Basic Elements, Primitive Data Types, and Operators

2

Chapter Topics

- Overview of basic language elements in Java: identifiers, keywords, separators, literals, whitespace, and comments
- Overview of primitive data types defined in Java: integral, floating-point, and boolean
- Representing integers in different number systems and in memory
- Understanding type conversion categories and conversion contexts, and which conversions are permissible in each conversion context
- Defining and evaluating arithmetic and boolean expressions, and the order in which operands and operators are evaluated
- Using Java operators, including precedence and associativity rules for expression evaluation

Java SE 17 Developer Exam Objectives	
[1.1] Use primitives and wrapper classes including Math API, parentheses, type promotion, and casting to evaluate arithmetic and boolean expressions ❍ *Primitive types, operators, expression evaluation, and type conversions are covered in this chapter.* ❍ *For wrapper classes, see §8.3, p. 439.* ❍ *For Math API, see §8.6, p. 488.*	*§2.2, p. 41 to §2.19, p. 93*
Java SE 11 Developer Exam Objectives	
[1.1] Use primitives and wrapper classes, including, operators, the use of parentheses, type promotion and casting ❍ *Primitive types, operators, expression evaluation, and type conversions are covered in this chapter.* ❍ *For wrapper classes, see §8.3, p. 439.*	*§2.2, p. 41 to §2.19, p. 93*

This chapter covers the low-level language elements from which high-level constructs are formed, the primitive data types that are provided by the language, and the operators that can be used to compose expressions. In addition to how expressions are evaluated, an understanding of which type conversions can be applied in which context is also essential.

2.1 Basic Language Elements

Like any other programming language, the Java programming language is defined by *grammar rules* that specify how *syntactically* legal constructs can be formed using the language elements, and by a *semantic definition* that specifies the *meaning* of syntactically legal constructs.

Lexical Tokens

The low-level language elements are called *lexical tokens* (or just *tokens*) and are the building blocks for more complex constructs. Identifiers, numbers, operators, and special characters are all examples of tokens that can be used to build high-level constructs like expressions, statements, methods, and classes.

Identifiers

A name in a program is called an *identifier*. Identifiers can be used to denote classes, methods, variables, and labels.

In Java, an *identifier* is composed of a sequence of characters, where each character can be either a *letter* or a *digit*. However, the first character in an identifier must always be a letter, as explained later.

Since Java programs are written in the Unicode character set (p. 37), characters allowed in identifier names are interpreted according to this character set. Use of the Unicode character set opens up the possibility of writing identifier names in many writing scripts used around the world. As one would expect, the characters `A` to `Z` and `a` to `z` are letters and the characters `0` to `9` are digits. A *connecting punctuation character* (such as *underscore* _) and any *currency symbol* (such as `$`, `¢`, `¥`, or `£`) are also allowed as letters in identifier names, but these characters should be used judiciously. Note also that the underscore (_) on its own is *not* a legal identifier name, but a keyword (Table 2.1, p. 31).

Identifiers in Java are *case sensitive*. For example, `price` and `Price` are two different identifiers.

Examples of Legal Identifiers

```
number, Number, sum_$, bingo, $$_100, _007, mål, grüß
```

Examples of Illegal Identifiers

```
48chevy, all@hands, grand-sum, _
```

The name `48chevy` is not a legal identifier because it starts with a digit. The character `@` is not a legal character in an identifier. It is also not a legal operator, so `all@hands` cannot be interpreted as a legal expression with two operands. The character `-` is not a legal character in an identifier, but it is a legal operator; thus `grand-sum` could be interpreted as a legal expression with two operands. An underscore (_) by itself is not a legal identifier.

Keywords

Keywords are reserved words or identifiers that are predefined in the language and cannot be used to denote other entities. All Java keywords are lowercase, and incorrect usage results in compile-time errors.

Keywords currently defined in the language are listed in Table 2.1. The keyword `strictfp` is obsolete as of Java SE 17, and its use is discouraged in new code. *Contextual keywords* that are restricted in certain contexts are listed in Table 2.2. Keywords currently reserved, but *not in use*, are listed in Table 2.3. In addition, three identifiers are reserved as predefined *literals* in the language: the `null` literal, and the boolean literals `true` and `false` (Table 2.4). A keyword cannot be used as an identifier. A contextual keyword cannot be used as an identifier in certain contexts. The index at the end of the book contains references to relevant sections where currently used keywords are explained.

Table 2.1 *Keywords in Java*

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

Table 2.2 *Contextual Keywords*

exports	opens	requires	uses
module	permits	sealed	var
non-sealed	provides	to	with

Table 2.2 *Contextual Keywords (Continued)*

`open`	`record`	`transitive`	`yield`

Table 2.3 *Reserved Keywords Not Currently in Use*

`const`	`goto`

Table 2.4 *Reserved Literals in Java*

`null`	`true`	`false`

Separators

Separators (also known as *punctuators*) are tokens that have meaning depending on the context in which they are used; they aid the compiler in performing syntax and semantic analysis of a program (Table 2.5). The semicolon (`;`) is used to terminate a statement. A pair of curly brackets, `{}`, can be used to group several statements. See the index entries for these separators for more details.

Table 2.5 *Separators in Java*

`{`	`}`	`[`	`]`	`(`	`)`
`.`	`;`	`,`	`...`	`@`	`::`

Literals

A *literal* denotes a constant value; in other words, the value that a literal represents remains unchanged in the program. Literals represent numerical (integer or floating-point), character, boolean, and string values. In addition, the literal `null` represents the null reference. Table 2.6 shows examples of literals in Java.

Table 2.6 *Examples of Literals*

Integer	`2000`	`0`	`-7`		
Floating-point	`3.14`	`-3.14`	`.5`	`0.5`	
Character	`'a'`	`'A'`	`'0'`	`':'`	`'-'` `')'`
Boolean	`true`	`false`			
String	`"abba"`	`"3.14"`	`"for"`	`"a piece of the action"`	

Integer Literals

Integer data types comprise the following primitive data types: `int`, `long`, `byte`, and `short` (p. 41).

The default data type of an integer literal is always `int`, but it can be specified as `long` by appending the suffix `L` (or `l`) to the integer value. The suffix `L` is often preferred because the suffix `l` and the digit `1` can be hard to distinguish. Without the suffix, the `long` literals `2020L` and `0L` will be interpreted as `int` literals. There is no direct way to specify a `short` or a `byte` literal.

In addition to the decimal number system, integer literals can be specified in the binary (*base* 2, *digits* `0`–`1`), octal (*base* 8, *digits* `0`–`7`), and hexadecimal (*base* 16, *digits* `0`–`9` and `a`–`f`) number systems. The digits `a` to `f` in the hexadecimal system correspond to decimal values `10` to `15`. Binary, octal, and hexadecimal numbers are specified with `0b` (or `0B`), `0`, and `0x` (or `0X`) as the base or radix prefix, respectively.

Examples of decimal, binary, octal, and hexadecimal literals are shown in Table 2.7. Note that the leading `0` (zero) digit is not the uppercase letter `O`. The hexadecimal digits from `a` to `f` can also be specified with the corresponding uppercase forms (`A`–`F`). Negative integers (e.g., `-90`) can be specified by prefixing the minus sign (`-`) to the magnitude of the integer regardless of the number system (e.g., `-0b1011010`, `-0132`, or `-0X5a`).

Table 2.7 *Examples of Decimal, Binary, Octal, and Hexadecimal Literals*

Decimal	Binary	Octal	Hexadecimal
8	0b1000	010	0x8
10L	0b1010L	012L	0xaL
16	0b10000	020	0x10
27	0b11011	033	0x1b
90L	0b1011010L	0132L	0x5aL
-90	-0b1011010 or 0b1111111111111111111111 1110100110	-0132 or 037777777646	-0x5a or 0xffffffa6
-1	-0b1 or 0b1111111111111111111111 1111111111	-01 or 037777777777	-0x1 or 0xffffffff
2147483647 (i.e., $2^{31} - 1$)	0b0111111111111111111111 1111111111	017777777777	0x7fffffff
-2147483648 (i.e., -2^{31})	0b1000000000000000000000 0000000000	020000000000	0x80000000
1125899906842624L (i.e., 2^{50})	0b1000000000000000000000 0000000000000000000000 00000L	040000000000000000L	0x4000000000000L

Representing Integers

Integer data types in Java represent *signed* integer values, meaning both positive and negative integer values. The values of type `char` can effectively be regarded as *unsigned* 16-bit integers.

Values of type `byte` are represented as shown in Table 2.8. A value of type `byte` requires 8 bits. With 8 bits, we can represent 2^8 or 256 values. Java uses two's complement (explained later) to store signed values of integer data types. For the `byte` data type, this means values are in the range –128 (i.e., -2^7) to +127 (i.e., $2^7 - 1$), inclusive.

Bits in an integral value are usually numbered from right to left, starting with the least significant bit `0` (also called the *rightmost bit*). The representation of the signed types sets the most significant bit to `1`, indicating negative values. Adding `1` to the maximum `int` value `2147483647` results in the minimum value `-2147483648`, such that the values wrap around for integers and no overflow or underflow is indicated.

Table 2.8 *Representing Signed* `byte` *Values Using Two's Complement*

Decimal value	Binary representation (8 bit)	Binary value with prefix `0b`	Octal value with prefix `0`	Hexadecimal value with prefix `0x`
127	01111111	0b1111111	0177	0x7f
126	01111110	0b1111110	0176	0x7e
...	...	...	...	...
41	00101001	0b101001	051	0x29
...	...	...	...	...
2	00000010	0b10	02	0x2
1	00000001	0b1	01	0x1
0	00000000	0b0	00	0x0
–1	11111111	0b11111111	0377	0xff
–2	11111110	0b11111110	0376	0xfe
...	...	...	...	...
–41	11010111	0b11010111	0327	0xd7
...	...	...	...	...
–127	10000001	0b10000001	0201	0x81
–128	10000000	0b10000000	0200	0x80

Calculating Two's Complement

Before we look at two's complement, we need to understand one's complement. The one's complement of a binary integer is computed by inverting the bits in the number. Thus the one's complement of the binary number 00101001 is 11010110. The one's complement of a binary number N_2 is denoted as $\sim N_2$. The following relations hold between a binary integer N_2, its one's complement $\sim N_2$, and its two's complement $-N_2$:

$$-N_2 = \sim N_2 + 1$$

$$0 = -N_2 + N_2$$

If N_2 is a positive binary integer, then $-N_2$ denotes its negative binary value, and vice versa. The second relation states that adding a binary integer N_2 to its two's complement $-N_2$ equals 0.

Given a positive `byte` value, say `41`, the binary representation of `-41` can be found as follows:

	Binary representation	Decimal value
Given a value, N_2:	00101001	41
Form one's complement, $\sim N_2$:	11010110	
Add 1:	00000001	
Result is two's complement, $-N_2$:	11010111	–41

Adding a number N_2 to its two's complement $-N_2$ gives 0, and the carry bit from the addition of the most significant bits (after any necessary extension of the operands) is ignored:

	Binary representation	Decimal value
Given a value, N_2:	00101001	41
Add two's complement, $-N_2$:	11010111	–41
Sum:	00000000	0

Subtraction between two integers is also computed as addition with two's complement:

$$N_2 - M_2 = N_2 + (-M_2)$$

For example, the expression $41_{10} - 3_{10}$ (with the correct result 38_{10}) is computed as follows:

	Binary representation	Decimal value
Given a value, N_2:	00101001	41
Add $-M_2$ (i.e., subtract M_2):	11111101	–3
Result:	00100110	38

The previous discussion of `byte` values applies equally to values of other integer types: `short`, `int`, and `long`. These types have their values represented by two's complement in 16, 32, and 64 bits, respectively.

Floating-Point Literals

Floating-point data types come in two flavors: `float` and `double`.

The default data type of a floating-point literal is `double`, but it can be explicitly designated by appending the suffix `D` (or `d`) to the value. A floating-point literal can also be specified to be a `float` by appending the suffix `F` (or `f`).

Floating-point literals can also be specified in scientific notation, where `E` (or `e`) stands for *exponent*. For example, the `double` literal `194.9E-2` in scientific notation is interpreted as 194.9×10^{-2} (i.e., `1.949`).

Examples of `double` *Literals*

```
0.0       0.0d       0D
0.49      .49        .49D
49.0      49.        49D
4.9E+1    4.9E+1D    4.9e1d   4900e-2  .49E2
```

Examples of `float` *Literals*

```
0.0F      0f
0.49F     .49F
49.0F     49.F       49F
4.9E+1F   4900e-2f   .49E2F
```

Note that the decimal point and the exponent are optional, and that at least one digit must be specified. Also, for the examples of `float` literals presented here, the suffix `F` is mandatory; if it were omitted, they would be interpreted as `double` literals.

Underscores in Numerical Literals

The underscore character (`_`) can be used to improve the readability of numerical literals in the source code. Any number of underscores can be inserted *between the digits* that make up the numerical literal. This rules out underscores adjacent to the sign (`+`, `-`), the radix prefix (`0b`, `0B`, `0x`, `0X`), the decimal point (`.`), the exponent (`e`, `E`), and the data type suffix (`l`, `L`, `d`, `D`, `f`, `F`), as well as before the first digit and after the

last digit. Note that octal radix prefix `0` is part of the definition of an octal literal and is therefore considered the first digit of an octal literal.

Underscores in identifiers are treated as letters. For example, the names `_XXL` and `_XXL_` are two distinct legal identifiers. In contrast, underscores are used as a notational convenience for numerical literals and are ignored by the compiler when used in such literals. In other words, a numerical literal can be specified in the source code using underscores between digits, such that `2_0_2_2` and `20__22` represent the same numerical literal `2022` in source code.

Examples of Legal Use of Underscores in Numerical Literals

```
0b0111_1111_1111_1111_1111_1111_1111_1111
0_377_777_777              0xff_ff_ff_ff
-123_456.00                1_2.345_678e1_2
2009__08__13               49_03_01d
```

Examples of Illegal Use of Underscores in Numerical Literals

```
_0_b_01111111111111111111111111111111_
_0377777777_               _0_x_ffffffff_
+_123456_._00_             _12_._345678_e_12_
_20090813_                 _490301_d_
```

Boolean Literals

The primitive data type `boolean` represents the truth values *true* and *false* that are denoted by the reserved literals `true` and `false`, respectively.

Character Literals

A character literal is quoted in single quotes (`'`). All character literals have the primitive data type `char`.

A character literal is represented according to the 16-bit Unicode character set, which subsumes the 8-bit ISO Latin-1 and the 7-bit ASCII characters. In Table 2.9, note that digits (`0`–`9`), uppercase letters (`A`–`Z`), and lowercase letters (`a`–`z`) have contiguous Unicode values. A Unicode character can always be specified as a four-digit hexadecimal number (i.e., 16 bits) with the prefix `\u`.

Table 2.9 *Examples of Character Literals*

Character literal	Character literal using Unicode value	Character
`' '`	`'\u0020'`	*Space*
`'0'`	`'\u0030'`	0
`'1'`	`'\u0031'`	1
`'9'`	`'\u0039'`	9

Table 2.9 *Examples of Character Literals (Continued)*

Character literal	Character literal using Unicode value	Character
'A'	'\u0041'	A
'B'	'\u0042'	B
'Z'	'\u005a'	Z
'a'	'\u0061'	a
'b'	'\u0062'	b
'z'	'\u007a'	z
'Ñ'	'\u0084'	Ñ
'å'	'\u008c'	å
'ß'	'\u00a7'	ß

Escape Sequences

Certain *escape sequences* define special characters, as shown in Table 2.10. These escape sequences allow representation of some special characters in character literals, string literals (p. 39), and text blocks (§8.4, p. 468). These escape sequences can be single-quoted to define character literals, or included in string literals and text blocks. For example, the escape sequence `\t` and the Unicode value `\u0009` are equivalent. However, the Unicode values `\u000a` and `\u000d` should not be used to represent a newline and a carriage return in the source code. These values are interpreted as line-terminator characters by the compiler and will cause compile-time errors. You should use the escape sequences `\n` and `\r`, respectively, for correct interpretation of these characters in the source code.

