of, 160–161
Loops
break and continue keywords as controls in, 186–189
case study: displaying prime numbers, 191–193
case study: finding greatest common denominator, 182–183
case study: guessing numbers, 163–166
case study: multiple subtraction quiz, 166–168
case study: predicting future tuition, 183–184
creating arrays, 261
deciding which to use, 176–178
design strategies, 166–168
do-while loop, 171–173
examples of determining Big *O*, 842–845
graph edges, 1048
input and output redirections, 170
iteration compared with recursion, 740
key terms, 193
for loops, 173–176
minimizing numeric errors related to, 180–181
nesting, 178–180
overview of, 160
quiz and exercises, 194
sentinel-controlled, 168–170
summary, 193–194
while loop, 160–163
Lottery game, 809
Lower-bound wildcards, 762
Low-level languages, 8
LR imbalance, AVL nodes, 997, 998
LR rotation
AVLTree class, 1003
balancing nodes on a path, 1000
options for balancing AVL nodes, 997, 998

M

Mac OS, 9
Machine language
bytecode compared with, 16
overview of, 7
translating source program into, 8, 9
Machine stacks. *see* Call stacks
Main class
defined, 325
in TestCircle.java example, 326
main method
in Circle.java (AlternativeCircle.java) example, 328, 329
in ComputeExpression.java, 14–15
invoking, 210
main class *vs.*, 325
receiving string arguments from command line, 276–277
in TestCircle.java example, 325
in TestTV.java example, 330–331
in Welcome.java, 12
in WelcomeWithThreeMessages.java, 14
Maintenance, in software development process, 62
MalformedURLException class, 487
Map interface
methods, 829–830, 832
overview of, 829
map method, 1134–1135
Maps
case study: counting occurrence of words using tree map, 833–834
containers supported by Java Collections Framework, 776
hash maps. *see* HashMap class
key terms, 836
linked hash maps. *see* LinkedHashMap class
overview of, 816, 828–833
quiz and exercises, 836–838
singleton and unmodifiable, 835
summary, 836
TestMap.java example, 831–833
tree maps. *see* TreeMap class
Maps, implementing with hashing
MyHashMap.java example, 1027–1033
MyMap.java example, 1026–1027
overview of, 1025
TestMyHashMap.java example, 1033–1034
mapToInt method, 1137, 1138
Marker interfaces, 518
Match braces, in Welcome.java, 12
matches method, strings, 391
Math class
BigInteger and BigDecimal classes, 387
case study: computing angles of a triangle, 125–126
complex numbers, 538–539
exponent methods, 123
invoking object methods, 333
methods generally, 122
random method, 89–90, 100–101, 124
rounding methods, 124
service methods, 122
trigonometric methods, 122
Matrices
adjacency matrices for representing edges, 1052–1054
case study: designing class for matrix using generic types, 766–767

Matrices (*Continued*)
GenericMatrix.java example, 767–769
IntegerMatrix.java example, 769
RationalMatrix.java example, 769–770
TestIntegerMatrix.java example, 770
TestRationalMatrix.java example, 770–771
two-dimensional arrays for storing, 290–291
max and min method, 1132, 1134
max method
defining and invoking, 208–210
finding minimum element in lists, 794
GeometricObjectComparator.java example, 788
MaxUsingGenericType.java example, 760–761
overloading, 221
overview of, 124
maxRow variable, for finding largest sum, 294
Mbps (million bits per second), 7
MBs (megabytes), of storage, 4
Media, 676–679
MediaPlayer, 676–679
MediaView, 676–679
Megabytes (MBs), of storage, 4
Megahertz (MHz), clock speed, 3
Memory, computers, 4
Merge sorts
algorithms, 892
heap sort compared with, 906
merge sort algorithms, 892
MergeSort.java example, 893–894
overview of, 892–896
quick sorts compared with, 900
recurrence relations and, 847
time complexity of, 895
mergeSort method, 895
Mersenne prime, 243
MessagePanel class
ClockPane.java, 582–584
DisplayClock.java, 581–582
Meters, converting to/from feet, 238
Method header, 207
Method modifiers, 207, 1168–1169
Method reference, 790
Method signature, 207
Methods
abstraction and, 227–234
accessing object methods, 333
calling, 208–210
case study: converting decimals to hexadecimals, 184–486
case study: converting a hexadecimal digit to a decimal value, 143–144
case study: generating random numbers, 225–227
case study: generic method for sorting array, 758–759
class, 339–344
Collection interface, 778
commenting, 18
Comparator interface, 787–789
defining, 206–208
generic, 756–758
identifiers, 40
implementation methods, 231–234
invoking, 208, 208, 333, 757
key terms, 234
modularizing code, 217–219
naming conventions, 44
object actions defined by, 324–326
overloading, 221–224
overriding, 1000–1001
overview of, 206
passing arrays to, 261–264
passing objects to, 349–353
passing arguments by values, 213–217
passing to two-dimensional arrays, 295–296
quiz and exercises, 236–248
recursive methods, 720
returning arrays from, 264–265
rounding, 123–124
static. *see* Static methods
stepwise refinement, 227–234
summary, 235
top-down and/or bottom-up implementation, 229–231
top-down design, 228–230
tracing or stepping over as debugging technique, 108
trigonometric, 123
variable scope and, 224–225
void method example, 211–213
MHz (Megahertz), clock speed, 3
Microsoft Windows, 9
Million bits per second (Mbps), 7
min method
finding minimum element in lists, 794
Math class, 24
Minimum spanning trees (MSTs)
MST algorithm, 1103–1105
overview of, 1092
Prim's minimum spanning tree algorithm, 1103–1105
TestMinimumSpanningTree.java, 1107–1109
weighted graphs and, 1087
WeightedGraph class, 1095
Mnemonics, in assembly language, 7
Modeling, graphs and, 1054–1064
Modems (modulator/demodulator), 6
Modifier keys, on keyboards, 6
Modifiers
list of, 1168–1169
method modifiers, 207
Modularizing code
GreatestCommonDivisorMethod.java, 217–218
overview of, 217
PrimeNumberMethod.java, 218–219
Monitors (displays), 6
Motherboard, 3
Mouse, as I/O device, 6
ControlCircleWithMouseAndKey.java, 615–616
event-driven programming, 611–612
MouseEvent, 611–612
MouseEvent, 611–612
MST algorithm, 1103–1105
MST class, 1106–1107
MSTs. *see* Minimum spanning trees (MSTs)
Multi-dimensional arrays. *see* Arrays, multi-dimensional
Multimedia. *see* JavaFX UI controls
Multiple-choice test, 296–298
Multiplication (*=) assignment operator, 56
Multiplication operator (*), 14, 46, 53
Multiplication table, 178, 179
Multiplicities, in object composition, 375
Multiprocessing, 10
Multiprogramming, 10
Multithreading, 10
Multi-way if-else statements
in computing taxes, 92–95
overview of, 83–85
Mutator methods. *see* Setter (mutator) methods

