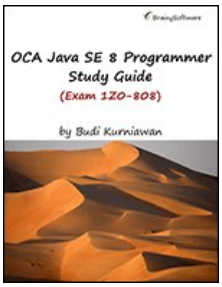

OCA Java SE 8 Programmer Study Guide (Exam 1Z0-808)

by Budi Kurniawan

Chapter 1: Language Fundamentals

Overview

Java is an object-oriented programming (OOP) language, therefore an understanding of OOP is of utmost importance. Chapter 3, "Objects and Classes" is the first lesson of OOP in this book. However, before you explore OOP features and techniques, you should first study Java language fundamentals.

1.1 ASCII and Unicode

Traditionally, computers in English speaking countries only used the ASCII (American Standard Code for Information Interchange) character set to represent alphanumeric characters. Each character in the ASCII is represented by 7 bits. There are therefore 128 characters in this character set. These include the lower case and upper case Latin letters, numbers, and punctuation marks.

The ASCII character set was later extended to include another 128 characters, such as the German characters ä, ö, ü and the British currency symbol £. This character set is called extended ASCII and each character is represented by 8 bits.

ASCII and the extended ASCII are only two of the many character sets available. Another popular one is the character set standardized by the ISO (International Standards Organization), ISO-8859-1, which is also known as Latin-1. Each character in ISO-8859-1 is represented by eight bits as well. This character set contains all the characters required for writing text in many of the western European languages, such as German, Danish, Dutch, French, Italian, Spanish, Portuguese and, of course, English. An eight-bit-per-character character set is convenient because a byte is also 8 bits long. As such, storing and transmitting text written in an 8-bit character set is most efficient.

However, not every language uses Latin letters. Chinese and Japanese are examples of languages that use different character sets. For example, each character in the Chinese language represents a word, not a letter. There are thousands of these characters and eight bits are not enough to represent all the characters in the character set. The Japanese use a different character set for their language too. In total, there are hundreds of different character sets for all the world languages. To unify all these characters sets, a computing standard called Unicode was created.

Unicode is a character set developed by a non-profit organization called the Unicode Consortium (www.unicode.org). This body attempts to include all characters in all languages in the world into one single character set. A unique number in Unicode represents exactly one character. Currently at version 7, Unicode is used in Java, XML, ECMAScript, LDAP, etc.

Initially, a Unicode character was represented by 16 bits, which were enough to represent more than 65,000 different characters. 65,000 characters are sufficient for encoding most of the characters in major languages in the world. However, the Unicode consortium planned to allow for encoding for as many as a million more characters. With this amount, you then need more than 16 bits to represent each character. In fact, a 32 bit system is considered a convenient way of storing Unicode characters.

Now, you see a problem already. While Unicode provides enough space for all the characters used in all languages, storing and transmitting Unicode text is not as efficient as storing and transmitting ASCII or Latin-1 characters. In the Internet world, this is a huge problem. Imagine having to transfer 4 times as much data as ASCII text!

Fortunately, character encoding can make it more efficient to store and transmit Unicode text. You can think of character encoding as analogous to data compression. And, there are many types of character encodings available today. The Unicode Consortium endorses three of them:

- UTF-8. This is popular for HTML and for protocols whereby Unicode characters are transformed into a variable length encoding of bytes. It has the advantages that the Unicode characters corresponding to the familiar ASCII set have the same byte values as ASCII, and that Unicode characters transformed into UTF-8 can be used with much existing software. Most browsers support the UTF-8 character encoding.
- UTF-16. In this character encoding, all the more commonly used characters fit into a single 16-bit code unit, and other less often used characters are accessible via pairs of 16-bit code units.
- UTF-32. This character encoding uses 32 bits for every single character. This is clearly not a choice for Internet applications. At least, not at present.

ASCII characters still play a dominant role in software programming. Java too uses ASCII for almost all input elements, except

comments, identifiers, and the contents of characters and strings. For the latter, Java supports Unicode characters. This means, you can write comments, identifiers, and strings in languages other than English.

1.2 Separators

Java uses certain characters as separators. These special characters are presented in Table 1.1.

Table 1.1: Java separators

Symbol	Name	Description
( )	Parentheses	Used in: 1. method signatures to contain lists of arguments. 2. expressions to raise operator precedence. 3. narrowing conversions. 4. loops to contain expressions to be evaluated
{ }	Braces	Used in: 1. declaration of types. 2. blocks of statements 3. array initialization.
[ ]	Brackets	Used in: 1. array declaration. 2. array value dereferencing
< >	Angle brackets	Used to pass parameter to parameterized types.
;	Semicolon	Used to terminate statements and in the **for** statement to separate the initialization code, the expression, and the update code.
:	Colon	Used in the **for** statement that iterates over an array or a collection.
,	Comma	Used to separate arguments in method declarations.
.	Period	Used to separate package names from subpackages and type names, and to separate a field or method from a reference variable.