Table 2.10 *Escape Sequences*

Escape sequence	Unicode value	Character
\b	\u0008	Backspace (BS)
\t	\u0009	Horizontal tab (HT or TAB)
\n	\u000a	Linefeed (LF), also known as newline (NL)
\f	\u000c	Form feed (FF)
\r	\u000d	Carriage return (CR)
\s	\u0020	Space (SP)
Line terminator	–	Line continuation in a text block
\'	\u0027	Apostrophe-quote, also known as single quote
\"	\u0022	Quotation mark, also known as double quote
\\	\u005c	Backslash

We can also use the escape sequence \ddd to specify a character literal as an octal value, where each digit d can be any octal digit (0–7), as shown in Table 2.11. The number of digits must be three or fewer, and the octal value cannot exceed \377; in other words, only the first 256 characters can be specified with this notation.

Table 2.11 *Examples of Escape Sequence* \ddd

Escape sequence \ddd	Character literal
'\141'	'a'
'\46'	'&'
'\60'	'0'

String Literals

A *string literal* is a sequence of characters that must be enclosed in double quotes and must occur on a single line. All string literals are objects of the class String (§8.4, p. 449).

Escape sequences as well as Unicode values can appear in string literals:

```
"Here comes a tab.\t And here comes another one\u0009!"          (1)
"What's on the menu?"                                            (2)
"\"String literals are double-quoted.\""                         (3)
"Left!\nRight!"                                                  (4)
"Don't split                                                     (5)
me up!"
```

In (1), the tab character is specified using the escape sequence and the Unicode value, respectively. In (2), the single quote need not be escaped in strings, but it would be if specified as a character literal ('\''). In (3), the double quotes in the string must be escaped. In (4), we use the escape sequence \n to insert a newline. The expression in (5) generates a compile-time error, as the string literal is split over several lines. Printing the strings from (1) to (4) will give the following result:

```
Here comes a tab.    And here comes another one    !
What's on the menu?
"String literals are double-quoted."
Left!
Right!
```

One should also use the escape sequences \n and \r, respectively, for correct interpretation of the characters \u000a (newline) and \u000d (form feed) in string literals.

Whitespace

Whitespace is a sequence of spaces, tabs, form feeds, and line terminator characters in a Java source file. Line terminators include the newline, carriage return, and carriage return–newline sequence.

A Java program is a free-format sequence of characters that is *tokenized* by the compiler—that is, broken into a stream of tokens for further analysis. Separators and operators help to distinguish tokens, but sometimes whitespace has to be inserted explicitly as a separator. For example, the identifier `classRoom` will be interpreted as a single token, unless whitespace is inserted to distinguish the keyword `class` from the identifier `Room`.

Whitespace aids not only in separating tokens, but also in formatting the program so that it is easy to read. The compiler ignores the whitespace once the tokens are identified.

Comments

A program can be documented by inserting comments at relevant places in the source code. These comments are for documentation purposes only and are ignored by the compiler.

Java provides three types of comments that can be used to document a program:

- A single-line comment: `// ... to the end of the line`
- A multiple-line comment: `/* ... */`
- A documentation (Javadoc) comment: `/** ... */'`

Single-Line Comment

All characters after the comment-start sequence `//` through to the end of the line constitute a *single-line comment*.

```
// This comment ends at the end of this line.
int age;       // From comment-start sequence to the end of the line is a comment.
```

Multiple-Line Comment

A *multiple-line comment*, as the name suggests, can span several lines. Such a comment starts with the sequence `/*` and ends with the sequence `*/`.

```
/* A comment
   on several
   lines.
*/
```

The comment-start sequences (//, /*, /**) are not treated differently from other characters when occurring within comments, so they are ignored. This means that trying to nest multiple-line comments will result in a compile-time error:

```
/* Formula for alchemy.
   gold = wizard.makeGold(stone);
   /* But it only works on Sundays. */
*/
```

The second occurrence of the comment-start sequence /* is ignored. The last occurrence of the sequence */ in the code is now unmatched, resulting in a syntax error.

Documentation Comment

A *documentation comment* is a special-purpose multiple-line comment that is used by the `javadoc` tool to generate HTML documentation for the program. Documentation comments are usually placed in front of classes, interfaces, methods, and field definitions. Special tags can be used inside a documentation comment to provide more specific information. Such a comment starts with the sequence /** and ends with the sequence */:

```
/**
 *  This class implements a gizmo.
 *  @author K.A.M.
 *  @version 4.0
 */
```

For details on the `javadoc` tool, see the tools documentation provided by the JDK.

2.2 Primitive Data Types

Figure 2.1 gives an overview of the *primitive data types* in Java.

Primitive data types in Java can be divided into three main categories:

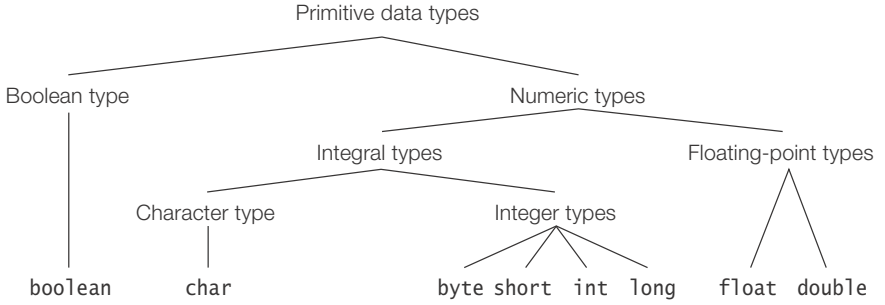
- *Integral types*: represent signed integers (`byte`, `short`, `int`, `long`) and unsigned character values (`char`)
- *Floating-point types* (`float`, `double`): represent fractional signed numbers
- *Boolean type* (`boolean`): represents logical values

Each primitive data type defines the range of values in the data type, and operations on these values are defined by special operators in the language (p. 51).

Primitive data values are not objects, but each primitive data type has a corresponding *wrapper* class that can be used to represent a primitive value as an object. Wrapper classes are discussed in §8.3, p. 439.

The Integer Types

Figure 2.1 *Primitive Data Types in Java*



The integer data types are `byte`, `short`, `int`, and `long` (Table 2.12). Their values are signed integers represented by two's complement (p. 35).

Table 2.12 *Range of Integer Values*

Data type	Width (bits)	Minimum value `MIN_VALUE`	Maximum value `MAX_VALUE`
`byte`	8	-2^7 (-128)	$2^7 - 1$ (+127)
`short`	16	-2^{15} (-32768)	$2^{15} - 1$ (+32767)
`int`	32	-2^{31} (-2147483648)	$2^{31} - 1$ (+2147483647)
`long`	64	-2^{63} (-9223372036854775808L)	$2^{63} - 1$ (+9223372036854775807L)

The `char` Type

The data type `char` represents characters (Table 2.13). Their values are unsigned integers that denote all of the 65,536 (2^{16}) characters in the 16-bit Unicode character set. This set includes letters, digits, and special characters.

Table 2.13 *Range of Character Values*

Data type	Width (bits)	Minimum Unicode value	Maximum Unicode value
`char`	16	0x0 (\u0000)	0xffff (\uffff)

The first 128 characters of the Unicode set are the same as the 128 characters of the 7-bit ASCII character set, and the first 256 characters of the Unicode set correspond to the 256 characters of the 8-bit ISO Latin-1 character set.

The integer types and the `char` type are collectively called *integral types*.

The Floating-Point Types

Floating-point numbers are represented by the `float` and `double` data types.

Floating-point numbers conform to the IEEE 754-1985 binary floating-point standard. Table 2.14 shows the range of values for positive floating-point numbers, but these apply equally to negative floating-point numbers with the minus sign (-) as a prefix. Zero can be either `0.0` or `-0.0`. The range of values represented by the `double` data type is wider than that of the `float` data type.

Table 2.14 *Range of Floating-Point Values*

Data type	Width (bits)	Minimum positive value `MIN_VALUE`	Maximum positive value `MAX_VALUE`
`float`	32	`1.401298464324817E-45f`	`3.402823476638528860e+38f`
`double`	64	`4.94065645841246544e-324`	`1.79769313486231570e+308`

Since the size for representation is a finite number of bits, certain floating-point numbers can be represented only as approximations. For example, the value of the expression `(1.0/3.0)` is represented as an approximation due to the finite number of bits used to represent floating-point numbers.

The `boolean` Type

The data type `boolean` represents the two logical values denoted by the literals `true` and `false` (Table 2.15).

Table 2.15 *Boolean Values*

Data type	Width	True value literal	False value literal
`boolean`	Not applicable	`true`	`false`

Boolean values are results of all *relational* (p. 75), *conditional* (p. 81), and *boolean* (p. 79) *logical operators*.

Table 2.16 summarizes the pertinent facts about the primitive data types: their width or size, which indicates the number of bits required to store a primitive value; their range of legal values, which is specified by the minimum and maximum values permissible; and the name of the corresponding wrapper class (§8.3, p. 439).

Table 2.16 *Summary of Primitive Data Types*

Data type	Width (bits)	Minimum value, maximum value	Wrapper class
`boolean`	Not applicable	`true, false`	`Boolean`
`byte`	8	-2^7, $2^7 - 1$	`Byte`

Table 2.16 *Summary of Primitive Data Types (Continued)*

Data type	Width (bits)	Minimum value, maximum value	Wrapper class
`short`	16	-2^{15}, $2^{15}-1$	`Short`
`char`	16	`0x0`, `0xffff`	`Character`
`int`	32	-2^{31}, $2^{31}-1$	`Integer`
`long`	64	-2^{63}, $2^{63}-1$	`Long`
`float`	32	±1.40129846432481707e-45f, ±3.40282347663852886e+38f	`Float`
`double`	64	±4.94065645841246544e-324, ±1.79769313486231570e+308	`Double`

2.3 Conversions

In this section we discuss the different kinds of *type conversions* that can be applied to values; in the next section we discuss the *contexts* in which these conversions are permitted. Some type conversions must be *explicitly* stated in the program, while others are performed *implicitly*. Some type conversions can be checked at compile time to guarantee their validity at runtime, while others will require an extra check at runtime.

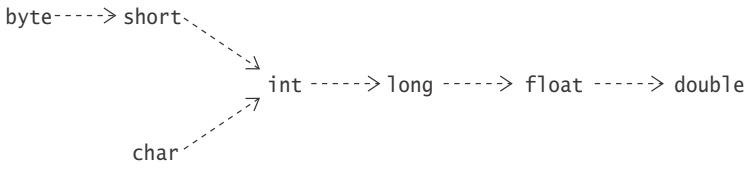
Widening and Narrowing Primitive Conversions

For the primitive data types, the value of a *narrower* data type can be converted to a value of a *wider* data type. This is called a *widening primitive conversion*. Widening conversions from one primitive type to the next wider primitive type are summarized in Figure 2.2. The conversions shown are transitive. For example, an `int` can be directly converted to a `double` without first having to convert it to a `long` and a `float`.

Note that the target type of a widening primitive conversion has a *wider range* of values than the source type—for example, the range of the `long` type subsumes the range of the `int` type. In widening conversions between *integral* types, the source value remains intact, with no loss of magnitude information. However, a widening conversion from an `int` or a `long` value to a `float` value, or from a `long` value to a `double` value, may result in a *loss of precision*. The floating-point value in the target type is then a correctly rounded approximation of the integer value. Note that precision relates to the number of significant bits in the value, and must not be confused with *magnitude*, which relates to how large the represented value can be.

Converting from a wider primitive type to a narrower primitive type is called a *narrowing primitive conversion*; it can result in a loss of magnitude information, and possibly in a loss of precision as well. Any conversion that is not a widening prim-

Figure 2.2 *Widening Primitive Conversions*



itive conversion according to Figure 2.2 is a narrowing primitive conversion. The target type of a narrowing primitive conversion has a *narrower range* of values than the source type—for example, the range of the `int` type does not include all the values in the range of the `long` type.

Note that all conversions between `char` and the two integer types `byte` and `short` are considered narrowing primitive conversions. The reason is that the conversions between the unsigned type `char` and the signed types `byte` and `short` can result in a loss of information. These narrowing conversions are done in two steps: first converting the source value to the `int` type, and then converting the `int` value to the target type.

Widening primitive conversions are usually done implicitly, whereas narrowing primitive conversions usually require a *cast* (p. 49). It is not illegal to use a cast for a widening conversion. However, the compiler will flag any conversion that requires a cast if none has been specified. Regardless of any loss of magnitude or precision, widening and narrowing primitive conversions *never* result in a runtime exception.

```
long year = 2020;    // (1) Implicit widening: long <----- int, assigned 2020L
int pi = (int) 3.14; // (2) Narrowing requires cast: int <----- double, assigned 3
```

Ample examples of widening and narrowing primitive conversions can be found in this chapter.

Widening and Narrowing Reference Conversions

The *subtype–supertype* relationship between reference types determines which conversions are permissible between them (§5.1, p. 195). Conversions *up* the *type hierarchy* are called *widening reference conversions* (also called *upcasting*). Such a conversion converts from a subtype to a supertype:

```
Object obj = "Upcast me";  // (1) Widening: Object <----- String
```

Conversions *down* the type hierarchy represent *narrowing reference conversions* (also called *downcasting*):

```
String str = (String) obj; // (2) Narrowing requires cast: String <----- Object
```

A subtype is a *narrower* type than its supertype in the sense that it is a specialization of its supertype. Contexts under which reference conversions can occur are discussed in §5.8, p. 266.

Widening reference conversions are usually done implicitly, whereas narrowing reference conversions usually require a cast, as illustrated in the second declaration statement above. The compiler will reject casts that are not legal or will issue an *unchecked warning* under certain circumstances if type-safety cannot be guaranteed.

Widening reference conversions do not require any runtime checks and never result in an exception during execution. This is not the case for narrowing reference conversions, which require a runtime check and can throw a `ClassCastException` if the conversion is not legal.

Boxing and Unboxing Conversions

Boxing and unboxing conversions allow interoperability between primitive values and their representation as objects of the wrapper types (§8.3, p. 439).

A *boxing conversion* converts the value of a primitive type to a corresponding value of its wrapper type, and an *unboxing conversion* converts the value of a wrapper type to a value of its corresponding primitive type. Both boxing and unboxing conversion are applied implicitly in the right context, but the wrapper classes also provide the `static` method `valueOf()` to explicitly box a primitive value in a wrapper object, and the method *primitiveType*`Value()` to explicitly unbox the value in a wrapper object as a value of *primitiveType*.

```
Integer iRef = 10;                   // (1) Implicit boxing: Integer <----- int
Double dRef = Double.valueOf(3.14);  // (2) Explicit boxing: Double <----- double

int i = iRef;                       // (3) Implicit unboxing: int <----- Integer
double d = dRef.doubleValue();      // (4) Explicit unboxing: double <----- Double
```

At (1) above, the `int` value `10` results in an object of type `Integer` implicitly being created; this object contains the `int` value `10`. We say that the `int` value `10` has been *boxed* in an object of the wrapper type `Integer`. This implicit boxing conversion is also called *autoboxing*. An explicit boxing by the `valueOf()` method of the wrapper classes is used at (2) to box a `double` value.

Unboxing conversion is illustrated by (3) and (4) above. Implicit unboxing is applied at (3) to unbox the value in the `Integer` object, and explicit unboxing is applied at (4) by calling the `doubleValue()` method of the `Double` class.

Note that both boxing and unboxing are done implicitly in the right context. Boxing allows primitive values to be used where an object of their wrapper type is expected, and unboxing allows the converse. Unboxing makes it possible to use a `Boolean` wrapper object as a `boolean` value in a boolean expression, and to use an integral wrapper object as an integral primitive value in an arithmetic expression. Unboxing a wrapper reference that has the `null` value results in a `NullPointerException`. Ample examples of boxing and unboxing can be found in this chapter and in §5.8, p. 266.

Other Conversions

Here we briefly mention some other conversions.

- *Identity conversions* allow conversions from a type to that same type. An identity conversion is always permitted.

```
int i = (int) 10;                  // int <---- int
String str = (String) "Hi";        // String <---- String
```

- *String conversions* allow a value of any other type to be converted to a `String` type in the context of the string concatenation operator + (p. 68).
- *Unchecked conversions* are permitted to facilitate operability between legacy and generic code (§11.2, p. 589).