N

Named constants. *see* Constants
Naming conventions
class design guidelines, 532
interfaces, 524
programming and, 44
wrapper classes, 382
Naming rules, identifiers, 40
`NavigableMap` interface, 831
N-by-*n* matrix, 241
Negative angles, drawing arcs, 577
Neighbors
depth-first searches (DFS), 1068
vertices, 1048
Nested classes. *see* Inner (nested) classes
Nested `if` statements
in computing body mass index, 91–92
in computing taxes, 92–95
overview of, 83
Nested loops, 178–180, 293, 843
NetBeans
built in debugging, 106
creating/editing Java source code, 15
Network interface cards (NICs), 6
`new` operator
creating arrays, 250
creating objects, 331
`next` method, whitespace characters and, 134
`nextLine()` method, whitespace characters and, 134
Next-line style, block styles, 19
NICs (network interface cards), 6
Nine tails problem
graphic approach to, 1077–1121
reducing to shortest path problem, 1114–1117
No-arg constructors
class design guidelines, 533
`Loan` class, 370
wrapper classes not having, 383
Node, 545
Nodes, AVL trees
balancing on a path, 1000–1001
creating, 1003
creating and storing in `AVLTreeNode` class, 999–1000
deleting elements, 1002
rotation, 1003–1004
Nodes, binary trees
deleting leaf node, 975
overview of, 960
representing binary search trees, 961–962
Nodes, `JavaFX`, 551–552
Nodes, linked lists
creating, 940, 941
deleting, 941–943
overview of, 935–937
storing elements in, 939, 940
`noneMatch` method, 1134
Nonleaves, finding, 991
Not (`!`) logical operator, 95–99
Not equal to (`!=`) comparison operator, 78
`NotSerializableException`, 709
`null` values, objects, 334
`NullPointerException`, as runtime error, 334
`Number` class
case study: abstract number class, 505
as root class for numeric wrapper classes, 505
Numbers/numeric types
abstract number class, 505–507
binary. *see* Binary numbers
case study: converting hexadecimals to decimals, 143–144, 219–221
case study: displaying prime numbers, 191–193
case study: generating random numbers, 225–227
case study: guessing numbers, 163–166
casting to/from `char` types, 128–129
conversion between strings and, 138–140
converting to/from strings, 392
decimal. *see* Decimal numbers
double. *see* `double` (double precision), numeric types
floating-point. *see* Floating-point numbers (`float` data type)
generating random numbers, 89–90
`GreatestCommonDivisorMethod.java`, 217–218
hexadecimal. *see* Hexadecimal numbers
integers. *see* Integers (`int` data type)
`LargestNumbers.java`, 506–507
overview of, 45–48
`PrimeNumberMethod.java`, 218–219
processing large numbers, 387–388
types of number systems, 1171–1174
Numerators, in rational numbers, 526
Numeric keypads, on keyboards, 6
Numeric literals, 48–50
Numeric operators
applied to characters, 129
overview of, 46–47

O

`Object` class, 424, 433–434
`Object I/O`. *see* `ObjectInputStream`/`ObjectOutputStream` classes
Object member access operator (`.`), 333, 431
Object reference variables, 333
`ObjectInputStream`/`ObjectOutputStream` classes
overview of, 706–707
`serializable` interface, 708–709
serializing arrays, 709–711
`TestObjectInputStream.java`, 708
`TestObjectOutputStream.java`, 707–708
Object-oriented programming (OOP), 324, 333, 372–375
Objects
accessing data and methods of, 333
accessing via reference variables, 332–333
array of, 263
`ArrayList` class, 434–435
arrays of, 353–355
automatic conversion between primitive types and wrapper class types, 386
`BigInteger` and `BigDecimal` classes, 387–388
cannot be created from abstract classes, 504
case study: designing class for stacks, 380–382
case study: designing `Course` class, 378–380
casting, 429–433
in `Circle.java` (for `CircleWithPrivateDataFields`) example, 347–348
`Circle.java` (for `CircleWithStaticMembers`) example, 340–341
class abstraction and encapsulation, 368–372
class design guidelines, 531–534
classes from Java Library, 336–339
comparing primitive variables with reference variables, 334–336
composing, 376–377
constructors, 331
creating, 326–331
data field encapsulation for maintaining classes, 346–349
`Date` class, 336–337
defining classes for, 324–326

Objects (*Continued*)
- edges defined as, 1052
- `equals` method of `Object` class, 433–434
- event listener object, 597
- event objects, 596
- immutable, 355–356
- inheritance. *see* Inheritance
- key terms, 361, 401
- `Loan.java`, 370–372
- `null` values, 334
- `Object` class, 424
- object-oriented thinking, 372–375
- `ObservablePropertyDemo.java`, 616–617
- overview of, 324, 368
- passing to methods, 349–353
- processing primitive data type values as, 382–386
- quiz and exercises, 362–366, 401–410
- `Random` class, 337–338
- reference data fields and, 334
- representing edges, 1051
- `ResizableCircleRectangle.java`, 617–618
- static variables, constants, and methods and, 339–344
- summary, 361–362, 401
- `TestCircle.java` example, 326–327
- in `TestCircleWithPrivateDataFields.java` example, 348–349
- in `TestCircleWithStaticMembers.java` example, 341–344
- in `TestTV.java` example, 330–331
- `this` reference and, 358–361
- `TotalArea.java` example, 354–355
- in `TV.java` example, 329–330
- variable scope and, 357–358
- vertices as object of any type, 1049
- visibility modifiers, 344–348