It is important that you are familiar with the symbols and names, but don't worry if you don't understand the terms in the Description column for now.

1.3 Primitives

When writing an object-oriented (OO) application, you create an object model that resembles the real world. For example, a payroll application would have **Employee** objects, **Tax** objects, **Company** objects, etc. In Java, however, objects are not the only data type. There is another data type called *primitive*. There are eight primitive types in Java, each with a specific format and size. Table 1.2 lists Java primitives.

Table 1.2: Java primitives

Primitive	Description	Range
byte	Byte-length integer (8 bits)	-128 (-2^7) to 127 (2^7-1)
short	Short integer (16 bits)	-32,768 (-2^{15}) to 32,767 ($-2^{15}-1$)
int	Integer (32 bits)	-2,147,483,648 (-2^{31}) to 2,147,483,647 ($-2^{31}-1$)
long	Long integer (64 bits)	-9,223,372,036,854,775,808 (-2^{63}) to 9,223,372,036,854,775,807 ($2^{63}-1$)
float	Single-precision floating point (32-bits)	Smallest positive nonzero: $14e^{-45}$ Largest positive nonzero: 3.4028234^{e38}
double	Double-precision floating point (64-bits)	Smallest positive nonzero: $4.9e^{-324}$

Table 1.2: Java primitives

Primitive	Description	Range
		Largest positive nonzero: $1.7976931348623157e^{308}$
char	A Unicode character	[See Unicode 6 specification]
boolean	A boolean value	true or false

The first six primitives **(byte, short, int, long, float, double)** represent numbers. Each has a different size. For example, a **byte** can contain any whole number between -128 and 127. To understand how the smallest and largest numbers for an integer were obtained, look at its size in bits. A byte is 8 bits long so there are 2^8 or 256 possible values. The first 128 values are reserved for -128 to -1, then 0 takes one place, leaving 127 positive values. Therefore, the range for a byte is -128 to 127.

If you need a placeholder to store number 1000000, you need an **int**. A **long** is even larger, and you might ask, if a **long** can contain a larger set of numbers than a **byte** and an **int**, why not just use a **long?** It is because a **long** takes 64 bits and therefore consume more memory space than a **byte** or an **int**. Thus, to save space, you want to use a primitive with the smallest possible data size.

The primitives **byte, short, int**, and **long** can only hold integers or whole numbers, for numbers with decimal points you need either a **float** or a **double**. A float is a 32-bit value that conforms to the Institute of Electrical and Electronics Engineer (IEEE) Standard 754. A double is a 64-bit value that conforms to the same standard.

A **char** can contain a single Unicode character, such as 'a', '9' or '&'. The use of Unicode allows **char**s to also contain characters that do not exist in the English alphabet. A **boolean** can contain one of two possible states (**false** or **true**).

Note The reason why not everything in Java is an object is speed. Objects are more expensive to create and operate on than primitives. In programming an operation is said to be expensive if it is resource intensive or consumes a lot of CPU cycles to complete.

Now that you know that there are two types of data in Java (primitives and objects), let's continue by studying how to use primitives. Let's start with variables.

1.4 Variables

Variables are data placeholders. Java is a strongly typed language, therefore every variable must have a declared type. There are two data types in Java:

- reference types. A variable of reference type provides a reference to an object.
- primitive types. A variable of primitive type holds a primitive.

How Java Stores Integer Values

You must have heard that computers work with binary numbers, which are numbers that consists of only zeros and ones. This section provides an overview that may come in useful when you learn mathematical operators.

A byte takes eight bits, meaning there are eight bits allocated to store a byte. The leftmost bit is the sign bit. 0 indicates a positive number, and 1 denotes a negative number. 0000 0000 is the binary representation of 0, 0000 0001 of 1, 0000 0010 of 2, 0000 0011 of 3, and 0111 1111 of 127, which is the largest positive number that a byte can contain.

Now, how do you get the binary representation of a negative number? It's easy. Get the binary representation of its positive equivalent first, and reverse all the bits and add 1. For example, to get the binary representation of -3 you start with 3, which is 0000 0011. Reversing the bits results in

```
1111 1100
```

Adding 1 gives you

```
1111 1101
```

which is -3 in binary.