2.4 Type Conversion Contexts

Selected conversion contexts and the conversions that are applicable in these contexts are summarized in Table 2.17. The conversions shown in each context occur *implicitly*, without the program having to take any special action. For other conversion contexts, see §2.3, p. 47.

Table 2.17 *Selected Conversion Contexts and Conversion Categories*

	Conversion contexts			
Conversion categories	**Assignment**	**Method invocation**	**Casting**	**Numeric promotion**
Widening/ narrowing *primitive* conversions	Widening Narrowing for *constant expressions* of non-`long` integral type, with optional boxing	Widening	Both	Widening
Widening/ narrowing *reference* conversions	Widening	Widening	Both, followed by optional unchecked conversion	Not applicable

Table 2.17 *Selected Conversion Contexts and Conversion Categories (Continued)*

Conversion categories	Conversion contexts			
	Assignment	**Method invocation**	**Casting**	**Numeric promotion**
Boxing/ unboxing conversions	Unboxing, followed by optional widening *primitive* conversion Boxing, followed by optional widening *reference* conversion	Unboxing, followed by optional widening *primitive* conversion Boxing, followed by optional widening *reference* conversion	Both	Unboxing, followed by optional widening *primitive* conversion

Assignment Context

Assignment conversions that can occur in an assignment context are shown in the second column of Table 2.17. An assignment conversion converts the type of an expression to the type of a target variable.

An expression (or its value) is *assignable* to the target variable, if the type of the expression can be converted to the type of the target variable by an assignment conversion. Equivalently, the type of the expression is *assignment compatible* with the type of the target variable.

For assignment conversion involving primitive data types, see §2.7, p. 55. Note the special case where a narrowing conversion occurs when assigning a non-`long` integer constant expression:

```
byte b = 10;   // Narrowing conversion: byte <--- int
```

For assignment conversions involving reference types, see §5.8, p. 266.

Method Invocation Context

Method invocation conversions that can occur in a method invocation context are shown in the third column of Table 2.17. Note that method invocation and assignment conversions differ in one respect: Method invocation conversions do not include the implicit narrowing conversion performed for non-`long` integral constant expressions.

```
// Assignment: (1) Implicit narrowing followed by (2) boxing.
Character space1 = 32;     // Character <-(2)-- char <-(1)-- int

// Invocation of method with signature: valueOf(char)
Character space2 = Character.valueOf(32);       // Compile-time error!
                                                // Call signature: valueOf(int)
Character space3 = Character.valueOf((char)32); // OK!
```

```
                                                    // Call signature: valueOf(char)
```

A method invocation conversion involves converting each argument value in a method or constructor call to the type of the corresponding formal parameter in the method or constructor declaration.

Method invocation conversions involving parameters of primitive data types are discussed in §3.10, p. 129, and those involving reference types are discussed in §5.8, p. 266.

Casting Context of the Unary Type Cast Operator (*type*)

Java, being a *strongly typed* language, checks for *type compatibility* (i.e., it checks whether a type can substitute for another type in a given context) at compile time. However, some checks are possible only at runtime (e.g., which type of object a reference actually denotes during execution). In cases where an operator would have incompatible operands (e.g., assigning a `double` to an `int`), Java demands that a *type cast* be used to *explicitly* indicate the type conversion. The type cast construct has the following syntax:

(*type*) *expression*

The *cast operator* (*type*) is applied to the value of the *expression*. At runtime, a cast results in a new value of *type,* which best represents the value of the *expression* in the old type. We use the term *casting* to mean applying the cast operator for *explicit* type conversion.

However, in the context of casting, *implicit* casting conversions can take place. These casting conversions are shown in the fourth column of Table 2.17. Casting conversions include more conversion categories than the assignment or the method invocation conversions. In the code that follows, the comments indicate the category of the conversion that takes place because of the cast operator on the right-hand side of each assignment—although casts are only necessary for the sake of the assignment at (1) and (2).

```
long l = (long) 10;          // Widening primitive conversion: long <--- int
int i = (int) l;             // (1) Narrowing primitive conversion: int <--- long
Object obj = (Object) "7Up"; // Widening ref conversion: Object <--- String
String str = (String) obj;   // (2) Narrowing ref conversion: String <--- Object
Integer iRef = (Integer) i;  // Boxing: Integer <--- int
i = (int) iRef;              // Unboxing: int <--- Integer
```

A casting conversion is applied to the value of the operand *expression* of a cast operator. Casting can be applied to primitive values as well as references. Casting between primitive data types and reference types is not permitted, except where boxing and unboxing is applicable. Boolean values cannot be cast to other data values, and vice versa. The reference literal `null` can be cast to any reference type.

Examples of casting between primitive data types are provided in this chapter. Casting reference values is discussed in §5.11, p. 274.

Numeric Promotion Context

Numeric operators allow only operands of certain types. Numeric promotion results in conversions being applied to the operands to convert them to permissible types. *Numeric promotion conversions* that can occur in a numeric promotion context are shown in the fifth column of Table 2.17. Permissible conversion categories are widening primitive conversions and unboxing conversions. A distinction is made between unary and binary numeric promotion.

Unary Numeric Promotion

Unary numeric promotion proceeds as follows:

- If the single operand is of type `Byte`, `Short`, `Character`, or `Integer`, it is unboxed. If the resulting value is narrower than `int`, it is promoted to a value of type `int` by a widening conversion.
- Otherwise, if the single operand is of type `Long`, `Float`, or `Double`, it is unboxed.
- Otherwise, if the single operand is of a type narrower than `int`, its value is promoted to a value of type `int` by a widening conversion.
- Otherwise, the operand remains unchanged.

In other words, *unary numeric promotion results in an operand value that is either* `int` *or wider.*

Unary numeric promotion is applied in the following expressions:

- Operand of the unary arithmetic operators `+` and `-` (p. 59)
- Array creation expression; for example, `new int[20]`, where the dimension expression (in this case, `20`) must evaluate to an `int` value (§3.9, p. 118)
- Indexing array elements; for example, `objArray['a']`, where the index expression (in this case, `'a'`) must evaluate to an `int` value (§3.9, p. 121)

Binary Numeric Promotion

Binary numeric promotion implicitly applies appropriate widening primitive conversions so that the widest numeric type of a pair of operands is always at least `int`. If `T` is the widest numeric type of two operands after any unboxing conversions have been performed, the operands are promoted as follows during binary numeric promotion:

> If `T` is wider than `int`, both operands are converted to `T`; otherwise, both operands are converted to `int`.

This means that *the resulting type of the operands is at least* `int`.

Binary numeric promotion is applied in the following expressions:

- Operands of the arithmetic operators `*`, `/`, `%`, `+`, and `-` (p. 59)
- Operands of the relational operators `<`, `<=`, `>`, and `>=` (p. 75)

- Operands of the numerical equality operators == and != (p. 76)
- Operands of the conditional operator ? :, under certain circumstances (p. 92)

2.5 Precedence and Associativity Rules for Operators

Precedence and associativity rules are necessary for deterministic evaluation of expressions. The operators are summarized in Table 2.18. The majority of them are discussed in subsequent sections in this chapter. See also the index entries for these operators.

The following remarks apply to Table 2.18:

- The operators are shown with decreasing precedence from the top of the table.
- Operators within the same row have the same precedence.
- Parentheses, (), can be used to override precedence and associativity.
- The *unary operators*, which require one operand, include the following: the postfix increment (++) and decrement (--) operators from the first row, all the prefix operators (+, -, ++, --, ~, !) in the second row, and the prefix operators (object creation operator new, cast operator (*type*)) in the third row.
- The conditional operator (? :) is *ternary*—that is, it requires three operands.
- All operators not identified previously as unary or ternary are *binary*—that is, they require two operands.
- All binary operators, except for the relational and assignment operators, associate from left to right. The relational operators are nonassociative.
- Except for unary postfix increment and decrement operators, all unary operators, all assignment operators, and the ternary conditional operator associate from right to left.

Depending on the context, brackets ([]), parentheses (()), the colon (:), and the dot operator (.) can also be interpreted as *separators* (p. 32). See the index entries for these separators for more details.

Table 2.18 *Operator Summary*

Array element access, member access, method invocation	[*expression*] . (*args*)
Unary postfix operators	*expression*++ *expression*--
Unary prefix operators	~ ! ++*expression* --*expression* +*expression* -*expression*
Unary prefix creation and cast	new (*type*)
Multiplicative	* / %
Additive	+ -

Table 2.18 *Operator Summary (Continued)*

Shift	`<< >> >>>`
Relational	`< <= > >= instanceof`
Equality	`== !=`
Bitwise/logical AND	`&`
Bitwise/logical XOR	`^`
Bitwise/logical OR	`|`
Conditional AND	`&&`
Conditional OR	`||`
Conditional	`?:`
Arrow operator	`->`
Assignment	`= += -= *= /= %= <<= >>= >>>= &= ^= |=`

Precedence rules are used to determine which operator should be applied first if there are two operators with a *different* precedence, and these operators follow each other in the expression. In such a case, the operator with the highest precedence is applied first.

The expression `2 + 3 * 4` is evaluated as `2 + (3 * 4)` (with the result `14`) since `*` has higher precedence than `+`.

Associativity rules are used to determine which operator should be applied first if there are two operators with the *same* precedence, and these operators follow each other in the expression.

Left associativity implies grouping from left to right: The expression `7 - 4 + 2` is interpreted as `((7 - 4) + 2)`, since the binary operators `+` and `-` both have the same precedence and left associativity.

Right associativity implies grouping from right to left: The expression `- - 4` is interpreted as `(- (- 4))` (with the result `4`), since the unary operator `-` has right associativity.

The precedence and associativity rules together determine the *evaluation order of the operators*.

2.6 Evaluation Order of Operands

To understand the result returned by an operator, it is important to understand the *evaluation order of its operands*. In general, the operands of operators are evaluated from left to right. The evaluation order also respects any parentheses, and the precedence and associativity rules of operators.

Examples illustrating how the operand evaluation order influences the result returned by an operator can be found in §2.7, p. 55, and §2.10, p. 70.

Left-Hand Operand Evaluation First

The left-hand operand of a binary operator is fully evaluated before the right-hand operand is evaluated.

The evaluation of the left-hand operand can have side effects that can influence the value of the right-hand operand. For example, in the code

```
int b = 10;
System.out.println((b=3) + b);
```

the value printed will be `6` and not `13`. The evaluation proceeds as follows:

```
(b=3) + b
  3   + b       b is assigned the value 3
  3   + 3
  6
```

If evaluation of the left-hand operand of a binary operator throws an exception (§7.1, p. 373), we cannot rely on the presumption that the right-hand operand has been evaluated.

Operand Evaluation before Operation Execution

Java guarantees that *all* operands of an operator are fully evaluated *before* the actual operation is performed. This rule does *not* apply to the short-circuit conditional operators `&&`, `||`, and `?:`.

This rule also applies to operators that throw an exception (the integer division operator `/` and the integer remainder operator `%`). The operation is performed only if the operands evaluate normally. Any side effects of the right-hand operand will have been effectuated before the operator throws an exception.

Example 2.1 illustrates the evaluation order of the operands and precedence rules for arithmetic expressions. We use the `eval()` method at (3) in Example 2.1 to demonstrate integer expression evaluation. The first argument to this method is the operand value that is returned by the method, and the second argument is a string to identify the evaluation order.

The argument to the `println()` method in the statement at (1) is an integer expression to evaluate 2 + 3 * 4. The evaluation of each operand in the expression at (1) results in a call of the `eval()` method declared at (3).

```
out.println(eval(j++, " + ") + eval(j++, " * ") * eval(j, "\n"));  // (1)
```

The output from Example 2.1 shows that the operands were evaluated first, from left to right, before operator execution, and that the expression was evaluated as `(2 + (3 * 4))`, respecting the precedence rules for arithmetic expression evaluation.

Note how the value of variable j changes successively from left to right as the first two operands are evaluated.

Example 2.1 *Evaluation Order of Operands and Arguments*

```
import static java.lang.System.out;

public class EvalOrder{
  public static void main(String[] args){

    int j = 2;
    out.println("Evaluation order of operands:");
    out.println(eval(j++, " + ") + eval(j++, " * ") * eval(j, "\n"));    // (1)

    int i = 1;
    out.println("Evaluation order of arguments:");
    add3(eval(i++, ", "), eval(i++, ", "), eval(i, "\n")); // (2) Three arguments.
  }

  public static int eval(int operand, String str) {        // (3)
    out.print(operand + str);       // Print int operand and String str.
    return operand;                 // Return int operand.
  }

  public static void add3(int operand1, int operand2, int operand3) {    // (4)
    out.print(operand1 + operand2 + operand3);
  }
}
```

Output from the program:

```
Evaluation order of operands:
2 + 3 * 4
14
Evaluation order of arguments:
1, 2, 3
6
```

Left-to-Right Evaluation of Argument Lists

In a method or constructor invocation, each argument expression in the argument list is fully evaluated before any argument expression to its right.

If evaluation of an argument expression does not complete normally, we cannot presume that any argument expression to its right has been evaluated.

We can use the `add3()` method at (4) in Example 2.1, which takes three arguments, to demonstrate the order in which the arguments in a method call are evaluated. The method call at (2)

```
add3(eval(i++, ", "), eval(i++, ", "), eval(i, "\n"));  // (2) Three arguments.
```

results in the following output, clearly indicating that the arguments were evaluated from left to right, before being passed to the method:

```
1, 2, 3
6
```

Note how the value of variable `i` changes successively from left to right as the first two arguments are evaluated.

2.7 The Simple Assignment Operator =

The assignment statement has the following syntax:

variable = *expression*

which can be read as "the target, *variable*, gets the value of the source, *expression*." The previous value of the target variable is overwritten by the assignment operator =.

The target *variable* and the source *expression* must be assignment compatible. The target variable must also have been declared. Since variables can store either primitive values or reference values, *expression* evaluates to either a primitive value or a reference value.

Assigning Primitive Values

The following examples illustrate assignment of primitive values:

```
int j, k;
j = 0b10;         // j gets the value 2.
j = 5;            // j gets the value 5. Previous value is overwritten.
k = j;            // k gets the value 5.
```

The assignment operator has the lowest precedence, so the expression on the right-hand side is evaluated before the assignment is done.

```
int i;
i = 5;            // i gets the value 5.
i = i + 1;        // i gets the value 6. + has higher precedence than =.
i = 20 - i * 2;   // i gets the value 8: (20 - (i * 2))
```

Assigning References

Copying reference values by assignment creates *aliases*. Below, the variable `pizza1` is a reference to a pizza that is hot and spicy, and `pizza2` is a reference to a pizza that is sweet and sour.

```
Pizza pizza1 = new Pizza("Hot&Spicy");
Pizza pizza2 = new Pizza("Sweet&Sour");

pizza2 = pizza1;
```

Assigning `pizza1` to `pizza2` means that `pizza2` now refers to the same pizza as `pizza1`, the hot and spicy one. After the assignment, these variables are aliases and either one can be used to manipulate the hot and spicy `Pizza` object.

Assigning a reference value does *not* create a copy of the source object denoted by the reference variable on the right-hand side. It merely assigns the reference value of the variable on the right-hand side to the variable on the left-hand side so that they denote the same object. Reference assignment also does not copy the *state* of the source object to any object denoted by the reference variable on the left-hand side.

A more detailed discussion of reference assignment can be found in §5.8, p. 266.