Off-by-one errors
- arrays and, 255
- in loops, 162

OOP (Object-oriented programming), 324, 333, 372–375

Open addressing, hashing
- collision handling using, 1019–1023
- double hashing, 1022–1023
- linear probing, 1019–1020
- quadratic probing, 1020–1021

Operands
- defined, 97
- incompatible, 97

Operators
- assignment operator (`=`), 42–43
- augmented assignment operators, 56–57
- bit operators, 1175
- comparison operators, 78
- increment and decrement operators, 57–58
- numeric operators, 46–48
- precedence and associativity, 106–107
- precedence and associativity chart, 1166–1167
- processing, 804
- unary and binary, 48

Option buttons. *see* Radio buttons

Or (`||`) logical operator, 95–99

OSs (operating systems)
- overview of, 9
- tasks of, 10

Output
- redirection, 170
- streams, 692

`OutputStream` classes
- `BufferedOutputStream`, 701–704
- case study: copying files, 705
- `DataOutputStream`, 698–701
- `DetectEndOfFile.java`, 701
- `FileOutputStream`, 695–696
- `FilterOutputStream`, 698
- `ObjectOutputStream`, 706–707
- overview of, 694–695
- serialization and, 709
- `TestDataStream.java`, 699–700
- `TestFileStream.java`, 696–698
- `TestObjectOutputStream.java`, 707–708

Overflows
- `Rational` class, 530
- variables, 66

Overloading methods, 221–224

Overriding methods, 421–422, 1000–1001

P

π (pi), estimating, 240

Package-private (package-access) visibility modifiers, 344

Packages
- organizing classes in, 345
- organizing programs in, 18

Page Down key, on keyboards, 6

Page Up key, on keyboards, 6

Pair of points, algorithm for finding closest, 861–864

Palindromes
- case study: checking if string is a palindrome, 189–191
- case study: ignoring nonalphanumeric characters when checking palindromes, 398–400
- palindrome integers, 236
- palindromic primes, 243
- `RecursivePalindrome.java`, 728–729
- `RecursivePalindromeUsingSubstring.java`, 727–728

Panels
- `ButtonInPane.java`, 546
- `MessagePanel` class. *see* `MessagePanel` class

Parallel edges, 1048

Parallel execution, order of, 1140

Parallel streams
- overview of, 1139
- `ParallelStreamDemo.java` example, 1139–1141
- *vs.* sequential streams, 1140–1141

Parameters
- actual parameters, 207
- defining methods and, 206–207
- generic classes, 756
- generic methods, 758
- generic parameters not allowed in static context, 765–766
- as local variable, 224
- order association, 214
- passing by values, 214–217
- variable-length argument lists, 268–269

Parent, 545

Parentheses (`( )`)
- defining and invoking methods and, 227
- in `Welcome.java`, 14

Parsing methods, 384

Pascal, high-level languages, 8

Pass-by-sharing
- arrays to methods, 262
- objects to methods, 350

Pass-by-value
- arrays to methods, 262
- `Increment.java` example, 214

objects to methods, 349
overview of, 214
`TestPassByValue.java` example, 215–217
`PasswordField`, 655
Passwords, checking if string is valid password, 241
`PathTransition`, 619–622
Pentagonal numbers, 236
Perfect hash function, 1016
Perfectly balanced trees, 996
Pivot element, 896
Pixels (picture elements), measuring resolution in, 6
Points
algorithm for finding closest pair of, 861–864
finding convex hull for a set of points, 867–869
`Polygon` and `Polyline`
overview, 577
`ShowPolygon.java`, 578–580
Polymorphism
`CastingDemo.java` example, 430–431
overview of, 425
`PolymorphismDemo.java` example, 425
Polynomial hash codes, 1018
Postfix decrement operator, 57–58
Postfix increment operator, 57–58
Postfix notation, 812–813
Postorder traversal
time complexity of, 978
tree traversal, 963
Posttest loops, 176
`pow` method, `Math` class, 48
Precedence, operator, 106–107, 1166–1167
Prefix decrement operator, 57–58
Prefix increment operator, 57–58
Preorder traversal
time complexity of, 978
tree traversal, 963
Pretest loops, 176
Prime numbers
algorithm for finding, 855–861
case study: displaying prime numbers, 191–193
comparing prime number algorithms, 861
`EfficientPrimeNumbers.java` example, 857–859
`PrimeNumberMethod.java`, 218–219
`PrimeNumbers.java` example, 856–857
`SieveOfEratosthenes.java` example, 860–861
types of, 243
Primitive types (fundamental types)
automatic conversion between primitive types and wrapper class types, 386, 753
casting, 431
comparing parameters of primitive type with parameters of reference types, 351
comparing primitive variables with reference variables, 334–336
converting wrapper object to/from (boxing/unboxing), 386
creating arrays of, 353
hash codes for, 1017
Prim's minimum spanning tree algorithm
Dijkstra's algorithm compared to, 1105
overview of, 1097–1106
`print` method, `PrintWriter` class, 38, 480–481
`printf` method, `PrintWriter` class, 480–481
Printing arrays, 293
`println` method, `PrintWriter` class, 38, 480–481
`printStackTrace` method, 465–466
`PrintWriter` class
case study: replacing text, 485–486
writing data to file using, 480–481
for writing text data, 692
Priority queues
implementing, 953–954
`MyPriorityQueue.java` example, 953–954
overview of, 800
`PriorityQueue` class, 802
for storing weighted edges, 1102
`TestPriorityQueue.java` example, 954
`PriorityQueue` class, 802
`private`
encapsulation of data fields and, 346–347
visibility modifier, 345–346, 443–444
Problems
breaking into subproblems, 192
creating programs to address, 34
solving with recursion, 726–728
Procedural paradigm, compared with object-oriented paradigm, 374–375
Procedures, 207. *see also* Methods
Processing arrays, 253–255
Processor, 1147
Programming errors. *see also* Exception handling
`ClassCastException`, 430
debugging, 108
logic errors, 21
minimizing numeric errors related to loops, 180–181
runtime errors, 20
selections, 85–89
syntax errors, 13, 19
using generic classes for detecting, 752–754
Programming languages
assembly language, 7
high-level languages, 8–9
Java. *see* Java programming
machine language, 7
overview of, 2
Programming style
block styles, 19
comments and, 18–19
indentation and spacing, 19
overview of, 18
Programs/programming
assignment statements and expressions, 42–43
case study: counting monetary units, 64–67
case study: displaying current time, 54–55
character data type, 126–131
coding incrementally, 164
evaluating expressions and operator precedence rules, 52–54
exponent operations, 48
identifiers, 40
increment and decrement operators, 57–58
introduction to, 34
with Java language. *see* Java programming
key terms, 68
modularizing code, 217–219
named constants, 43–44
naming conventions, 44
numeric literals, 48–53
numeric operators, 46–48
numeric type conversions, 58–60
numeric types, 45
overview of, 2
questions and exercises, 70–75
reading input from console, 37–39
recursive methods in, 720
software development process, 61–64