For **int**s, the rule is the same, i.e. the leftmost bit is the sign bit. The only difference is that an **int** takes 32 bits. To calculate the binary form of -1 in an **int**, we start from 1, which is

```
0000 0000 0000 0000 0000 0000 0000 0001
```

Reversing all the bits results in:

```
1111 1111 1111 1111 1111 1111 1111 1110
```

Adding 1 gives us the number we want (-1).

```
1111 1111 1111 1111 1111 1111 1111 1111
```

In addition to the data type, a Java variable also has a name or an identifier. There are a few ground rules in choosing identifiers.

1. An identifier is an unlimited-length sequence of Java letters and Java digits. An identifier must begin with a Java letter.
2. An identifier must not be a Java keyword (given in Table 1.3), a **boolean** literal, or the **null** literal.
3. It must be unique within its scope. Scopes are discussed in Chapter 3, "Objects and Classes."

Java Letters and Java Digits	Java letters include uppercase and lowercase ASCII Latin letters A to Z (\u0041-\u005a—note that \u denotes a Unicode character) and a to z (\u0061-\u007a), and, for historical reasons, the ASCII underscore (_ or \u005f) and the dollar sign ($, or \u0024). The $ character should be used only in mechanically generated source code or, rarely, to access preexisting names on legacy systems. Java digits include the ASCII digits 0-9 (\u0030-\u0039).

Table 1.3: Java keywords

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

Here are some legal identifiers:

```
salary
x2
_x3
row_count
```

Here are some invalid variables:

```
2x
java+variable
```

2x is invalid because it starts with a number. **java+variable** is invalid because it contains a plus sign.

Also note that names are case-sensitive. **x2** and **X2** are two different identifiers.

You declare a variable by writing the type first, followed by the name plus a semicolon. Here are some examples of variable declarations.

```
byte x;
int rowCount;
char c;
```

In the examples above you declare three variables:

- The variable **x** of type **byte**
- The variable **rowCount** of type **int**
- The variable **c** of type **char**

x, rowCount and **c** are variable names or identifiers.

It is also possible to declare multiple variables having the same type on the same line, separating two variables with a comma. For instance:

```
int a, b;
```

which is the same as

```
int a;
int b;
```

However, writing multiple declarations on the same line is not recommended as it reduces readability.

Finally, it is possible to assign a value to a variable at the same time the variable is declared:

```
byte x = 12;
int rowCount = 1000;
char c = 'x';
```

Naming Convention for Variables	Variable names should be short yet meaningful. They should be in mixed case with a lowercase first letter. Subsequent words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters. For example, here are some examples of variable names that are in compliance with Sun's code conventions: **userName, count, firstTimeLogin**.

1.5 Constants

In Java constants are variables whose values, once assigned, cannot be changed. You declare a constant by using the keyword **final**. By convention, constant names are all in upper case with words separated by underscores.

Here are examples of constants or final variables.

```
final int ROW_COUNT = 50;
final boolean ALLOW_USER_ACCESS = true;
```

1.6 Literals

From time to time you need to assign values to variables in your program, such as number 2 to an **int** or the character 'c' to a **char**. For this, you need to write the value representation in a format that the Java compiler understands. This source code representation of a value is called *literal*. There are three types of literals: literals of primitive types, string literals, and the **null** literal. Only literals of primitive types are discussed in this chapter. The **null** literal is discussed in Chapter 3, "Objects and Classes" and string literals in Chapter 4, "Core Classes."

Literals of primitive types have four subtypes: integer literals, floating-point literals, character literals and boolean literals. Each of these subtypes is explained below.

Integer Literals

Integer literals may be written in decimal (base 10, something we are used to), hexadecimal (base 16) or octal (base 8). For example, one hundred can be expressed as **100**. The following are integer literals in decimal:

```
2
123456
```

As another example, the following code assigns 10 to variable **x** of type **int**.

```
int x = 10;
```

Hexadecimal integers are written by using the prefixes **0x** or **0X**. For example, the hexadecimal number 9E is written as 0X9E or 0x9E. Octal integers are written by prefixing the numbers with 0. For instance, the following is an octal number 567:

```
0567
```

Integer literals are used to assign values to variables of types **byte, short, int**, and **long**. Note, however, you must not assign a value that exceeds the capacity of a variable. For instance, the highest number for a **byte** is 127. Therefore, the following code generates a compile error because 200 is too big for a **byte**.

```
byte b = 200;
```

To assign a value to a **long**, suffix the number with the letter **L** or **l**. L is preferable because it is easily distinguishable from digit 1. A **long** can contain values between -9223372036854775808L and 9223372036854775807L (2^{63}).