Multiple Assignments

The assignment statement is an *expression statement*, which means that application of the binary assignment operator returns the value of the expression on the *right-hand* side.

```
int j, k;
j = 10;     // (1) j gets the value 10, which is returned
k = j;      // (2) k gets the value of j, which is 10, and this value is returned
```

The value returned by an assignment statement is usually discarded, as in the two assignment statements above. We can verify the value returned as follows:

```
System.out.println(j = 10);   // j gets the value 10, which is printed.
System.out.println(k = j);    // k gets the value of j, i.e. 10, which is printed
```

The two assignments (1) and (2) above can be written as multiple assignments, illustrating the right associativity of the assignment operator:

```
k = j = 10;     // (k = (j = 10))
```

Multiple assignments are equally valid with references:

```
Pizza pizzaOne, pizzaTwo;
pizzaOne = pizzaTwo = new Pizza("Supreme"); // Aliases
```

The following example shows the effect of operand evaluation order:

```
int[] a = {10, 20, 30, 40, 50}; // An array of int (§3.9, p. 120)
int index = 4;
a[index] = index = 2;           // (1)
```

What is the value of `index`, and which array element `a[index]` is assigned a value in the multiple assignment statement at (1)? The evaluation proceeds as follows:

```
a[index] = index = 2;
a[4]     = index = 2;
a[4]     = (index = 2);        // index gets the value 2. = is right associative.
a[4]     =       2;            // The value of a[4] is changed from 50 to 2.
```

The following declaration statement will not compile, as the variable `v2` has not been declared:

```
int v1 = v2 = 2016;            // Only v1 is declared. Compile-time error!
```

Type Conversions in an Assignment Context

If the target and the source have the same type in an assignment, then obviously the source and the target are assignment compatible and the source value need not be converted. Otherwise, if a widening primitive conversion is permissible, then the widening conversion is applied implicitly; that is, the source type is converted to the target type in an assignment context.

```
// Widening Primitive Conversions
int    smallOne = 1234;              // No widening necessary.
long   bigOne   = 2020;              // Widening: int to long.
double largeOne = bigOne;            // Widening: long to double.
double hugeOne  = (double) bigOne;   // Cast redundant but allowed.
```

A widening primitive conversion can result in loss of *precision*. In the next example, the precision of the least significant bits of the `long` value may be lost when it is converted to a `float` value:

```
long bigInteger = 98765432112345678L;
float fpNum = bigInteger;  // Widening but loss of precision: 9.8765436E16
```

Additionally, *implicit narrowing primitive conversions* on assignment can occur in cases where *all* of the following conditions are fulfilled:

- The source is a *constant expression* of type `byte`, `short`, `char`, or `int`.
- The target type is of type `byte`, `short`, or `char`.
- The value of the source is determined to be in the range of the target type at compile time.

A *constant expression* is an expression that denotes either a primitive or a `String` literal; it is composed of operands that can be only *literals* or *constant variables*, and operators that can be evaluated only at compile time (e.g., arithmetic and numerical comparison operators, but not increment/decrement operators and method calls). A *constant variable* is a `final` variable of either a primitive type or the `String` type that is initialized with a constant expression.

```
int result = 100;              // Not a constant variable. Not declared final.
final char finalGrade = 'A';   // Constant variable. 'A'
System.out.printf("%d%n%s%n%d%n%.2f%n%b%n%d%n%d%n",
    2022,                      // Constant expression. 2022
    "Trust " + "me!",          // Constant expression. "Trust me"
    2 + 3 * 4,                 // Constant expression. 14
    Math.PI * Math.PI * 10.0,  // Constant expression. 98.70
    finalGrade == 'A',         // Constant expression. true
    Math.min(2020, 2021),      // Not constant expression. Method call.
    ++result                   // Not constant expression. Increment operator.
);
```

Here are some examples that illustrate how the conditions mentioned previously affect narrowing primitive conversions:

```
// Conditions fulfilled for implicit narrowing primitive conversions.
short s1 = 10;       // int value in range.
short s2 = 'a';      // char value in range.
char c1 = 32;        // int value in range.
char c2 = (byte)35;  // byte value in range. (int value in range, without cast.)
byte b1 = 40;        // int value in range.
byte b2 = (short)40; // short value in range. (int value in range, without cast.)
final int i1 = 20;   // Constant variable
byte b3 = i1;        // final value of i1 in range.
```

All other narrowing primitive conversions will produce a compile-time error on assignment and will explicitly require a cast. Here are some examples:

```
// Conditions not fulfilled for implicit narrowing primitive conversions.
// A cast is required.
int i2 = -20;            // i2 is not a constant variable. i2 is not final.
final int i3 = i2;       // i3 is not a constant variable, since i2 is not.
final int i4 = 200;      // i4 is a constant variable.
final int i5;            // i5 is not a constant variable.
short s3 = (short) i2;   // Not constant expression.
char  c3 = (char)  i3;   // Final value of i3 not determinable at compile time.
char  c4 = (char)  i2;   // Not constant expression.
byte  b4 = (byte)  128;  // int value not in range.
byte  b5 = (byte)  i4;   // Value of constant variable i4 is not in range.
i5 = 100;                // Initialized at runtime.
short s4 = (short) i5;   // Final value of i5 not determinable at compile time.
```

Floating-point values are truncated when cast to integral values.

```
// The value is truncated to fit the size of the target type.
float huge   = (float) 1.7976931348623157d;  // double to float.
long  giant  = (long)  4415961481999.03D;    // (1) double to long.
int   big    = (int)   giant;                // (2) long to int.
short small  = (short) big;                  // (3) int to short.
byte  tiny   = (byte)  small;                // (4) short to byte.
char  symbol = (char)  112.5F;               // (5) float to char.
```

Table 2.19 shows how the values are truncated for assignments from (1) to (5).

Table 2.19 *Examples of Truncated Values*

Binary	Decimal	
00000000000000000000010000000100001010111101000011000011000011111	4415961481999	(1)
00101011110100001100001100001111	735101711	(2)
1100001100001111	-15601	(3)
00001111	15	(4)
0000000001110000	'p'	(5)

The discussion of numeric assignment conversions also applies to numeric parameter values at method invocation (§3.10, p. 129), except for the narrowing conversions, which always require a cast.

The following examples illustrate boxing and unboxing in an assignment context:

```
Boolean   boolRef = true;  // Boxing.
Byte      bRef =  2;       // Constant in range: narrowing, then boxing.
// Byte  bRef2 =  257;     // Constant not in range. Compile-time error!

short s = 10;              // Narrowing from int to short.
// Integer   iRef1 = s;    // short not assignable to Integer.
Integer iRef3 = (int) s;   // Explicit widening with cast to int and boxing

boolean bv1 = boolRef;     // Unboxing.
byte  b1 = bRef;           // Unboxing.
int   iVal = bRef;         // Unboxing and widening.

Integer iRefVal = null;           // Always allowed.
// int j = iRefVal;               // NullPointerException at runtime.
if (iRef3 != null) iVal = iRef3;  // Avoids exception at runtime.
```

2.8 Arithmetic Operators: *, /, %, +, -

Arithmetic operators are used to construct mathematical expressions as in algebra. Their operands are of a numeric type (which includes the `char` type).

Floating-point operations are now consistently *strict*; that is, they are executed in accordance with the IEEE-754 32-bit (`float`) and 64-bit (`double`) standard formats. This means that floating-point arithmetic operations give the same results on any JVM implementation. The keyword `strictfp` used to enforce strict behavior for floating-point arithmetic is now obsolete and should not be used in new code.

Arithmetic Operator Precedence and Associativity

In Table 2.20, the precedence of the operators appears in decreasing order, starting from the top row, which has the highest precedence. Unary subtraction has higher precedence than multiplication. The operators in the same row have the same precedence. Binary multiplication, division, and remainder operators have the same precedence. The unary operators have right associativity, and the binary operators have left associativity.

Table 2.20 *Arithmetic Operators*

Unary	+ *Plus*		- *Minus*		
Binary	* *Multiplication*		/ *Division*	% *Remainder*	
	+ *Addition*		- *Subtraction*		

Evaluation Order in Arithmetic Expressions

Java guarantees that the operands are fully evaluated from left to right before an arithmetic binary operator is applied. If evaluation of an operand results in an error, the subsequent operands will not be evaluated.

In the expression `a + b * c`, the operand `a` will always be fully evaluated before the operand `b`, which will always be fully evaluated before the operand `c`. However, the multiplication operator `*` will be applied before the addition operator `+`, respecting the precedence rules. Note that `a`, `b`, and `c` are arbitrary arithmetic expressions that have been determined to be the operands of the operators.

Example 2.1, p. 54, illustrates the evaluation order and precedence rules for arithmetic expressions.

Range of Numeric Values

As we have seen, all numeric types have a range of valid values (p. 41). This range is given by the constants named `MAX_VALUE` and `MIN_VALUE`, which are defined in each numeric wrapper type.

The arithmetic operators are overloaded, meaning that the operation of an operator varies depending on the type of its operands. Floating-point arithmetic is performed if any operand of an operator is of floating-point type; otherwise, integer arithmetic is performed.

Values that are out of range or are the results of invalid expressions are handled differently depending on whether integer or floating-point arithmetic is performed.

Integer Arithmetic

Integer arithmetic always returns a value that is in range, except in the case of integer division by zero and remainder by zero, which cause an `ArithmeticException` (see the later discussion of the division operator `/` and the remainder operator `%`). A valid value does not necessarily mean that the result is correct, as demonstrated by the following examples:

```
int tooBig   = Integer.MAX_VALUE + 1;    // -2147483648 which is Integer.MIN_VALUE.
int tooSmall = Integer.MIN_VALUE - 1;    //  2147483647 which is Integer.MAX_VALUE.
```

These results should be values that are out of range. However, integer arithmetic *wraps around* the result if it is out of range; that is, the result is reduced modulo in the range of the result type. To avoid wrapping around out-of-range values, programs should use either explicit checks or a wider type. If the type `long` were used in the earlier examples, the results would be correct in the `long` range:

```
long notTooBig   = Integer.MAX_VALUE + 1L;   //  2147483648L in range.
long notTooSmall = Integer.MIN_VALUE - 1L;   // -2147483649L in range.
```

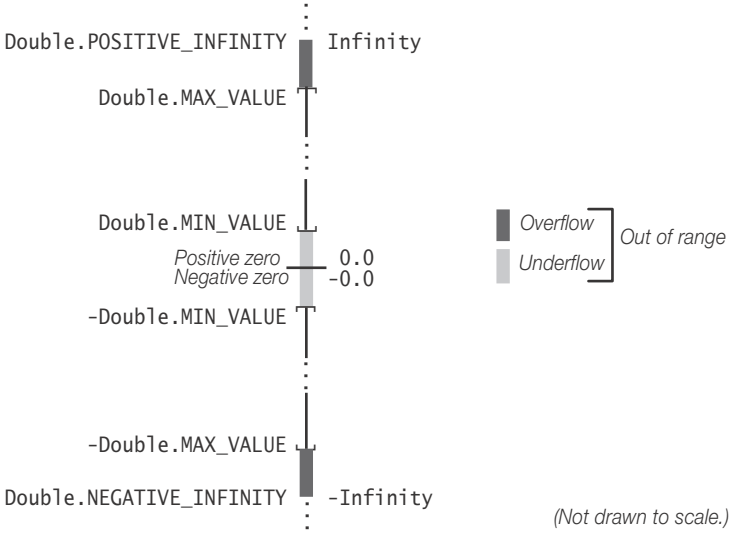
Floating-Point Arithmetic

Certain floating-point operations result in values that are out of range. Typically, adding or multiplying two very large floating-point numbers can result in an out-of-range value that is represented by *infinity* (Figure 2.3). Attempting floating-point division by zero also returns infinity. The following examples show how this value is printed as signed infinity:

```
System.out.println( 4.0 / 0.0);         // Prints:  Infinity
System.out.println(-4.0 / 0.0);         // Prints: -Infinity
```

Both positive and negative infinity represent *overflow* to infinity; that is, the value is too large to be represented as a `double` or `float` (Figure 2.3). Signed infinity is represented by the named constants `POSITIVE_INFINITY` and `NEGATIVE_INFINITY` in the wrapper classes `java.lang.Float` and `java.lang.Double`. A value can be compared with these constants to detect overflow.

Figure 2.3 *Overflow and Underflow in Floating-Point Arithmetic*



Floating-point arithmetic can also result in *underflow* to zero, when the value is too small to be represented as a `double` or `float` (Figure 2.3). Underflow occurs in the following situations:

- The result is between `Double.MIN_VALUE` (or `Float.MIN_VALUE`) and zero, as with the result of `(5.1E-324 - 4.9E-324)`. Underflow then returns positive zero `0.0` (or `0.0F`).
- The result is between `-Double.MIN_VALUE` (or `-Float.MIN_VALUE`) and zero, as with the result of `(-Double.MIN_VALUE * 1E-1)`. Underflow then returns negative zero `-0.0` (or `-0.0F`).

Negative zero compares equal to positive zero; in other words, `(-0.0 == 0.0)` is `true`.

Certain operations have no mathematical result, and are represented by *NaN* (*Not-a-Number*). For example, calculating the square root of –1 results in a NaN. Another example is (floating-point) dividing zero by zero:

```
System.out.println(0.0 / 0.0);        // Prints: NaN
```

NaN is represented by the constant named `NaN` in the wrapper classes `java.lang.Float` and `java.lang.Double`. Any operation involving NaN produces NaN. Any comparison (except inequality, `!=`) involving NaN and any other value (including NaN) returns `false`. An inequality comparison of NaN with another value (including NaN) always returns `true`. However, the recommended way of checking a value for NaN is to use the static method `isNaN()` defined in both wrapper classes, `java.lang.Float` and `java.lang.Double`.

Unary Arithmetic Operators: -, +

The unary operators have the highest precedence of all the arithmetic operators. The unary operator - negates the numeric value of its operand. The following example illustrates the right associativity of the unary operators:

```
int value = - -10;              // (-(-10)) is 10
```

Notice the blank space needed to separate the unary operators; without the blank space, these would be interpreted as the decrement operator -- (p. 70), which would result in a compile-time error because a literal cannot be decremented. The unary operator + has no effect on the evaluation of the operand value.

Multiplicative Binary Operators: *, /, %

*Multiplication Operator: **

The multiplication operator * multiplies two numbers, as one would expect.

```
int    sameSigns     = -4    * -8;   // result:  32
double oppositeSigns =  4    * -8.0; // Widening of int 4 to double. result: -32.0
int    zero          =  0    * -0;   // result:   0
```

Division Operator: /

The division operator / is overloaded. If its operands are integral, the operation results in *integer division*.

```
int    i1 = 4  / 5;   // result: 0
int    i2 = 8  / 8;   // result: 1
double d1 = 12 / 8;   // result: 1.0; integer division, then widening conversion
```

Integer division always returns the quotient as an integer value; that is, the result is truncated toward zero. Note that the division performed is integer division if the operands have integral values, even if the result will be stored in a floating-point type. The integer value is subjected to a widening conversion in the assignment context.

An `ArithmeticException` is thrown when integer division with zero is attempted, meaning that integer division by zero is an illegal operation.

If any of the operands is a floating-point type, the operation performs *floating-point division*, where relevant operand values undergo binary numeric promotion:

```
double d2 = 4.0 / 8;      // result: 0.5
double d3 = 8 / 8.0;      // result: 1.0
float d4  = 12.0F / 8;    // result: 1.5F

double result1 = 12.0 / 4.0 * 3.0;       // ((12.0 / 4.0) * 3.0) which is 9.0
double result2 = 12.0 * 3.0 / 4.0;       // ((12.0 * 3.0) / 4.0) which is 9.0
```

Remainder Operator: `%`

In mathematics, when we divide a number (the *dividend*) by another number (the *divisor*), the result can be expressed in terms of a *quotient* and a *remainder*. For example, when 7 is divided by 5, the quotient is 1 and the remainder is 2. The remainder operator `%` returns the remainder of the division performed on the operands.

```
int quotient  = 7 / 5;   // Integer division operation: 1
int remainder = 7 % 5;   // Integer remainder operation: 2
```

For *integer remainder operation*, where only integer operands are involved, evaluation of the expression `(x % y)` always satisfies the following relation:

$x == (x / y) * y + (x \% y)$

In other words, the right-hand side yields a value that is always equal to the value of the dividend. The following examples show how we can calculate the remainder so that this relation is satisfied:

Calculating (7 % 5):

```
7 == (7 / 5) * 5 + (7 % 5)
  == (  1  ) * 5 + (7 % 5)
  ==           5 + (7 % 5)
2 ==               (7 % 5)          (7 % 5) is equal to 2
```

Calculating (7 % –5):

```
7 == (7 / -5) * -5 + (7 % -5)
  == (  -1  ) * -5 + (7 % -5)
  ==            5 + (7 % -5)
2 ==                (7 % -5)       (7 % –5) is equal to 2
```

Calculating (–7 % 5):

```
-7 == (-7 / 5) * 5 + (-7 % 5)
   == (  -1  ) * 5 + (-7 % 5)
   ==           -5 + (-7 % 5)
-2 ==               (-7 % 5)       (–7 % 5) is equal to –2
```

Calculating (–7 % –5):

```
-7 == (-7 / -5) * -5 + (-7 % -5)
```

```
   == (   1   ) * -5 + (-7 % -5)
   ==             -5 + (-7 % -5)
-2 ==                  (-7 % -5)      (–7 % –5) is equal to –2
```

The remainder can be negative only if the dividend is negative, and the sign of the divisor is irrelevant. A shortcut to evaluating the remainder involving negative operands is the following: Ignore the signs of the operands, calculate the remainder, and negate the remainder if the dividend is negative.

```
int  r0 =  7  %  7;     //  0
int  r1 =  7  %  5;     //  2
long r2 =  7L % -5L;    //  2L
int  r3 = -7  %  5;     // -2
long r4 = -7L % -5L;    // -2L
boolean relation = -7L == (-7L / -5L) * -5L + r4;  // true
```

An `ArithmeticException` is thrown if the divisor evaluates to zero.