Programs/programming (*Continued*)
  string data type, 131–140
  summary, 69–70
  variables, 40–42
  writing a simple program, 34–37
protected
  data and methods, 442–444
  visibility modifier, 345, 443–444
Protected data fields, 999
Pseudocode, 34
Public classes, 327
public method, 348
public visibility modifier, 344–346, 442–444
Python, high-level languages, 8

Q

Quadratic algorithm, 843, 848
Quadratic probing, collision handling, 1020–1021
Query methods, Map interface, 829
Query operations, Collection interface, 777
Queue interface, 800
Queues
  breadth-first search algorithm, 1074
  bucket sorts and, 908–909
  as collection type, 776
  Dequeue interface, 800–802
  GenericQueue.java example, 951
  implementing, 949–953
  overview of, 800
  priority queues. *see* Priority queues
  Queue interface, 800
  TestStackQueue.java example, 951–953
  WeightedGraph class, 1095–1102
Quick sorts
  algorithm, 895–897
  merge sorts compared with, 900
  overview of, 895
  QuickSort.java example, 897–900
Quincunx, 288
Quotients
  Quotient.java example, 454
  QuotientWithException.java example, 456–458
  QuotientWithIf.java example, 455
  QuotientWithMethod.java example, 455–456

R

Radio buttons, 651–653
RadioButtonDemo.java, 652–653
Radix sorts, 907–909
Ragged arrays, 292–293, 1052
RAM (random-access memory), 4
Random class, java.util, 337
random method
  case study: generating random numbers, 225–227
  case study: lottery, 100–101
  Math class, 89–90, 124
Random numbers
  case study: generating random numbers, 225–227
  case study: lottery, 100–101
  generating, 89–90
Random-access files
  overview of, 711–712
  TestRandomAccessFile.java, 713–714
Random-access memory (RAM), 4
Rational class
  case study: designing class for matrix using generic types, 766–767
  overview of, 526–527
  Rational.java example, 528–531
  RationalMatrix.java example, 769–770
  TestRationalClass.java example, 527–528
  TestRationalMatrix.java example, 770–771
Rational numbers, representing and processing, 526–528
Raw types, backward compatiblity and, 760–761
readASolution() method, applying to Sudoku grid, 302
Read-only streams, 711. *see also* InputStream classes
Read-only views, Collections class, 835
Rebalancing AVL trees, 996–998
Rectangle
  overview, 570–571
  ShowRectangle.java, 571–572
Recurrence relations, in analysis of algorithm complexity, 847
Recursion
  binary searches, 730
  case study: computing factorials, 720–723
  case study: computing Fibonacci numbers, 723–726
  case study: determining directory size, 731–732
  case study: fractals, 736–739
  case study: Towers of Hanoi, 33–736
  ComputeFactorial.java, 721–723
  ComputeFactorialTailRecursion.java, 741
  ComputeFibonacci.java, 724–726
  depth-first searches (DFS), 1068
  DirectorySize.java, 731–732
  displaying/visualizing binary trees, 980
  helper method, 728
  iteration compared with, 740
  key terms, 741
  overview of, 720
  problem solving by thinking recursively, 726–728
  questions and exercises, 742–750
  RecursivePalindrome.java, 728–729
  RecursivePalindromeUsingSubstring.java, 727–728
  RecursiveSelectionSort.java, 729
  selection sorts, 729
  SierpinskiTriangle.java, 736–739
  summary, 742
  tail recursion, 740–741
  TowersOfHanoi.java, 734–736
Recursive methods, 720
Red–black trees, 1016
reduce method, stream reduction using, 1141–1144
Reduction, characteristics of recursion, 726
Reference data fields, 359
Reference types
  classes as, 332
  comparing parameters of primitive type with parameters of reference types, 351
  comparing primitive variables with, 334–336
  generic types as, 752
  reference data fields, 334
  string data type as, 131
Reference variables
  accessing objects with, 332–333
  array of objects as array of, 353
  comparing primitive variables with, 334–336
Register listeners
  ControlCircle.java, 600–601
  ControlCircleWithMouseAndKey.java, 598–599, 615–616
  KeyEventDemo.java, 614–615
  LoanCalculator.java, 610–611
  overview of, 597–598

Regular expressions
- matching strings with, 391, 1176
- replacing and splitting strings, 1179–1180
- syntax, 1176–1179

Rehashing
- load factor and, 1025
- time complexity of hashing methods and, 1033

Relative file names, 477–478
Remainder (`%=`) assignment operator, 56–57
Remainder (`%`) or modulo operator, 46, 52
`remove` method, linked lists, 935, 938
Repetition
- determining Big *O* for repetition statements, 842–845
- loops. *see* Loops

`replace` method, strings, 390
`replaceAll` method, strings, 390, 1179
`replaceFirst` method, strings, 390, 1179
Requirements specification, in software development process, 61
Reserved words. *see* Keywords (reserved words)
Resources, role of OSs in allocating, 10
Responsibilities, separation as class design principle, 532
`return` statements, 209
Return value type
- constructors not having, 331
- in defining methods, 207