Beginners of Java often ask why we need to use the suffix l or L, because even without it, such as in the following, the program still compiles.

```
long a = 123;
```

This is only partly true. An integer literal without a suffix L or l is regarded as an **int**. Therefore, the following will generate a compile error because 9876543210 is larger than the capacity for an **int:**

```
long a = 9876543210;
```

To rectify the problem, add an L or l at the end of the number like this:

```
long a = 9876543210L;
```

Longs, ints, shorts, and bytes can also be expressed in binaries by prefixing the numbers with **0B** or **0b**. For instance:

```
byte twelve = 0B1100; // = 12
```

If an integer literal is too long, readability suffers. For this reason, starting from Java 7 you can use underscores to separate digits in integer literals. For example, these two have the same meaning but the second one is obviously easier to read.

```
int million = 1000000;
int million = 1_000_000;
```

It does not matter where you put the underscores. You can use one every three digits, like the example above, or any number of digits. Here are some more examples:

```
short next = 12_345;
int twelve = 0B_1100;
long multiplier = 12_34_56_78_90_00L;
```

Floating-Point Literals

Numbers such as 0.4, 1.23, $0.5e^{10}$ are floating point numbers. A floating point number has the following parts:

- a whole number part
- a decimal point
- a fractional part
- an optional exponent

Take 1.23 as an example. For this floating point, the whole number part is 1, the fractional part is 23, and there is no optional exponent. In $0.5e^{10}$, 0 is the whole number part, 5 the fractional part, and 10 is the exponent.

In Java, there are two types of floating points:

- **float**. 32 bits in size. The largest positive float is 3.40282347e+38 and the smallest positive finite nonzero float is 1.40239846e-45.
- **double**. 64 bits in size. The largest positive double is 1.79769313486231570e+308 and the smallest positive finite nonzero double is 4.94065645841246544e-324.

In both **float**s and **double**s, a whole number part of 0 is optional. In other words, 0.5 can be written as .5. Also, the exponent can be represented by either e or E.

To express float literals, you use one of the following formats.

```
Digits . [Digits] [ExponentPart] f_or_F
. Digits [ExponentPart] f_or_F
Digits ExponentPart f_or_F
Digits [ExponentPart] f_or_F
```

Note that the part in brackets is optional.

The *f_or_F* part makes a floating point literal a **float**. The absence of this part makes a float literal a **double**. To explicitly express a double literal, you can suffix it with D or d.

To write double literals, use one of these formats.

```
Digits . [Digits] [ExponentPart] [d_or_D]
. Digits [ExponentPart] [d_or_D]
Digits ExponentPart [d_or_D]
Digits [ExponentPart] [d_or_D]
```

In both floats and doubles, *ExponentPart* is defined as follows.

```
ExponentIndicator SignedInteger
```

where *ExponentIndicator* is either **e** or **E** and *SignedInteger* is.

$$Sign_{opt}\ Digits$$

and *Sign* is either **+** or **-** and a plus sign is optional.

Examples of **float** literals include the following:

```
2e1f
8.f
.5f
0f
3.14f
9.0001e+12f
```

Here are examples of **double** literals:

```
2e1
8.
.5
0.0D
3.14
9e-9d
7e123D
```

Boolean Literals

The **boolean** type has two values, represented by literals **true** and **false**. For example, the following code declares a **boolean** variable **includeSign** and assigns it the value of **true**.

```
boolean includeSign = true;
```

Character Literals

A character literal is a Unicode character or an escape sequence enclosed in single quotes. An escape sequence is the representation of a Unicode character that cannot be entered using the keyboard or that has a special function in Java. For example, the carriage return and linefeed characters are used to terminate a line and do not have visual representation. To express a linefeed character, you need to escape it, i.e. write its character representation. Also, single quote characters need to be escaped because single quotes are used to enclosed characters.