Note that the remainder operator accepts not only integral operands, but also floating-point operands. The *floating-point remainder* `r` is defined by the relation

$r == a - (b * q)$

where `a` and `b` are the dividend and the divisor, respectively, and `q` is the *integer* quotient of `(a/b)`. The following examples illustrate a floating-point remainder operation:

```
double  dr0 =  7.0  %  7.0;    //  0.0
float   fr1 =  7.0F %  5.0F;   //  2.0F
double  dr1 =  7.0  % -5.0;    //  2.0
float   fr2 = -7.0F %  5.0F;   // -2.0F
double  dr2 = -7.0  % -5.0;    // -2.0
boolean fpRelation = dr2  == (-7.0) - (-5.0) * (long)(-7.0 / -5.0);  // true
float   fr3 = -7.0F %  0.0F;   // NaN
```

Additive Binary Operators: +, -

The addition operator + and the subtraction operator - behave as their names imply: They add and subtract values, respectively. The binary operator + also acts as *string concatenation* if any of its operands is a string (p. 68).

Additive operators have lower precedence than all the other arithmetic operators. Table 2.21 includes examples that show how precedence and associativity are used in arithmetic expression evaluation.

Table 2.21 *Examples of Arithmetic Expression Evaluation*

Arithmetic expression	Evaluation	Result when printed
`3 + 2 - 1`	`((3 + 2) - 1)`	`4`
`2 + 6 * 7`	`(2 + (6 * 7))`	`44`

Table 2.21 *Examples of Arithmetic Expression Evaluation (Continued)*

Arithmetic expression	Evaluation	Result when printed
-5 + 7 - -6	(((-5) + 7) - (-6))	8
2 + 4 / 5	(2 + (4 / 5))	2
13 % 5	(13 % 5)	3
11.5 % 2.5	(11.5 % 2.5)	1.5
10 / 0		ArithmeticException
2 + 4.0 / 5	(2.0 + (4.0 / 5.0))	2.8
4.0 / 0.0	(4.0 / 0.0)	Infinity
-4.0 / 0.0	((-4.0) / 0.0)	-Infinity
0.0 / 0.0	(0.0 / 0.0)	NaN

Numeric Promotions in Arithmetic Expressions

Unary numeric promotion is applied to the single operand of the unary arithmetic operators - and +. When a unary arithmetic operator is applied to an operand whose type is narrower than `int`, the operand is promoted to a value of type `int`, with the operation resulting in an `int` value. If the conditions for implicit narrowing conversion are not fulfilled (p. 57), assigning the `int` result to a variable of a narrower type will require a cast. This is demonstrated by the following example, where the `byte` operand `b` is promoted to an `int` in the expression `(-b)`:

```
byte b = 3;          // int literal in range. Narrowing conversion.
b = (byte) -b;       // Cast required on assignment.
```

Binary numeric promotion is applied to operands of binary arithmetic operators. Its application leads to type promotion for the operands, as explained in §2.4, p. 50. The result is of the promoted type, which is always type `int` or wider. For the expression at (1) in Example 2.2, numeric promotions proceed as shown in Figure 2.4. Note the integer division performed in evaluating the subexpression `(c / s)`.

Example 2.2 *Numeric Promotion in Arithmetic Expressions*

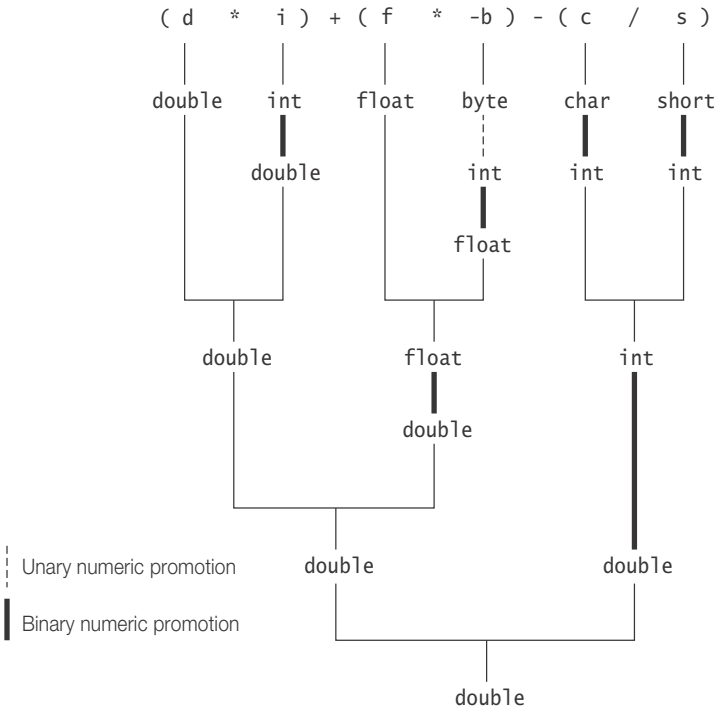
```
public class NumPromotion {
  public static void main(String[] args) {
    byte   b = 32;
    char   c = 'z';                              // Unicode value 122 (\u007a)
    short  s = 256;
    int    i = 10000;
    float  f = 3.5F;
    double d = 0.5;
    double v = (d * i) + (f * -b) - (c / s);     // (1) 4888.0D
    System.out.println("Value of v: " + v);
  }
```

```
}
```

Output from the program:

```
Value of v: 4888.0
```

Figure 2.4 *Numeric Promotion in Arithmetic Expressions*



In addition to the binary numeric promotions in arithmetic expression evaluation, the resulting value can undergo an implicit widening conversion if assigned to a variable. In the first two declaration statements that follow, only assignment conversions take place. Numeric promotions take place in the evaluation of the right-hand expression in the other declaration statements.

```
Byte   b = 10;       // Constant in range: narrowing and boxing on assignment.
Short  s = 20;       // Constant in range: narrowing and boxing on assignment.
char   c = 'z';      // 122 (\u007a)
int    i = s * b;    // Values in s and b promoted to int: unboxing, widening.
long   n = 20L + s;  // Value in s promoted to long: unboxing, widening.
float  r = s + c;    // Value in s is unboxed. This short value and the char
                     // value in c are promoted to int, followed by implicit
                     // widening conversion of int to float on assignment.
double d = r + i;    // Value in i promoted to float, followed by implicit
                     // widening conversion of float to double on assignment.
```

Binary numeric promotion for operands of binary operators implies that each operand of a binary operator is promoted to type `int` or a broader numeric type, if necessary. As with unary operators, care must be exercised in assigning the value resulting from applying a binary operator to operands of these types.

```
short h = 40;          // OK: int converted to short. Implicit narrowing.
h = h + 2;             // Error: cannot assign an int to short.
```

The value of the expression `h + 2` is of type `int`. Although the result of the expression is in the range of `short`, this cannot be determined at compile time. The assignment requires a cast.

```
h = (short) (h + 2);   // OK
```

Notice that applying the cast operator `(short)` to the individual operands does not work:

```
h = (short) h + (short) 2;     // The resulting value should be cast.
```

Neither does the following approach, which results in a compile-time error:

```
h = (short) h + 2;             // The resulting value should be cast.
```

In this case, binary numeric promotion leads to an `int` value as the result of evaluating the expression on the right-hand side, and therefore, requires an additional cast to narrow it to a `short` value.

Arithmetic Compound Assignment Operators: *=, /=, %=, +=, -=

A *compound assignment operator* has the following syntax:

variable op= expression

and the following semantics:

variable = (*type*) ((*variable*) *op* (*expression*))

The type of the *variable* is *type* and the *variable* is evaluated only once. Note the cast and the parentheses implied in the semantics. Here *op*= can be any of the compound assignment operators specified in Table 2.18. The compound assignment operators have the lowest precedence of all the operators in Java, allowing the expression on the right-hand side to be evaluated before the assignment. Table 2.22 defines the arithmetic compound assignment operators.

Table 2.22 *Arithmetic Compound Assignment Operators*

Expression	Given `T` as the numeric type of `x`, the expression is evaluated as:
`x *= a`	`x = (T) ((x) * (a))`
`x /= a`	`x = (T) ((x) / (a))`
`x %= a`	`x = (T) ((x) % (a))`
`x += a`	`x = (T) ((x) + (a))`
`x -= a`	`x = (T) ((x) - (a))`

The implied cast operator, (T), in the compound assignments becomes necessary when the result must be narrowed to the target type. This is illustrated by the following examples:

```
int i = 2;
i *= i + 4;             // (1) Evaluated as i = (int) ((i) * (i + 4)).

Integer iRef = 2;
iRef *= iRef + 4;       // (2) Evaluated as iRef = (Integer) ((iRef) * (iRef + 4)).

byte b = 2;
b += 10;                // (3) Evaluated as b = (byte) (b + 10).
b = b + 10;             // (4) Will not compile. Cast is required.
```

At (1) the source int value is assigned to the target int variable, and the cast operator (int) in this case is an *identity conversion* (i.e., conversion from a type to the same type). Such casts are permitted. The assignment at (2) entails unboxing to evaluate the expression on the right-hand side, followed by boxing to assign the int value. However, at (3), as the source value is an int value because the byte value in b is promoted to int to carry out the addition, assigning it to a target byte variable requires an implicit narrowing conversion. The situation at (4) with simple assignment will not compile because implicit narrowing conversion is not applicable.

The *variable* is evaluated only once in the expression, not twice, as one might infer from the definition of the compound assignment operator. In the following assignment, a[i] is evaluated just once:

```
int[] a = new int[] { 2020, 2030, 2040 };
int i = 2;
a[i] += 1;      // Evaluates as a[2] = a[2] + 1, and a[2] gets the value 2041.
```

Implicit narrowing conversions are also applied to increment and decrement operators (p. 70).

Boolean logical compound assignment operators are covered in §2.14, p. 79.

2.9 The Binary String Concatenation Operator +

The binary operator + is overloaded in the sense that the operation performed is determined by the type of the operands. When one of the operands is a String object, a string concatenation is performed rather than numeric addition. String concatenation results in a newly created String object in which the characters in the text representation of the left-hand operand precede the characters in the text representation of the right-hand operand. It might be necessary to perform a *string conversion* on the non-String operand before the string concatenation can be performed. The String class is discussed in §8.4, p. 449.

A *string conversion* is performed on the non-String operand as follows:

- For an operand of a primitive data type, its value is converted to a text representation.

- For all reference value operands, a text representation is constructed by calling the no-argument `toString()` method on the referred object. Most classes override this method from the `Object` class so as to provide a more meaningful text representation of their objects. Discussion of the `toString()` method can be found in §8.2, p. 435.
- Values like `true`, `false`, and `null` have text representations that correspond to their names. A reference variable with the value `null` also has the text representation `"null"` in this context.

The operator + is left associative and has the same precedence level as the additive operators, whether it is performed as a string concatenation or as a numeric addition.

```
String strVal = "" + 2020;                    // (1) "2020"
String theName = " Uranium";
theName = " Pure" + theName;                  // (2) " Pure Uranium"
String trademark1 = 100 + "%" + theName;      // (3) "100% Pure Uranium"
```

Since the + operator is left-associative, the evaluation at (3) proceeds as follows: The `int` value `100` is concatenated with the string literal `"%"`, followed by concatenation with the contents of the `String` object referred to by the `theName` reference.

Note that using the character literal `'%'` instead of the string literal `"%"` in line (2) does not give the same result:

```
String trademark2 = 100 + '%' + theName;      // (4) "137 Pure Uranium"
```

Integer addition is performed by the first + operator: `100 + '%'`; that is, `(100 + 37)`.

Caution should be exercised because the + operator might not be applied as intended, as shown by the following example:

```
System.out.println("We can put two and two together and get " + 2 + 2);   // (5)
```

This statement prints `"We can put two and two together and get 22"`. String concatenation proceeds from left to right: The `String` literal is concatenated with the first `int` literal `2`, followed by concatenation with the second `int` literal `2`. Both occurrences of the + operator are treated as string concatenation. To convey the intended meaning of the sentence, parentheses are necessary:

```
System.out.println("We can put two and two together and get " + (2 + 2)); // (6)
```

This statement prints `"We can put two and two together and get 4"`, since the parentheses enforce integer addition in the expression `(2 + 2)` before string concatenation is performed with the contents of the `String` operand.

The following statement will print the correct result, even without the parentheses, because the * operator has higher precedence than the + operator:

```
System.out.println("2 * 2 = " + 2 * 2);       // (7) 2 * 2 = 4
```

Creation of temporary `String` objects might be necessary to store the results of performing successive string concatenations in a `String`-valued expression. For a

`String`-valued *constant expression* ((1), (5), (6), and (7) in the preceding examples), the compiler computes such an expression at compile time, and the result is treated as a string literal in the program. The compiler uses a *string builder* to avoid the overhead of temporary `String` objects when applying the string concatenation operator (+) in `String`-valued non-constant expressions ((2), (3), and (4) in the preceding examples), as explained in §8.5, p. 480.

2.10 Variable Increment and Decrement Operators: ++, --

Variable increment (++) and decrement (--) operators come in two flavors: *prefix* and *postfix*. These unary operators have the side effect of changing the value of the arithmetic operand, which must evaluate to a variable. Depending on the operator used, the variable is either incremented by 1 or decremented by 1.

These operators cannot be applied to a variable that is declared `final` and that has been initialized, as the side effect would change the value in such a variable.

These operators are useful for updating variables in loops where only the side effect of the operator is of interest.

The Increment Operator ++

The prefix increment operator has the following semantics: `++i` adds `1` to the value in `i`, and stores the new value in `i`. It returns the *new* value as the value of the expression. It is equivalent to the following statements:

```
i += 1;
result = i;
return result;
```

The postfix increment operator has the following semantics: `j++` adds `1` to the value in `j`, and stores the new value in `j`. It returns the *original* value that was in `j` as the value of the expression. It is equivalent to the following statements:

```
result = j;
j += 1;
return result;
```

The Decrement Operator --

The prefix decrement operator has the following semantics: `--i` subtracts `1` from the value of `i`, and stores the new value in `i`. It returns the *new* value as the value of the expression. It is equivalent to the following statements:

```
i -= 1;
result = i;
return result;
```

The postfix decrement operator has the following semantics: `j--` subtracts `1` from the value of `j`, and stores the new value in `j`. It returns the *original* value that was in `j` as the value of the expression. It is equivalent to the following statements:

```
result = j;
j -= 1;
return result;
```

This behavior of decrement and increment operators applies to any variable whose type is a numeric primitive type or its corresponding numeric wrapper type. Necessary numeric promotions are performed on the value `1` and the value of the variable. Before the new value is assigned to the variable, it is subjected to any narrowing primitive conversion and/or boxing that might be necessary.