Reusable code
- benefits of stepwise refinement, 234
- code modularization and, 217
- method enabling, 210
- methods for, 206

`reverse` method
- applying to lists, 792
- returning arrays from methods, 262

Right subtree, of binary trees, 960
Right-heavy, balancing AVL nodes, 996, 1004
RL imbalance, AVL nodes, 997, 998
RL rotation
- `AVLTree` class, 1004, 1005
- balancing nodes on a path, 1000
- options for balancing AVL nodes, 997, 998

Root, of binary trees, 960, 961
Rotation
- `AVLTree` class, 1003–1004
- balancing nodes on a path, 1000–1001
- implementing, 1001
- methods for performing, 1007
- options for balancing AVL nodes, 996–998

Rounding methods, `Math` class, 123–124
Row index, 291
RR imbalance, AVL nodes, 996, 998
RR rotation
- `AVLTree` class, 1004, 1005
- balancing nodes on a path, 1000
- options for balancing AVL nodes, 996, 998

Runtime errors
- debugging, 108
- declaring, 461
- exception handling and, 39, 454
- `NullPointerException` as, 334
- programming errors, 21

Runtime stacks. *see* Call stacks

S

`Scanner` class
- obtaining input with, 68
- for reading console input, 37–39
- reading data from file using, 82–483
- for reading text data, 692

Scanners
- case study: replacing text, 485–486
- creating, 458

`Scene`, 542–545
Scheduling operations, 10
Scientific notation, of floating-point literals, 50
Scope, of variables, 41, 224–225
Screen resolution, 6
Scroll bars
- overview of, 665–666
- `ScrollBarDemo.java`, 666–667

Scroll panes
- `DescriptionPanel.java`, 657
- overview of, 656
- scrolling lists, 663

`search` method, `AVLTree` class, 1011
Searches
- arrays, 267
- binary search trees. *see* Binary search trees (BST)
- binary searches, 270–273, 730
- linear searches, 269–270
- recursive approach to searching for words, 720
- search keys, 1016, 1042

`SearchTree` class
- as inner class of `UnweightedGraph` class, 1062
- `MST` class extending, 1101–1102
- `ShortestPathTree` class extending, 1101
- traversing graphs and, 1067

Secondary clustering, quadratic probing issue, 1021
Segments, merging, 913
Selection sort algorithm
- analyzing, 846
- recurrence relations and, 847

Selection sorts
- arrays, 273–274
- `RecursiveSelectionSort.java`, 729
- using recursion, 729

Selection statements, 78, 80
- determining Big *O* for, 842–845

Selections
- `Addition.Quiz.java` example, 79–80
- `boolean` data type, 78–79
- case study: computing Body Mass Index, 91–92
- case study: computing taxes, 92–93
- case study: determining leap year, 99–100
- case study: guessing birthdays, 140–143
- case study: lottery, 100–101
- common errors, 85–89
- conditional operators, 105–106
- debugging, 108
- formatting output consoles, 146–150
- generating random numbers, 89–90
- `if` statements, 80–81
- `if-else` statements, 82–83
- key terms, 109
- logical operators, 95–99
- nested `if` statements and multi-way `if-else` statements, 83–85
- operator precedence and associativity, 106–107
- overview of, 78
- questions and exercises, 110–120
- summary, 109
- `switch` statements, 102–105

Semicolons (`;`), common errors, 86
Sentinel-controlled loops, 168–170

Separate chaining
  handling collision in hashing, 1023
  implementing map using hashing, 1025
Sequence statements, determining Big *O* for, 842–845
Sequential files, input/output streams, 711
Sequential streams, 1139
  parallel streams *vs.*, 1140–1141
Serialization
  of arrays, 709–711
  of objects, 709
set method, **List** interface, 783
Set operations, **Collection** interface, 777
setLength method, **StringBuilder** class, 397
setRadius method
  Circle example, 327
  CircleWithPrivateDataFields.java example, 348
Sets
  case study: counting keywords, 827–828
  as collection type, 776
  comparing list performance with, 824–826
  HashSet class, 816–820
  key terms, 836
  LinkedHashSet class, 820
  overview of, 816
  quiz and exercises, 836–838
  singleton and unmodifiable, 835
  summary, 836
  TestHashSet.java example, 817–818
  TestLinkedHashSet.java example, 820
  TestMethodsInCollection.java example, 818–819
  TestTreeSet.java example, 821
  TestTreeSetWithComparator.java example, 823–824
  TreeSet class, 820–824
Sets, implementing with hashing
  MyHashSet.java example, 1034–1041
  overview of, 1034
  TestMyHashSet.java example, 1041–1042
Setter (mutator) methods
  ArrayList class and, 438
  encapsulation of data fields and, 347–348
  implementing linked lists, 935
Seven Bridges of Königsberg problem, 1047
Shallow copies, **clone** method and, 520–521
Shapes, 545–548
  Arc, 575–577
  Circle and **Ellipse**, 572–574
  Line, 569–570
  Polygon and **Polyline**, 577–580
  Rectangle, 570–572
  Text, 567–569
Sharing code, 210
short, numeric types
  hash codes for primitive types, 1017
  overview of, 45
Short-circuit operator, 98
Shortest path tree, 1109
Shortest paths
  case study: weighted nine tails problem, 1118–1119
  Dijkstra's algorithm, 1110–1111
  finding with graph, 1046, 1049
  nine tails problem, 1077–1083
  overview of, 1105–1106
  TestShortestPath.java, 1115–1116
  WeightedGraph class and, 1095
ShortestPathTree class, 1110
Shuffling arrays, 254, 294–295
Sibling, 960
Sierpinski triangle
  case study, 736–737
  computing recursively, 744, 747, 749
  SierpinskiTriangle.java, 736–739
Sieve of Eratosthenes, 859–861
Simple graphs, 1048
sin method, trigonometry, 122–123
Single abstract method (SAM) interface, 607
Single precision numbers. *see* Floating-point numbers (**float** data type)
Single-dimensional arrays. *see* Arrays, single-dimensional
Single-source shortest-path algorithm, Dijkstra's, 1110–1111
Singly linked lists. *see* **LinkedList** class
Sinking sorts, 283, 890–892
Sliders
  overview of, 668
  SliderDemo.java, 669–670
Software
  development process, 61–64
  programs as, 2
sort method
  Arrays class, 275, 276
  ComparableRectangle.java example, 515–516
  lists and, 792
  SortRectangles.java example, 516–517
  using recursion, 730
sorted method, 1133–1134
SortedMap interface, 830, 831
Sorting
  adding nodes to heaps, 901
  arrays using heaps, 906
  bubble sort, 890–892
  bucket sorts and radix sorts, 907–909
  complexity of external sorts, 916
  complexity of heap sorts, 906–907
  CreateLargeFile.java example of external sorts, 909–910
  external sorts, 909–910
  Heap class and, 904–905
  heap sort, 900–907
  Heap.java example, 904–905
  HeapSort.java example, 906
  implementation phases of external sorts, 911–915
  insertion sorts, 888–890
  key terms, 916
  merge sorts, 892–895
  overview of, 888
  quick sort, 896–900
  quiz and exercises, 917–921
  removing root from heap, 902–903
  storing heaps, 901
  summary, 917
Sorting arrays
  bubble sorts, 283
  case study: generic method for, 758–759
  insertion sorts, 888–892
  overview of, 273
  selection sorts, 273–274
Source objects, event sources and, 596–597
Source program or source code, 8
Space complexity, 841
Spacing, programming style and, 19
Spanning trees
  graphs, 1048
  minimum spanning trees, 1099–1101
  MST algorithm, 1103–1105
  Prim's minimum spanning tree algorithm, 1103–1105

TestMinimumSpanningTree.java, 1107–1108
traversing graphs and, 1067
Special characters, 13
Specific import, 38
split method, strings, 390, 391, 1179, 1180
Stack class, 799
StackOfIntegers class, 380–381
StackOverflowError, recursion causing, 740
Stacks
case study: designing class for stacks, 380–382
case study: evaluating expressions, 803–804
EvaluateExpression.java example, 805–808
GenericStack class, 755–756
implementing, 949–953
Stack class, 799
TestStackQueue.java example, 951–953
Stage, 542, 545
State, of objects, 324
Statements
break statements, 103
continue statements, 187–188
executing one at a time, 108
executing repeatedly (loops), 160
in high-level languages, 8
if. *see* if statements
if-else. *see* if-else statements
return statements, 209
switch statements, 102–103
terminators, 12
Static methods
in Circle.java (for CircleWithStaticMembers), 340–341
class design guidelines, 533
declaring, 340
defined, 340
for lists and collections, 792–795
Stream interface, 1130
when to use instance methods *vs.* static, 341–344
wrapper classes and, 384
Static variables
in Circle.java (for CircleWithStaticMembers), 340–341
class, 339–344
class design guidelines, 533
declaring, 340
instance variables compared with, 339–341
in TestCircleWithStaticMembers.java example, 341
when to use instance variables *vs.* static, 341–343
Stepwise refinement
benefits, 234
implementation methods, 231–234
method abstraction, 227–234
top-down and/or bottom-up implementation, 229–231
top-down design, 228–231
Storage devices
CDs and DVDs, 5
Cloud storage, 5
disks, 5
overview of, 4–5
USB flash drives, 5
Storage units, for measuring memory, 3
Stream.of method, 1133
Streams, 1130
AnalyzeNumbersUsingStream.java example, 1150–1151
case study: analyzing numbers, 1150–1152
case study: counting keywords, 1155–1156
case study: counting occurrences of each letter, 1151–1152
case study: counting occurrences of each letter in string, 1152–1153
case study: finding directory size, 1154–1155
case study: occurrences of words, 1157–1158
case study: processing all elements in two-dimensional array, 1153–1154
CollectDemo.java example, 1145–1147
CollectGroupDemo.java example, 1148–1150
CountKeywordStream.java example, 1155–1156
CountLettersUsingStream.java example, 1151–1152
CountOccurrenceOfLettersInAString.java example, 1152–1153
CountOccurrenceOfWordsStream.java example, 1157–1158
DirectorySizeStream.java example, 1154–1155
DoubleStream, 1136–1139
grouping elements using groupingby collector, 1147–1150
IntStream, 1136–1139
IntStreamDemo.java example, 1136–1139
LongStream, 1136–1139
overview of, 1130
parallel streams, 1139–1141
ParallelStreamDemo.java example, 1139–1141
quiz and exercises, 1158–1159
Stream class, 1131
stream pipelines, 1130–1136
stream reduction using collect method, 1144–1147
stream reduction using reduce method, 1141–1144
StreamDemo.java example, 1132–1133
StreamReductionDemo.java example, 1142–1144
summary, 1158
TwoDimensionalArrayStream.java example, 1153–1154
String class, 388
String concatenation operator (+), 36
String literals, 388
String matching
StringMatch.java, 870
Boyer-Moore Algorithm, 870–873
String variables, 389
StringBuffer class, 395–396
StringBuilder class
case study: ignoring nonalphanumeric characters when checking palindromes, 398–400
modifying strings in, 395–397
overview of, 395
toString, capacity, length, setLength, and charAt methods, 397
Strings
in binary I/O, 698–699
case study: checking if string is a palindrome, 189–191
case study: converting hexadecimals to decimals, 219–221
case study: counting the occurrences of each letter in a string, 1152–1153
case study: ignoring nonalphanumeric characters when checking palindromes, 398–401
case study: revising the lottery program, 145–146
Character class, 189–191
command-line arguments, 276–279
concatenating, 36, 131
constructing, 388
conversion between numbers and, 138–139
converting to/from arrays, 392
finding characters or substrings in, 390–391
formatting, 392–395
generic method for sorting array of Comparable objects, 758–759
hash codes for, 1017–1018
immutable and interned, 388–390
matching, replacing, and splitting by patterns, 391, 1179–1180
overview of, 388
replacing, and splitting, 390
string data type, 131
StringBuilder and StringBuffer classes, 395–401
substrings, 37, 137
in Welcome.java, 12