Here are some examples that illustrate the behavior of increment and decrement operators:

```
// (1) Prefix order: increment/decrement operand before use.
int i = 10;
int k = ++i + --i;  // ((++i) + (--i)). k gets the value 21 and i becomes 10.
--i;                // Only side effect utilized. i is 9. (expression statement)

Integer iRef = 11;  // Boxing on assignment
--iRef;             // Only side effect utilized. iRef refers to an Integer
                    // object with the value 10. (expression statement)
k = ++iRef + --iRef;// ((++iRef) + (--iRef)). k gets the value 21 and
                    // iRef refers to an Integer object with the value 10.

// (2) Postfix order: increment/decrement operand after use.
long j = 10;
long n = j++ + j--; // ((j++) + (j--)). n gets the value 21L and j becomes 10L.
j++;                // Only side effect utilized. j is 11L. (expression statement)
```

An increment or decrement operator, together with its operand, can be used as an *expression statement* (§3.3, p. 101).

Execution of the assignment in the second declaration statement under (1) proceeds as follows:

```
k = ((++i) + (--i))          Operands are evaluated from left to right.
k = ( 11   + (--i))          Side effect: i += 1, i gets the value 11.
k = ( 11   +  10)            Side effect: i -= 1, i gets the value 10.
k = 21
```

Execution of the expression statement `--iRef;` under (1) proceeds as follows:

- The value in the `Integer` object referred to by the reference `iRef` is unboxed, resulting in the `int` value `11`.
- The value `11` is decremented, resulting in the value `10`.
- The value `10` is boxed in an `Integer` object, and this object's reference value is assigned to the reference `iRef`.
- The `int` value `10` of the expression statement is discarded.

Expressions where variables are modified multiple times during the evaluation should be avoided because the order of evaluation is not always immediately apparent.

We cannot associate increment and decrement operators. Given that `a` is a variable, we cannot write `(++(++a))`. The reason is that any operand to `++` must evaluate to a variable, but the evaluation of `(++a)` results in a value.

In the next example, both binary numeric promotion and an implicit narrowing conversion are performed to achieve the side effect of modifying the value of the operand. The `int` value of the expression `(++b)` (i.e., `11`) is assigned to the `int` variable `i`. The side effect of incrementing the value of the `byte` variable `b` requires binary numeric promotion to perform `int` addition, followed by an implicit narrowing conversion of the `int` value to `byte` to assign the value to variable `b`.

```
byte b = 10;
int  i = ++b;        // i is 11, and so is b.
```

The following example illustrates applying the increment operator to a floating-point operand. The side effect of the ++ operator is overwritten by the assignment.

```
double x = 4.5;
x = x + ++x;         // x gets the value 10.0.
```

Review Questions

2.1 Which of the following is not a legal comment in Java?
Select the one correct answer.

(a) `/* // */`
(b) `/* */ //`
(c) `// /* */`
(d) `/* /* */`
(e) `/* /* */ */`
(f) `// //`

2.2 What will be the result of compiling and running the following program?

```
public class Assignment {
  public static void main(String[] args) {
    int a, b, c;
    b = 10;
    a = b = c = 20;
    System.out.println(a);
  }
}
```

Select the one correct answer.

(a) The code will fail to compile because the compiler will report that the variable `c` in the multiple assignment statement `a = b = c = 20;` has not been initialized.

(b) The code will fail to compile because the multiple assignment statement `a = b = c = 20;` is illegal.
(c) The code will compile and print `10` at runtime.
(d) The code will compile and print `20` at runtime.

2.3 What will be the result of compiling and running the following program?

```
public class MyClass {
  public static void main(String[] args) {
    String a, b, c;
    c = new String("mouse");
    a = new String("cat");
    b = a;
    a = new String("dog");
    c = b;

    System.out.println(c);
  }
}
```

Select the one correct answer.

(a) The program will fail to compile.
(b) The program will print `mouse` at runtime.
(c) The program will print `cat` at runtime.
(d) The program will print `dog` at runtime.
(e) The program will randomly print either `cat` or `dog` at runtime.

2.4 Which of the following expressions evaluate to a floating-point value? Select the three correct answers.

(a) `2.0 * 3`
(b) `2 * 3`
(c) `2/3 + 5/7`
(d) `2.4 + 1.6`
(e) `0x10 * 1L * 300.0`

2.5 What is the value of the expression `(1 / 2 + 3 / 2 + 0.1)`? Select the one correct answer.

(a) `1`
(b) `1.1`
(c) `1.6`
(d) `2`
(e) `2.1`

2.6 What is the value of evaluating the following expression: `(- -1-3 * 10 / 5-1)`? Select the one correct answer.

(a) `-8`
(b) `-6`
(c) `7`
(d) `8`
(e) `10`
(f) None of the above

2.7 What is the result of compiling and running the following program?

```
public class Prog1 {
  public static void main(String[] args) {
    int k = 1;
    int i = ++k + k++ + + k;     // (1)
    System.out.println(i);
  }
}
```

Select the one correct answer.

(a) The program will fail to compile because of errors in the expression at (1).
(b) The program will print the value `3` at runtime.
(c) The program will print the value `4` at runtime.
(d) The program will print the value `7` at runtime.
(e) The program will print the value `8` at runtime.

2.8 Which is the first line that will cause a compile-time error in the following program?

```
public class MyClass {
  public static void main(String[] args) {
    char c;
    int i;
    c = 'a'; // (1)
    i = c;   // (2)
    i++;     // (3)
    c = i;   // (4)
    c++;     // (5)
  }
}
```

Select the one correct answer.

(a) (1)
(b) (2)
(c) (3)
(d) (4)
(e) (5)
(f) None of the above. The compiler will not report any errors.

2.9 What will be the result of compiling and running the following program?

```
public class EvaluationOrder {
  public static void main(String[] args) {
    int[] array = { 4, 8, 16 };
    int i = 1;
    array[++i] = --i;
    System.out.println(array[0] + array[1] + array[2]);
  }
}
```

Select the one correct answer.

(a) `13`
(b) `14`

(c) 20
(d) 21
(e) 24

2.11 Boolean Expressions

As the name implies, a boolean expression has the `boolean` data type and can only evaluate to the value `true` or `false`. Boolean expressions, when used as conditionals in control statements, allow the program flow to be controlled during execution.

Boolean expressions can be formed using *relational operators* (p. 75), *equality operators* (p. 76), *boolean logical operators* (p. 79), *conditional operators* (p. 81), the *assignment operator* (p. 55), and the `instanceof` operator (§5.11, p. 274).

2.12 Relational Operators: <, <=, >, >=

Given that `a` and `b` represent numeric expressions, the relational (also called *comparison*) operators are defined as shown in Table 2.23.

Table 2.23 *Relational Operators*

`a < b`	`a` less than `b`?
`a <= b`	`a` less than or equal to `b`?
`a > b`	`a` greater than `b`?
`a >= b`	`a` greater than or equal to `b`?

All relational operators are binary operators and their operands are numeric expressions. Binary numeric promotion is applied to the operands of these operators. The evaluation results in a `boolean` value. Relational operators have lower precedence than arithmetic operators, but higher than that of the assignment operators.

```
double hours = 45.5;
Double time = 18.0;             // Boxing of double value.
boolean overtime = hours >= 35; // true. Binary numeric promotion: double <-- int.
boolean beforeMidnight = time < 24.0;// true. Unboxing of value in time reference.
char letterA = 'A';
```

```
boolean order = letterA < 'a';  // true. Binary numeric promotion: int <-- char.
```

Relational operators are nonassociative. Mathematical expressions like $a \leq b \leq c$ must be written using relational and boolean logical/conditional operators.

```
int a = 1, b = 7, c = 10;
boolean illegal = a <= b <= c;       // (1) Illegal. Compile-time error!
boolean valid2 = a <= b && b <= c;   // (2) OK.
```

Since relational operators have left associativity, the evaluation of the expression `a <= b <= c` at (1) in these examples would proceed as follows: `((a <= b) <= c)`. Evaluation of `(a <= b)` would yield a `boolean` value that is not permitted as an operand of a relational operator; that is, (*boolean value* `<= c`) would be illegal.

2.13 Equality

We distinguish between *primitive data value equality*, *object reference equality*, and *object value equality*.

The equality operators have lower precedence than the relational operators, but higher precedence than the assignment operators.

Primitive Data Value Equality: ==, !=

Given that `a` and `b` represent operands of primitive data types, the primitive data value equality operators are defined as shown in Table 2.24.

Table 2.24 *Primitive Data Value Equality Operators*

`a == b`	Determines whether `a` and `b` are equal—that is, have the same primitive value (*equality*).
`a != b`	Determines whether `a` and `b` are not equal—that is, do not have the same primitive value (*inequality*).

The equality operator `==` and the inequality operator `!=` can be used to compare primitive data values, including `boolean` values. Binary numeric promotion may be applied to the non-boolean operands of these equality operators.

```
int year = 2002;
boolean isEven  = year % 2 == 0;     // true.
boolean compare = '1' == 1;          // false. Binary numeric promotion applied.
boolean test    = compare == false;  // true.
```

Care must be exercised when comparing floating-point numbers for equality, as an infinite number of floating-point values can be stored only as approximations in a finite number of bits. For example, the expression `(1.0 - 2.0/3.0 == 1.0/3.0)` returns `false`, although mathematically the result should be true.

Analogous to the discussion for relational operators, mathematical expressions like $a = b = c$ must be written using relational and logical/conditional operators. Since equality operators have left associativity, the evaluation of the expression `a == b == c` would proceed as follows: `((a == b) == c)`. Evaluation of `(a == b)` would yield a `boolean` value that *is* permitted as an operand of a data value equality operator, but (*boolean value* `== c`) would be illegal if `c` had a numeric type. This problem is illustrated in the following examples. The expression at (1) is illegal, but those at (2) and (3) are legal.

```
int a, b, c;
a = b = c = 5;
boolean illegal = a == b == c;              // (1) Illegal.
boolean valid2 = a == b && b == c;          // (2) Legal.
boolean valid3 = a == b == true;            // (3) Legal.
```

Object Reference Equality: ==, !=

The equality operator == and the inequality operator != can be applied to reference variables to test whether they refer to the same object. Given that `r` and `s` are reference variables, the reference equality operators are defined as shown in Table 2.25.

Table 2.25 *Reference Equality Operators*

`r == s`	Determines whether `r` and `s` are equal—that is, have the same reference value and therefore refer to the same object (also called *aliases*) (*equality*).
`r != s`	Determines whether `r` and `s` are not equal—that is, do not have the same reference value and therefore refer to different objects (*inequality*).

The operands must be cast compatible: It must be possible to cast the reference value of the one into the other's type; otherwise a compile-time error will result. Casting of references is discussed in §5.8, p. 266.

```
Pizza pizzaA = new Pizza("Sweet&Sour");      // new object
Pizza pizzaB = new Pizza("Sweet&Sour");      // new object
Pizza pizzaC = new Pizza("Hot&Spicy");       // new object

String banner = "Come and get it!";          // new object

boolean test  = banner == pizzaA;            // (1) Compile-time error
boolean test1 = pizzaA == pizzaB;            // false
boolean test2 = pizzaA == pizzaC;            // false

pizzaA = pizzaB;                        // Denote the same object; are aliases
boolean test3 = pizzaA == pizzaB;       // true
```

The comparison `banner == pizzaA` at (1) is illegal because the `String` and `Pizza` types are incompatible operand types as the reference value of one type cannot be cast to the other type. The values of `test1` and `test2` are `false` because the three references denote different objects, regardless of the fact that `pizzaA` and `pizzaB` are both sweet and sour pizzas. The value of `test3` is `true` because now both `pizzaA` and `pizzaB` denote the same object.

The equality and inequality operators are applied to object references to check whether two references denote the same object. The state of the objects that the references denote is not compared. This is the same as testing whether the references are *aliases*, meaning that they denote the same object.

The `null` literal can be assigned to any reference variable, and the reference value in a reference variable can be compared for equality with the `null` literal. The comparison can be used to avoid inadvertent use of a reference variable that does not denote any object.

```
if (objRef != null) {
    // ... use objRef ...
}
```

Note that only when the type of *both* operands is either a reference type or the `null` type do these operators test for object reference equality. Otherwise, they test for primitive data equality (see also §8.3, p. 442). In the following code snippet, binary numeric promotion involving unboxing is performed at (1):

```
Integer iRef = 10;
boolean b1 = iRef == null;         // Object reference equality
boolean b2 = iRef == 10;           // (1) Primitive data value equality
boolean b3 = null == 10;           // Compile-time error!
```

Object Value Equality

The `Object` class provides the method `public boolean equals(Object obj)`, which can be *overridden* (§5.1, p. 200) to give the right semantics of *object value equality*. The default implementation of this method in the `Object` class returns `true` only if the object is compared with itself, as if the equality operator == had been used to compare aliases of an object. Consequently, if a class does not override the semantics of the `equals()` method from the `Object` class, object value equality is the same as object reference equality.

Certain classes in the Java SE API override the `equals()` method, such as `java.lang.String` and the wrapper classes for the primitive data types. For two `String` objects, value equality means they contain identical character sequences. For the wrapper classes, value equality means the wrapper objects have the same primitive value and are of the same wrapper type (see also §8.3, p. 442).

```
// Equality for String objects means identical character sequences.
String movie1 = new String("The Revenge of the Exception Handler");
String movie2 = new String("High Noon at the Java Corral");
String movie3 = new String("The Revenge of the Exception Handler");
boolean test0 = movie1.equals(movie2);             // false.
boolean test1 = movie1.equals(movie3);             // true.

// Equality for wrapper classes means same type and same primitive value.
Boolean flag1 = true;                              // Boxing.
Boolean flag2 = false;                             // Boxing.
boolean test2 = flag1.equals("true");              // false. Not same type.
boolean test3 = flag1.equals(!flag2);              // true. Same type and value.
```

```
Integer iRef = 100;                                 // Boxing.
Short sRef = 100;                                   // Boxing <--- short <--- int
boolean test4 = iRef.equals(100);                   // true. Same type and value.
boolean test5 = iRef.equals(sRef);                  // false. Not same type.
boolean test6 = iRef.equals(3.14);                  // false. Not same type.

// The Pizza class does not override the equals() method, so we can use either
// equals() method inherited from the Object class or equality operator ==.
Pizza pizza1 = new Pizza("Veggies Delight");
Pizza pizza2 = new Pizza("Veggies Delight");
Pizza pizza3 = new Pizza("Cheese Delight");
boolean test7 = pizza1.equals(pizza2);              // false.
boolean test8 = pizza1.equals(pizza3);              // false.
boolean test9 = pizza1 == pizza2;                   // false.
pizza1 = pizza2;                                    // Creates aliases.
boolean test10 = pizza1.equals(pizza2);             // true.
boolean test11 = pizza1 == pizza2;                  // true.
```

2.14 Boolean Logical Operators: !, ^, &, |

Boolean logical operators include the unary operator ! (*logical complement*) and the binary operators & (*logical* AND), | (*logical inclusive* OR), and ^ (*logical exclusive* OR, also called *logical* XOR). These operators can be applied to `boolean` or `Boolean` operands, returning a `boolean` value. The operators &, |, and ^ can also be applied to integral operands to perform *bitwise* logical operations (p. 84).

Given that *x* and *y* represent boolean expressions, the boolean logical operators are defined in Table 2.26. The precedence of the operators decreases from left to right in the table.

These operators always evaluate *both* of the operands, unlike their counterpart *conditional operators* && and || (p. 81). Unboxing is applied to the operand values, if necessary. Truth values for boolean logical operators are shown in Table 2.26, where `x` and `y` are either of type `boolean` or `Boolean`.