Subclasses
  abstract methods and, 501
  abstracting, 504
  constructors, 418
  of Exception class, 460–461
  inheritance and, 412–418
  of `RuntimeException` class, 461
Subdirectories, 731
Subgraphs, 1048
Subinterfaces, 523
`substring` method, 136, 137, 728
Substrings, 137
Subtraction (`-=`) assignment operator, 56–57
Subtraction (`-`) operator, 46, 53
Subtrees
  of binary trees basics, 960
  searching for elements in BST, 962
Sudoku puzzle, 300–303, 884–885
`sum` method, 206, 1137, 1138
`super` keyword, 418
Superclass methods, 420–421
Superclasses
  of abstract class can be concrete, 504
  classes extending, 523
  inheritance and, 412–418
  subclasses related to, 501
Supplementary characters, Unicode, 127
`swap` method
  swapping elements in an array, 263–264
  in `TestPassByValue.java` example, 215, 216
`switch` statements
  `ChineseZodiac.java` example, 104–105
  with enumerated types, 1183
  overview of, 102–103
Syntax errors (compile errors)
  common errors, 13, 14
  debugging, 108
  programming errors, 19
Syntax rules, in `Welcome.java`, 13–14
System activities, role of OSs, 10
System analysis, in software development process, 61–63
System design, in software development process, 61, 63
System errors, 460
`System.in`, 37
`System.out`, 37, 146–150

T

Tables, storing, 290
Tail recursion
  `ComputeFactorialTailRecusion.java`, 741
  overview of, 740–741
`tan` method, trigonometry, 122–123
TBs (terabytes), of storage, 4–5
Teamwork, facilitated by stepwise refinement, 234
Terabytes (TBs), of storage, 4–5
Terminal method, `Stream` interface, 1130
Testing
  benefits of stepwise refinement, 234
  in software development process, 62, 64
`TestShortestPath.java`, 1115–1116
Text
  case study: replacing text, 485–487
  files, 692
  overview, 567
  `ShowText.java`, 568–569
  `TextArea`, 655–658
  `TextAreaDemo.java`, 658
  `TextField`, 654–655
  `TextFieldDemo.java`, 654–655
  .txt files (text), 694
Text I/O
  vs. binary I/O, 693–694
  handling in Java, 692–693
  overview of, 692
TextPad, for creating/editing Java source code, 12
`thenComparing` method, 791
`this` reference
  invoking constructors with, 360
  overview of, 358
  referencing data fields with, 359
Three-dimensional arrays. *see* Arrays, multi-dimensional
`throw` keyword
  chained exceptions, 473–474
  `throw ex` for rethrowing exceptions, 473
  for throwing exceptions, 467
`Throwable` class
  generic classes not extending, 766
  getting information about exceptions, 465
  `java.lang`, 459–461
Throwing exceptions, 462–463, 467–470
  `CircleWithException.java` example, 467
  `QuotientWithException.java` example, 456–458
  rethrowing, 472–473
  `TestCircleWithCustomException.java` example, 476
  `throw` keyword for, 463
`throws` keyword
  chained exceptions, 473–474
  for declaring exceptions, 462, 467
  `IOException`, 695, 696
  for throwing exceptions, 463
Tic-tac-toe game, 310
Time complexity, 841
  AVL trees, 1011
  `BST` class, 978
  bubble sort, 892
  heap sorts, 906–907
  insertion sorts, 890
  merge sorts, 895
  rehashing, 1033
`toArray` method, 1132, 1135–1136
`toCharArray` method, converting strings into arrays, 392
`ToggleButton`, 651
`ToggleGroup`, 652
Token reading methods, `Scanner` class, 484–485
Top-down design, 228–231
Top-down implementation, 229–231
`toString` method
  `ArrayList` class, 437
  `Arrays` class, 276
  `Date` class, 337
  implementing `MyLinkedList`, 947
  `MyArrayList.java` example, 932, 933
  `Object` class, 433
  `StringBuilder` class, 397
`total` variable, for storing sums, 294
Touchscreens, 6
Towers of Hanoi problem
  analyzing algorithm for, 846–847
  computing recursively, 744
  nonrecursive computation, 814
  recurrence relations and, 847

Tracing a program, 36
`transient` keyword, serialization and, 709
Transistors, CPUs, 3
Traveling salesperson problem (TSP), 1123
Traversing binary search trees, 963–965
Traversing graphs
  breadth-first searches (BFS), 1074–1077
  case study: connected circles problem, 1072–1076
  depth-first searches (DFS), 1068–1072
  overview of, 1068
  `TestWeightedGraph.java`, 1100
`Tree` interface, `BST` class, 965
Tree traversal, 963–965
`TreeMap` class
  case study: counting occurrence of words, 833–834
  concrete implementation of `Map` class, 828–830
  implementation of `Map` class, 1016
  overview of, 831
  `TestMap.java` example, 831–833
  types of maps, 828–829
Trees
  AVL trees. *see* AVL trees
  binary search. *see* Binary search trees (BST)
  connected graphs, 1048
  creating BFS trees, 1075
  Huffman coding. *see* Huffman coding trees
  overview of, 960
  Red–black trees, 1016
  spanning trees. *see* Spanning trees
  traversing, 963–965
`TreeSet` class
  implementation of `Set` class, 1034
  overview of, 820–821
  `TestTreeSet.java` class, 821
  `TestTreeSetWithComparator.java` class, 821–823
  `TestTreeSetWithComparator.java` example, 508–510
  types of sets, 816
Trigonometric methods, `Math` class, 122–123
`trimToSize` method, 932
True/false (Boolean) values, 78
Truth tables, 95–96
`try-catch` blocks
  catching exceptions, 461, 463–465
  chained exceptions, 473–474
  exception classes cannot be generic, 766
  `InputMismatchExceptionDemo.java` example, 458–459
  `QuotientWithMethod.java` example, 455–456
  rethrowing exceptions, 472–473
  `TestCircleWithException.java` example, 468–470
  when to use exceptions, 472–473
Twin primes, 244
Two-dimensional arrays. *see* Arrays, two-dimensional
Type casting
  between `char` and numeric types, 128
  generic types and, 754
  loss of precision, 66
  for numeric type conversion, 58–59
Type erasure, erasing generic types, 764–765

U

UML (Unified Modeling Language)
  aggregation shown in, 376
  class diagrams with, 325
  diagram for `Loan` class, 369
  diagram of `StackOfIntegers`, 380
  diagram of static variables and methods, 339–340
Unary operators, 48
Unbounded wildcards, 762
Unboxing, 386
Unchecked exceptions, 461
Unconditional AND operator, 98
Underflow, floating point numbers, 67
Undirected graphs, 1047
Unicode
  character data type (`char`) and, 126–130
  data input and output streams, 698–699
  generating random numbers and, 225
  text encoding, 692
  text I/O *vs.* binary I/O, 693–694
Unified Modeling Language. *see* UML (Unified Modeling Language)
Uniform Resource Locators. *see* URLs (Uniform Resource Locators)
Unique addresses, for each byte of memory, 4
Universal serial bus (USB) flash drives, 5
UNIX epoch, 54
Unweighted graphs
  defined, 1048
  modeling graphs and, 1048, 1055
  `UnweightedGraph.java` example, 1058–1064
Upcasting objects, 429
Update methods, `Map` interface, 829
`URL` class, `java.net`, 487
URLs (Uniform Resource Locators)
  `ReadFileFromURL.java` example, 487–488
  reading data from Web, 487–488
USB (universal serial bus) flash drives, 5
UTF, 699. *see also* Unicode

V

`valueOf` methods
  converting strings into arrays, 391
  wrapper classes and, 384
Value-returning methods
  `return` statements required by, 209
  `TestReturnGradeMethod.java`, 211–213
  `void` method and, 207
Values
  hashing functions, 1016
  maps and, 1042
`values` method, `Map` interface, 830
Variable-length argument lists, 268–269
Variables
  Boolean variables. *see* Boolean variables
  comparing primitive variables with reference variables, 334–336
  control variables, in for loops, 173–174
  declaring, 35–36, 41
  declaring array variables, 250
  declaring for two-dimensional arrays, 290–291
  displaying/modifying, 108
  hidden, 357
  identifiers, 40
  naming conventions, 44
  overflow, 67
  overview of, 40–42
  reference variables, 332–333
  scope of, 41, 225–226, 357–358
  static variables, 339–344
`Vector` class
  methods, 798–799
  overview of, 798
  `Stack` class extending, 800
Vertex-weighted graphs, 1093

Vertical scroll bars, 666
Vertical sliders, 668, 669
Vertices
  adjacent and incident, 1048
  depth-first searches (DFS), 1068
  `Graph.java` example, 1056
  on graphs, 1048
  Prim's algorithm and, 1099
  representing on graphs, 1048–1050
  shortest paths. *see* Shortest paths
  `TestBFS.java`, 1076
  `TestGraph.java` example, 1056
  `TestMinimumSpanningTree.java`, 1107
  `TestWeightedGraph.java`, 1100
  vertex-weighted graphs, 1093
  weighted adjacency matrices, 1094
`WeightedGraph` class, 1091–1092
Video, `MediaDemo.java`, 677–679
Virtual machines (VMs), 16. *see also* JVM (Java Virtual Machine)
Visibility modifiers, 1169
Visibility (accessibility) modifiers
  classes and, 344–346
  `protected, public`, and `private`, 442–444
Visual Basic, high-level languages, 8
Visualizing (displaying) graphs
  `Displayable.java` example, 1064
  `DisplayUSMap.java` example, 1066–1067
  `GraphView.java` example, 1064–1065
  overview of, 1064
VLSI (very large-scale integration), 720
VMs (virtual machines), 15. *see also* JVM (Java Virtual Machine)
`void` method
  defined, 207
  defining and invoking, 211
  `TestVoidMethod.java`, 211

W

Web, reading file data from, 487–488
Weighted graphs
  case study: weighted nine tails problem, 840–843
  defined, 1048
  Dijkstra's single-source shortest-path algorithm, 1109–1111
  `getMinimumSpanningTree` method, 1103
  key terms, 843
  minimum spanning trees, 1103
  modeling graphs and, 1054
  MST algorithm, 1103–1104
  overview of, 1046–1047
  Prim's minimum spanning tree algorithm, 1103–1105
  priority adjacency lists, 1103–1104
  questions and exercises, 1112–1128
  representing, 1093
  shortest paths, 1109
  summary, 1122
  `TestMinimumSpanningTree.java`, 1107–1109
  `TestShortestPath.java`, 1115–1118
  `TestWeightedGraph.java`, 1100–1102
  weighted adjacency matrices, 1094
  weighted edges using edge array, 1093–1094
  `WeightedGraph` class, 1095–1096
  `WeightedGraph.java`, 1096–100
`WeightedEdge` class, 1088
`WeightedGraph` class
  `getMinimumSpanningTree` method, 1103–1104
  overview of, 1095
  `ShortestPathTree` class as inner class of, 1114
  `TestWeightedGraph.java`, 1100–1101
  `WeightedGraph.java`, 1096–1100
Well-balanced trees
  AVL trees, 996
  binary search trees, 1016
`while` loops
  case study: guessing numbers, 163–166
  case study: multiple subtraction quiz, 166–168
  case study: predicting future tuition, 183
  deciding when to use, 176–178
  design strategies, 166
  `do-while` loops. *see* `do-while` loops
  input and output redirections, 169–170
  overview of, 160–161
  `RepeatAdditionQuiz.java` example, 162–163
  sentinel-controlled, 168–170
  syntax of, 160
Whitespace
  characters, 134
  as delimiter in token reading methods, 484
Wildcard import, 38
Wildcards, for specifying range of generic types, 761–764
Windows. *see* Frames (windows)
Windows OSs, 9
Wireless networking, 6
Worst-case input
  heap sorts and, 906
  measuring algorithm efficiency, 840, 854
  quick sort and, 899
Wrapper classes
  automatic conversion between primitive types and wrapper class types, 753
  `File` class as, 47
  numeric, 526
  primitive types and, 382–386
Wrapping lines of text or words, 656, 658
Write-only streams, 711. *see also* `OutputStream` classes

X

`Xlint:unchecked` error, compile time errors, 760