Table 2.26 *Truth Values for Boolean Logical Operators*

x	y	Complement !x	AND x & y	XOR x ^ y	OR x | y
true	true	false	true	false	true
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Operand Evaluation for Boolean Logical Operators

In the evaluation of boolean expressions involving boolean logical AND, XOR, and OR operators, both the operands are evaluated. The order of operand evaluation is always from left to right.

```
if (i > 0 & i++ < 10) {/*...*/} // i will be incremented, regardless of value in i.
```

The binary boolean logical operators have lower precedence than the arithmetic and relational operators, but higher precedence than the assignment, conditional AND, and OR operators (p. 81). This is illustrated in the following examples:

```
boolean b1, b2, b3 = false, b4 = false;
Boolean b5 = true;
b1 = 4 == 2 & 1 < 4;          // false, evaluated as (b1 = ((4 == 2) & (1 < 4)))
b2 = b1 | !(2.5 >= 8);        // true
b3 = b3 ^ b5;                 // true, unboxing conversion on b5
b4 = b4 | b1 & b2;            // false
```

Here, the order of evaluation is illustrated for the last expression statement:

```
    (b4 = (b4 | (b1 & b2)))
⇒ (b4 = (false | (b1 & b2)))
⇒ (b4 = (false | (false & b2)))
⇒ (b4 = (false | (false & true)))
⇒ (b4 = (false | false))
⇒ (b4 = false)
⇒ false
```

Note that `b2` was evaluated, although strictly speaking, it was not necessary. This behavior is guaranteed for boolean logical operators.

Boolean Logical Compound Assignment Operators: &=, ^=, |=

Compound assignment operators for the boolean logical operators are defined in Table 2.27. The left-hand operand must be a boolean variable, and the right-hand operand must be a boolean expression. An identity conversion is applied implicitly on assignment. These operators can also be applied to integral operands to perform *bitwise* compound assignments (p. 84). See also the discussion on arithmetic compound assignment operators (p. 67).

Table 2.27 *Boolean Logical Compound Assignment Operators*

Expression	Given `a` and `b` are of type `boolean` or `Boolean`, the expression is evaluated as:
`b &= a`	`b = (b & (a))`
`b ^= a`	`b = (b ^ (a))`
`b |= a`	`b = (b | (a))`

Here are some examples to illustrate the behavior of boolean logical compound assignment operators:

```
boolean b1 = false, b2 = true, b3 = false;
Boolean b4 = false;
b1 |= true;             // true
b4 ^= b1;               // (1) true, unboxing in (b4 ^ (b1)), boxing on assignment
b3 &= b1 | b2;          // (2) false, b3 = (b3 & (b1 | b2))
b3 = b3 & b1 | b2;      // (3) true,  b3 = ((b3 & b1) | b2)
```

The assignment at (1) entails unboxing to evaluate the expression on the right-hand side, followed by boxing to assign the `boolean` result. It is also instructive to compare how the assignments at (2) and (3) are performed, as they lead to different results with the same values of the operands, showing how the precedence affects the evaluation.

2.15 Conditional Operators: &&, ||

The conditional operators `&&` and `||` are similar to their counterpart logical operators & and |, except that their evaluation is *short-circuited*. Given that `x` and `y` represent values of `boolean` or `Boolean` expressions, the conditional operators are defined in Table 2.28. In the table, the operators are listed in decreasing order of precedence.

Table 2.28 *Conditional Operators*

Conditional AND	`x && y`	`true` if both operands are true; otherwise `false`.
Conditional OR	`x || y`	`true` if either or both operands are true; otherwise `false`.

Unlike their logical counterparts & and |, which can also be applied to integral operands for bitwise operations, the conditional operators `&&` and `||` can be applied to only `boolean` operands. Their evaluation results in a `boolean` value. Truth values for conditional operators are shown in Table 2.29. Not surprisingly, the conditional operators have the same truth values as their counterpart logical operators. However, unlike with their logical counterparts, there are no compound assignment operators for the conditional operators.

Table 2.29 *Truth Values for Conditional Operators*

`x`	`y`	AND `x && y`	OR `x || y`
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`

Short-Circuit Evaluation

In the evaluation of boolean expressions involving conditional AND and OR, the left-hand operand is evaluated before the right-hand operand, and the evaluation is short-circuited (i.e., if the result of the boolean expression can be determined from the left-hand operand, the right-hand operand is not evaluated). In other words, the right-hand operand is evaluated conditionally.

The binary conditional operators have lower precedence than the arithmetic, relational, and logical operators, but higher precedence than the assignment operators. Unboxing of the operand value takes place when necessary, before the operation is performed. The following examples illustrate usage of conditional operators:

```
Boolean b1 = 4 == 2 && 1 < 4;   // false, short-circuit evaluated as
                                // (b1 = ((4 == 2) && (1 < 4)))
boolean b2 = !b1 || 2.5 > 8;    // true, short-circuit evaluated as
                                // (b2 = ((!b1) || (2.5 > 8)))
Boolean b3 = !(b1 && b2);       // true
boolean b4 = b1 || !b3 && b2;   // false, short-circuit evaluated as
                                // (b4 = (b1 || ((!b3) && b2)))
```

The order of evaluation for computing the value stored in the boolean variable `b4` proceeds as follows:

```
    (b4 = (b1 || ((!b3) && b2)))
⇒ (b4 = (false || ((!b3) && b2)))
⇒ (b4 = (false || ((!true) && b2)))
⇒ (b4 = (false || ((false) && b2)))
⇒ (b4 = (false || false))
⇒ (b4 = false)
```

Note that `b2` is not evaluated, short-circuiting the evaluation. Example 2.3 illustrates the short-circuit evaluation of the initialization expressions in the declaration statements given in the code snippet above. In addition, it shows an evaluation (see the declaration of `b5`) involving boolean logical operators that always evaluate both operands. The output shows how many operands were evaluated for each expression. See also Example 2.1, p. 54, which uses a similar approach to illustrate the order of operand evaluation in arithmetic expressions.

Example 2.3 *Short-Circuit Evaluation Involving Conditional Operators*

```
public class ShortCircuit {
  public static void main(String[] args) {
    // Boolean b1 = 4 == 2 && 1 < 4;
    Boolean b1 = operandEval(1, 4 == 2) && operandEval(2, 1 < 4);
    System.out.println("Value of b1: " + b1);

    // boolean b2 = !b1 || 2.5 > 8;
    boolean b2 = !operandEval(1, b1) || operandEval(2, 2.5 > 8);
    System.out.println("Value of b2: " + b2);
```

```
    // Boolean b3 = !(b1 && b2);
    Boolean b3 = !(operandEval(1, b1) && operandEval(2, b2));
    System.out.println("Value of b3: " + b3);

    // boolean b4 = b1 || !b3 && b2;
    boolean b4 = operandEval(1, b1) || !operandEval(2, b3) && operandEval(3, b2);
    System.out.println("Value of b4: " + b4);

    // boolean b5 = b1 | !b3 & b2;    // Using boolean logical operators
    boolean b5 = operandEval(1, b1) | !operandEval(2, b3) & operandEval(3, b2);
    System.out.println("Value of b5: " + b5);
  }

  static boolean operandEval(int opNum, boolean operand) {                // (1)
    System.out.println(opNum);
    return operand;
  }
}
```

Output from the program:

```
1
Value of b1: false
1
Value of b2: true
1
Value of b3: true
1
2
Value of b4: false
1
2
3
Value of b5: false
```

Short-circuit evaluation can be used to ensure that a reference variable denotes an object before it is used.

```
if (objRef != null && objRef.equals(other)) { /*...*/ }
```

The method call is now conditionally dependent on the left-hand operand and will not be executed if the variable `objRef` has the null reference. If we use the logical `&` operator and the variable `objRef` has the null reference, evaluation of the right-hand operand will result in a `NullPointerException`.

In summary, we employ the conditional operators `&&` and `||` if the evaluation of the right-hand operand is conditionally dependent on the left-hand operand. We use the boolean logical operators `&` and `|` if both operands must be evaluated. The subtlety of conditional operators is illustrated by the following examples:

```
if (i > 0 && i++ < 10) {/*...*/}   // i is not incremented if i > 0 is false.
if (i > 0 || i++ < 10) {/*...*/}   // i is not incremented if i > 0 is true.
```

2.16 Integer Bitwise Operators: ~, &, |, ^

A review of integer representation (p. 34) is recommended before continuing with this section on how integer bitwise operators can be applied to values of *integral* data types.

Integer bitwise operators include the unary operator ~ (*bitwise complement*) and the binary operators & (*bitwise* AND), | (*bitwise inclusive* OR), and ^ (*bitwise exclusive* OR, also known as *bitwise* XOR). The operators &, |, and ^ are overloaded, as they can be applied to `boolean` or `Boolean` operands to perform *boolean* logical operations (p. 79).

The binary bitwise operators perform bitwise operations between corresponding individual bit values in the operands. Unary numeric promotion is applied to the operand of the unary bitwise complement operator ~, and binary numeric promotion is applied to the operands of the binary bitwise operators. The result is a new integer value of the promoted type, which can be either `int` or `long`.

Given that `A` and `B` are corresponding bit values (either `0` or `1`) in the left-hand and right-hand operands, respectively, these bitwise operators are defined as shown in Table 2.30. The operators are listed in order of decreasing precedence.

Table 2.30 *Integer Bitwise Operators*

Operator name	Notation	Effect on each bit of the binary representation
Bitwise complement	`~A`	Invert the bit value: `1` to `0`, `0` to `1`.
Bitwise AND	`A & B`	`1` if both bits are `1`; otherwise `0`.
Bitwise OR	`A | B`	`1` if either or both bits are `1`; otherwise `0`.
Bitwise XOR	`A ^ B`	`1` if and only if one of the bits is `1`; otherwise `0`.

The result of applying bitwise operators between two corresponding bits in the operands is shown in Table 2.31, where `A` and `B` are corresponding bit values in the left-hand and right-hand operands, respectively. Table 2.31 is analogous to Table 2.26 for boolean logical operators, if we consider bit value `1` to represent `true` and bit value `0` to represent `false`.

Table 2.31 *Result Table for Bitwise Operators*

A	B	Complement ~A	AND A & B	XOR A ^ B	OR A | B
1	1	0	1	0	1
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

Examples of Bitwise Operator Application

```
char v1 = ')';          // Unicode value 41
byte v2 = 13;

int result1 = ~v1;      // -42
int result2 = v1 & v2;  // 9
int result3 = v1 | v2;  // 45
int result4 = v1 ^ v2;  // 36
```

Table 2.32 shows how the result is calculated. Unary and binary numeric promotions are applied first, converting the operands to `int` in these cases. Note that the operator semantics are applied to corresponding individual bits—that is, the first bit of the left-hand operand and the first bit of the right-hand operand, the second bit of the left-hand operand and the second bit of the right-hand operand, and so on.

Table 2.32 *Examples of Bitwise Operations*

~v1	v1 & v2	v1 | v2	v1 ^ v2
~ 0...0010 1001	0...0010 1001	0...0010 1001	0...0010 1001
	& 0...0000 1101	| 0...0000 1101	^ 0...0000 1101
= 1...1101 0110	= 0...0000 1001	= 0...0010 1101	= 0...0010 0100
= 0xfffffffd6	= 0x00000009	= 0x0000002d	= 0x00000024
= -42	= 9	= 45	= 36

It is instructive to run examples and print the result of a bitwise operation in different notations, as shown in Example 2.4. The integer bitwise operators support a programming technique called *bit masking*. The value `v2` is usually called a *bit mask*. Depending on the bitwise operation performed on the value `v1` and the mask `v2`, we see how the resulting value reflects the bitwise operation performed between the individual corresponding bits of the value `v1` and the mask `v2`. By choosing appropriate values for the bits in the mask `v2` and the right bitwise operation, it is possible to extract, set, and toggle specific bits in the value `v1`.

Methods for converting integers to strings in different notations can be found in the `Integer` class (§8.3, p. 445).

Example 2.4 *Bitwise Operations*

```
public class BitOperations {
  public static void main(String[] args) {
    char v1 = ')';                      // Unicode value 41
    byte v2 = 13;
    printIntToStr("v1:", v1);           // 41
    printIntToStr("v2:", v2);           // 13
    printIntToStr("~v1:", ~v1);         // -42
    printIntToStr("v1 & v2:", v1 & v2); // 9
    printIntToStr("v1 | v2:", v1 | v2); // 45
```

```
    printIntToStr("v1 ^ v2:", v1 ^ v2); // 36
  }

  public static void printIntToStr(String label, int result) {
    System.out.println(label);
    System.out.println("    Binary:  " + Integer.toBinaryString(result));
    System.out.println("    Hex:     " + Integer.toHexString(result));
    System.out.println("    Decimal: " + result);
  }
}
```

Output from the program:

```
v1:
    Binary:  101001
    Hex:     29
    Decimal: 41
v2:
    Binary:  1101
    Hex:     d
    Decimal: 13
~v1:
    Binary:  11111111111111111111111111010110
    Hex:     ffffffd6
    Decimal: -42
v1 & v2:
    Binary:  1001
    Hex:     9
    Decimal: 9
v1 | v2:
    Binary:  101101
    Hex:     2d
    Decimal: 45
v1 ^ v2:
    Binary:  100100
    Hex:     24
    Decimal: 36
```

Bitwise Compound Assignment Operators: &=, ^=, |=

Bitwise compound assignment operators for the bitwise operators are defined in Table 2.33. Type conversions for these operators, when applied to integral operands, are the same as for other compound assignment operators: An implicit narrowing conversion is performed on assignment when the destination data type is either `byte`, `short`, or `char`. These operators can also be applied to boolean operands to perform logical compound assignments (p. 80).

Table 2.33 *Bitwise Compound Assignment Operators*

Expression	Given `T` is the integral type of `b`, the expression is evaluated as:
`b &= a`	`b = (T) ((b) & (a))`
`b ^= a`	`b = (T) ((b) ^ (a))`

Table 2.33 *Bitwise Compound Assignment Operators (Continued)*

Expression	Given T is the integral type of b, the expression is evaluated as:
`b |= a`	`b = (T) ((b) | (a))`

Examples of Bitwise Compound Assignment

```
int  v0 = -42;
char v1 = ')';  // 41
byte v2 = 13;

v0 &= 15;       //     1...1101 0110 & 0...0000 1111 => 0...0000 0110 (= 6)
v1 |= v2;       // (1) 0...0010 1001 | 0...0000 1101 => 0...0010 1101 (= 45, '-')
```

At (1) in these examples, both the `char` value in `v1` and the `byte` value in `v2` are first promoted to `int`. The result is implicitly narrowed to the destination type `char` on assignment.

2.17 Shift Operators: <<, >>, >>>

The binary shift operators return a new value formed by shifting bits either left or right a specified number of times in a given integral value. The number of shifts (also called the *shift distance*) is given by the right-hand operand, and the value that is to be shifted is given by the left-hand operand. Note that *unary* numeric promotion is applied to each operand *individually*. The value returned has the promoted type of the left-hand operand. Also, the value of the left-hand operand is *not* affected by applying the shift operator.

The shift distance is calculated by AND-ing the value of the right-hand operand with a mask value of `0x1f` (31) if the left-hand operand has the promoted type `int`, or using a mask value of `0x3f` (63) if the left-hand operand has the promoted type `long`. This effectively means masking the five lower bits of the right-hand operand in the case of an `int` left-hand operand, and masking the six lower bits of the right-hand operand in the case of a `long` left-hand operand. Thus the shift distance is always in the range 0 to 31 when the promoted type of the left-hand operand is `int` (which has size 32 bits), and in the range 0 to 63 when the promoted type of the left-hand operand is `long` (which has size 64 bits).

Given that `v` contains the value whose bits are to be shifted and `n` specifies the number of bits to shift, the bitwise operators are defined in Table 2.34. It is implied that the value `n` in Table 2.34 is subject to the shift distance calculation outlined above, and that the shift operations are always performed on the value of the left-hand operand represented in two's complement.

Table 2.34 *Shift Operators*

Shift left	`v << n`	Shift all bits in `v` left `n` times, filling with `0` from the right.

Table 2.34 *Shift Operators (Continued)*

Shift right with sign bit	`v >> n`	Shift all bits in `v` right `n` times, filling with the sign bit from the left.
Shift right with zero fill	`v >>> n`	Shift all bits in `v` right `n` times, filling with `0` from the left.

Since `char`, `byte`, and `short` operands are promoted to `int`, the result of applying these bitwise operators is always either an `int` or a `long` value. Care must be taken in employing a cast to narrow the resulting value, as this can result in a loss of information as the upper bits are discarded during conversion.

Note that regardless of the promotion of the values in the operands or determination of the shift distance, the operands `v` and `n` are not affected by these three shift operators. However, the shift compound assignment operators, discussed in this section, can change the value of the left-hand operand `v`.

Bit values shifted out (*falling off*) from bit `0` or the most significant bit are lost. Since bits can be shifted both left and right, a positive value when shifted can result in a negative value, and vice versa.

The Shift-Left Operator <<

As the bits are shifted left, zeros are always filled in from the right.

```
int i = 12;
int result = i << 4;    // 192
```

The bits in the `int` value for `i` are shifted left four places as follows:

```
i << 4
= 0000 0000 0000 0000 0000 0000 0000 1100 << 4
= 0000 0000 0000 0000 0000 0000 1100 0000
= 0x000000c0
= 192
```

Each left-shift corresponds to multiplication of the value by 2. In the above example, `12 * 2`^{4} is `192`.

The sign bit of a `byte` or `short` value is extended to fill the higher bits when the value is promoted, as illustrated by the example below:

```
byte  b = -42;          // 11010110
short n = 4;
int   result = b << n;  // -672
```

The values of the two operands, `b` and `n`, in the previous example are promoted individually. The `short` value in `n` is promoted to `int`. The `byte` value in `b`, after promotion to `int`, is shifted left `4` places:

```
b << n
= 1101 0110 << 0000 0000 0000 0100
= 1111 1111 1111 1111 1111 1111 1101 0110 <<0000 0000 0000 0000 0000 0000 0000 0100
= 1111 1111 1111 1111 1111 1111 1101 0110 << 4
```

```
= 1111 1111 1111 1111 1111 1101 0110 0000
= 0xfffffd60
= -672
```

In the above example, `-42 * 2^4` is `-672`.

Care must also be taken when assigning the result of a shift operator to a narrower data type.

```
byte a = 32;
int  j = a << 3;            // 256
byte b = (byte) (a << 3);   // 0. Cast mandatory.
```

The result of `(a << 3)` is `256`.

```
a << 3
= 0000 0000 0000 0000 0000 0000 0010 0000 << 3
= 0000 0000 0000 0000 0000 0001 0000 0000
= 0x00000100
= 256
```

The value that `j` gets is `256`, but the value that `b` gets is `0`, as the higher bits are discarded in the explicit narrowing conversion.

The examples above do not show how the shift distance is determined. It is obvious from the value of the right-hand operand, which is within the range 0 to 31, inclusive. An example with the shift-left operator, where the value of the right-hand operand is out of range, is shown below.

```
12 << 36
= 0000 0000 0000 0000 0000 0000 0000 1100 << (0...0010 0100 & 0001 1111)
= 0000 0000 0000 0000 0000 0000 0000 1100 << 0...0000 0100
= 0000 0000 0000 0000 0000 0000 0000 1100 << 4
= 0000 0000 0000 0000 0000 0000 1100 0000
= 0x000000c0
= 192
```

The value of the right-hand operand, `36`, is AND-ed with the mask `11111` (i.e., `31`, `0x1f`), giving the shift distance `4`. This is the same as `(36 % 32)`. It is not surprising that `(12 << 36)` is equal to `(12 << 4)` (i.e., `192`).

The Shift-Right-with-Sign-Fill Operator >>

As the bits are shifted right, the sign bit (the most significant bit) is used to fill in from the left. So, if the left-hand operand is a positive value, zeros are filled in from the left, but if the operand is a negative value, ones are filled in from the left.

```
int i = 12;
int result = i >> 2;    // 3
```

The value for `i` is shifted right with sign-fill two places.

```
i >> 2
= 0000 0000 0000 0000 0000 0000 0000 1100 >> 2
= 0000 0000 0000 0000 0000 0000 0000 0011
= 0x00000003
```

```
= 3
```

Each right-shift corresponds to integer division of the value by 2, but this can give unexpected results if care is not exercised, as bits start falling off. In the above example, `12 / 2`^{2} is `3`.

Similarly, when a negative value is shifted right, ones are filled in from the left.

```
byte b = -42;          // 11010110
int result = b >> 4;   // -3
```

The `byte` value for `b`, after promotion to `int`, is shifted right with sign-fill four places.

```
b >> 4
= 1111 1111 1111 1111 1111 1111 1101 0110 >> 4
= 1111 1111 1111 1111 1111 1111 1111 1101
= 0xfffffffd
= -3
```

In the following example, the right-hand operand has a negative value:

```
-42 >> -4
= 1111 1111 1111 1111 1111 1111 1101 0110 >> (1...1111 1100 & 0001 1111)
= 1111 1111 1111 1111 1111 1111 1101 0110 >> 0...0001 1100
= 1111 1111 1111 1111 1111 1111 1101 0110 >> 28
= 1111 1111 1111 1111 1111 1111 1111 1111
= 0xffffffff
= -1
```

The value of the right-hand operand, `-4`, is AND-ed with the mask `11111` (i.e., `31`, `0x1f`), giving the shift distance `28`. This is the same as `(-4 % 32)`. The value of `(-42 >> -4)` is equivalent to `(-42 >> 28)`.

The Shift-Right-with-Zero-Fill Operator >>>

As the bits are shifted right, zeros are filled in from the left, regardless of whether the operand has a positive or a negative value.

Obviously, for positive values, the shift-right-with-zero-fill `>>>` and shift-right-with-sign-fill `>>` operators are equivalent. The expression `(12 >> 2)` and the expression `(12 >>> 2)` return the same value:

```
12 >>> 2
= 0000 0000 0000 0000 0000 0000 0000 1100 >>> 2
= 0000 0000 0000 0000 0000 0000 0000 0011
= 0x00000003
= 3
```

Individual unary numeric promotion of the left-hand operand is shown in the following example:

```
byte b = -42;              // 1101 0110
int result = b >>> 4;      // 268435453
```

It is instructive to compare the value of the expression (-42 >>> 4) with that of the expression (-42 >> 4), which has the value -3. The byte value for b, after unary numeric promotion to int, is shifted right with zero-fill four places.

```
b >>> 4
= 1111 1111 1111 1111 1111 1111 1101 0110 >>> 4
= 0000 1111 1111 1111 1111 1111 1111 1101
= 0x0fffffd
= 268435453
```

In the following example, the value of the right-hand operand is out of range, resulting in a shift distance of 28 (as we have seen before):

```
-42 >>> -4
= 1111 1111 1111 1111 1111 1111 1101 0110 >>> 28
= 0000 0000 0000 0000 0000 0000 0000 1111
= 0x0000000f
= 15
```

Shift Compound Assignment Operators: <<=, >>=, >>>=

Table 2.35 lists shift compound assignment operators. Type conversions for these operators, when applied to integral operands, are the same as for other compound assignment operators: An implicit narrowing conversion is performed on assignment when the destination data type is either byte, short, or char.

Table 2.35 *Shift Compound Assignment Operators*

Expression	Given T as the integral type of v, the expression is evaluated as:
v <<= n	v = (T) ((v) << (n))
v >>= n	v = (T) ((v) >> (n))
v >>>= n	v = (T) ((v) >>> (n))

Examples of Shift Compound Assignment Operators

```
int i = -42;
i >>= 4;                    // 1...1101 0110 >> 4 => 1...1111 1101 (= -3).

byte a = 12;
a <<= 5;                    // (1) -128. Evaluated as a = (byte)((int)a << 5)
a = a << 5;                 // Compile-time error! Needs explicit cast.
```

The example at (1) illustrates the truncation that takes place on narrowing to the destination type. The byte value in a is first promoted to int (by applying unary numeric promotion in this case), then shifted left five places, followed by implicit narrowing to byte:

```
a = (byte) (a << 5)
  = (byte) (0000 0000 0000 0000 0000 0000 0000 1100 << 5)
  = (byte)  0000 0000 0000 0000 0000 0001 1000 0000
  = 1000 0000
  = 0x80
```

```
= -128
```

2.18 The Conditional Operator ?:

The ternary conditional operator ?: allows *conditional expressions* to be defined. The conditional expression has the following syntax:

condition ? $expression_1$: $expression_2$

It is called ternary because it has three operands. If the boolean expression *condition* is `true`, then $expression_1$ is evaluated; otherwise, $expression_2$ is evaluated. Both $expression_1$ and $expression_2$ must evaluate to values that can be converted to the *type* of the conditional expression. This type is determined from the types of the two expressions. The value of the evaluated expression is converted to the type of the conditional expression, and may involve autoboxing and unboxing.

Evaluation of a conditional expression is an example of short-circuit evaluation. As only one of the two expressions is evaluated, one should be wary of any side effects in a conditional expression.

In the following code snippet at (1), both expressions in the conditional expression are of type `byte`. The type of the conditional expression is therefore `byte`. That a value of type `byte` can be converted to an `int` by an implicit widening numeric conversion to be assignment compatible with the `int` variable `daysInFebruary` is secondary in determining the type of the conditional expression. Note that the conditional operator at (1) has higher precedence than the assignment operator `=`, making it unnecessary to enclose the conditional expression in parentheses.

```
boolean leapYear = false;
byte v29 = 29;
byte v28 = 28;
int daysInFebruary = leapYear ? v29 : v28;   // (1)
```

The following examples illustrate the use of conditional expressions. The type of the conditional expression at (2) is `int`, and no conversion of any expression value is necessary. The type of the conditional expression at (3) is `double`, due to binary numeric promotion: The `int` value of the first expression is promoted to a `double`. The compiler reports an error because a `double` cannot be assigned to an `int` variable. The type of the conditional expression at (4) is also `double` as at (3), but now the `double` value is assignment compatible with the `double` variable `minDoubleValue`.

```
int i = 3;
int j = 4;
int minValue1 = i < j ? i : j;                          // (2) int
int minValue2 = i < j ? i : Double.MIN_VALUE;           // (3) double. Not OK.
double minDoubleValue = i < j ? i : Double.MIN_VALUE; // (4) double
```

At (5) below, the primitive values of the expressions can be boxed and assigned to an `Object` reference. At (6), the `int` value of the first expression can be boxed in an

Integer. The println() method creates and prints a text representation of any object whose reference value is passed as a parameter.

```
// Assume i and j are of type int and initialized correctly.
Object obj = i < j ? i : true;        // (5) value of i boxed in Integer or
                                      //     literal true boxed in Boolean
System.out.println(i < j ? i : "Hi"); // (6) value of i boxed in Integer or
                                      //     String object "Hi"
```

The conditional expression is *not* an expression statement. The following code will not compile:

```
(i < j) ? i : j;    // Compile-time error!
```

The conditional expression can be nested, and the conditional operator associates from right to left.

a?b:c?d:e?f:g evaluates as (a?b:(c?d:(e?f:g)))

The value of this conditional expression is g if, and only if, a, c, and e are false. A nested conditional expression is used in the next example. As a convention, the condition in a conditional expression is enclosed in parentheses to aid in reading the code. Typically, a conditional expression is used when it makes the code easier to read, especially when the expressions are short and without side effects.

```
int n = 3;
String msg = (n==0) ? "no cookies." : (n==1) ? "one cookie." : "many cookies.";
System.out.println("You get " + msg); // You get many cookies.
```

The conditional operator is the expression equivalent of the if-else statement (§4.1, p. 155).

2.19 Other Operators: new, [], instanceof, ->

The new operator is used to create objects, such as instances of classes and arrays. It is used with a constructor call to instantiate classes (§5.11, p. 274) and with the [] notation to create arrays (§3.9, p. 119). It is also used to instantiate anonymous arrays (§3.9, p. 123).

```
Pizza onePizza = new Pizza();       // Create an instance of the Pizza class.
```

The [] notation is used to declare and construct arrays, and is also used to access array elements (§3.9, p. 121).

```
int[] anArray = new int[5];// Declare and construct an int array of 5 elements.
anArray[4] = anArray[3];   // Element at index 4 gets value of element at index 3.
```

The boolean, binary, and infix operator instanceof is used for either type comparison or pattern matching (§5.11, p. 274).

```
Pizza myPizza = new Pizza();
boolean test1 = myPizza instanceof Pizza; // true.
boolean test2 = "Pizza" instanceof Pizza; // Compile-time error. Incompatible
                                          // operand types.
```

```
boolean test3 = null instanceof Pizza; // Always false. null is not an instance.
```

The arrow operator -> is used in a `switch` statement (§4.2, p. 157), in a `switch` expression (§4.3, p. 166), and in the definition of a lambda expression (§13.2, p. 695).

```
java.util.function.Predicate<String> predicate = str -> str.length() % 2 == 0;
boolean test4 = predicate.test("The lambda strikes back!");    // true.
```

Review Questions

2.10 Which of the following statements are true?
Select the two correct answers.

(a) The remainder operator `%` can be used only with integral operands.
(b) Short-circuit evaluation occurs with boolean logical operators.
(c) The arithmetic operators `*`, `/`, and `%` have the same level of precedence.
(d) A `short` value ranges from `-128` to `+127`, inclusive.
(e) `(+15)` is a legal expression.

2.11 Which of the following statements are true about the lines of output printed by the following program?

```
public class BoolOp {
  static void op(boolean a, boolean b) {
    boolean c = a != b;
    boolean d = a ^ b;
    boolean e = c == d;
    System.out.println(e);
  }

  public static void main(String[] args) {
    op(false, false);
    op(true, false);
    op(false, true);
    op(true, true);
  }
}
```

Select the three correct answers.

(a) All lines printed are the same.
(b) At least one line contains `false`.
(c) At least one line contains `true`.
(d) The first line contains `false`.
(e) The last line contains `true`.

2.12 What is the result of running the following program?

```
public class OperandOrder {
  public static void main(String[] args) {
    int i = 0;
    int[] a = {3, 6};
    a[i] = i = 9;
    System.out.println(i + " " + a[0] + " " + a[1]);
```

```
  }
}
```

Select the one correct answer.

(a) When run, the program throws an `ArrayIndexOutOfBoundsException`.
(b) When run, the program will print `9 9 6`.
(c) When run, the program will print `9 0 6`.
(d) When run, the program will print `9 3 6`.
(e) When run, the program will print `9 3 9`.

2.13 Which of the following statements are true about the output from the following program?

```
public class Logic {
  public static void main(String[] args) {
    int i = 0;
    int j = 0;

    boolean t = true;
    boolean r;

    r = (t &  0 < (i+=1));
    r = (t && 0 < (i+=2));
    r = (t |  0 < (j+=1));
    r = (t || 0 < (j+=2));
    System.out.println(i + " " + j);
  }
}
```

Select the two correct answers.

(a) The first digit printed is 1.
(b) The first digit printed is 2.
(c) The first digit printed is 3.
(d) The second digit printed is 1.
(e) The second digit printed is 2.
(f) The second digit printed is 3.

2.14 Given the following code:

```
int x = 1, y = 2, z = 3;
if (x < y || ++z > 4) {
  System.out.println("a" + x + y + z);
}
if (x < y && ++z > 4) {
  System.out.println("b" + x + y + z);
}
```

What will be the output?
Select the one correct answer.

(a) `a124`
    `b125`
(b) `a123`

(c)
```
a123
b124
```

2.15 Which of the following statements when inserted at (1) will not result in a compile-time error?

```
public class RQ05A200 {
  public static void main(String[] args) {
    int i = 20;
    int j = 30;
    // (1) INSERT STATEMENT HERE
  }
}
```

Select the three correct answers.

(a) `int result1 = i < j ? i : j * 10d;`
(b) `int result2 = i < j ? { ++i } : { ++j };`
(c) `Number number = i < j ? i : j * 10D;`
(d) `System.out.println(i < j ? i);`
(e) `System.out.println(i < j ? ++i : ++j);`
(f) `System.out.println(i == j ? i == j : "i not equal to j");`

2.16 Which of the following statements are true about the following code?

```
public class RQ05A100 {
  public static void main(String[] args) {
    int n1 = 10, n2 = 10;
    int m1 = 20, m2 = 30;
    int result = n1 != n2? n1 : m1 != m2? m1 : m2;
    System.out.println(result);
  }
}
```

Select the one correct answer.

(a) The program will fail to compile.
(b) The program will throw an `ArithmeticException` at runtime.
(c) The program will compile and print `10` when run.
(d) The program will compile and print `20` when run.
(e) The program will compile and print `30` when run.