Here are some examples of character literals:

```
'a'
'Z'
'0'
'ü'
```

Here are character literals that are escape sequences:

```
'\b'  the backspace character
'\t'  the tab character
'\\'  the backslash
'\''  single quote
```

```
'\"'  double quote
'\n'  linefeed
'\r'  carriage return
```

In addition, Java allows you to escape a Unicode character so that you can express a Unicode character using a sequence of ASCII characters. For example, the Unicode code for the character £ is 00A3. You can write the following character literal to express this character:

```
'£'
```

However, if you do not have the tool to produce that character using your keyboard, you can escape it this way:

```
'\u00A3'
```

1.7 Primitive Conversions

When dealing with different data types, you often need to perform conversions. For example, assigning the value of a variable to another variable involves a conversion. If both variables have the same type, the assignment will always succeed. Conversion from a type to the same type is called identity conversion. For example, the following operation is guaranteed to be successful:

```
int a = 90;
int b = a;
```

However, conversion to a different type is not guaranteed to be successful or even possible. There are two other kinds of primitive conversions, the widening conversion and the narrowing conversion.

The Widening Conversion

The widening primitive conversion occurs from a type to another type whose size is the same or larger than that of the first type, such as from **int** (32 bits) to **long** (64 bits). The widening conversion is permitted in the following cases:

- **byte** to **short, int, long, float**, or **double**
- **short** to **int, long, float**, or **double**
- **char** to **int, long, float**, or **double**
- **int** to **long, float**, or **double**
- **long** to **float** or **double**
- **float** to **double**

A widening conversion from an integer type to another integer type will not risk information loss. At the same token, a conversion from **float** to **double** preserves all the information. However, a conversion from an **int** or a **long** to a **float** may result in loss of precision.

The widening primitive conversion occurs implicitly. You do not need to do anything in your code. For example:

```
int a = 10;
long b = a; // widening conversion
```

The Narrowing Conversion

The narrowing conversion occurs from a type to a different type that has a smaller size, such as from a **long** (64 bits) to an **int** (32 bits). In general, the narrowing primitive conversion can occur in these cases:

- **short** to **byte** or **char**
- **char** to **byte** or **short**
- **int** to **byte, short**, or **char**
- **long** to **byte, short**, or **char**

- **float** to **byte, short, char, int**, or **long**
- **double** to **byte, short, char, int, long**, or **float**

Unlike the widening primitive conversion, the narrowing primitive conversion must be explicit. You need to specify the target type in parentheses. For example, here is a narrowing conversion from **long** to **int**.

```
long a = 10;
int b = (int) a; // narrowing conversion
```

The **(int)** on the second line tells the compiler that a narrowing conversion should occur.

The narrowing conversion may incur information loss, if the converted value is larger than the capacity of the target type. The preceding example did not cause information loss because 10 is small enough for an **int**. However, in the following conversion, there is some information loss because 9876543210L is too big for an **int**.

```
long a = 9876543210L;
int b = (int) a; // the value of b is now 1286608618
```

A narrowing conversion that results in information loss introduces a defect in your program.

1.8 Operators

A computer program is a collection of operations that together achieve a certain function. There are many types of operations, including addition, subtraction, multiplication, division, and bit shifting. In this section you will learn various Java operations.

An operator performs an operation on one, two or three operands. Operands are the targets of an operation and the operator is a symbol representing the action. For example, here is an additive operation:

```
x + 4
```

In this case, **x** and 4 are the operands and + is the operator.

An operator may or may not return a result.

Note Any legal combination of operators and operands are called an expression. For example, **x + 4** is an expression. A boolean expression results in either **true** or **false**. An integer expression produces an integer. And, the result of a floating-point expression is a floating point number.

Operators that require only one operand are called unary operators. There are a few unary operators in Java. Binary operators, the most common type of Java operator, take two operands. There is also one ternary operator, the **? :** operator, that requires three operands.

Table 1.4 list Java operators.

Table 1.4: Java operators

=	>	<	!	~	? :	instanceof				
==	<=	>=	!=	&&	||	++	--			
+	-	*	/	&	|	^	%	<<	>>	>>>
+=	-=	*=	/=	&=	|=	^=	%=	<<=	>>=	>>>=

In Java, there are six categories of operators.

- Unary operators
- Arithmetic operators
- Relational and conditional operators
- Shift and logical operators
- Assignment operators
- Other operators

Each of these operators is discussed in the following sections.

Unary Operators

Unary operators operate on one operand. There are six unary operators, all discussed in this section.

Unary Minus Operator –

The unary minus operator returns the negative of its operand. The operand must be a numeric primitive or a variable of a numeric primitive type. For example, in the following code, the value of **y** is -4.5;

```
float x = 4.5f;
float y = -x;
```

Unary Plus Operator +

This operator returns the value of its operand. The operand must be a numeric primitive or a variable of a numeric primitive type. For example, in the following code, the value of **y** is 4.5.

```
float x = 4.5f;
float y = +x;
```

This operator does not have much significance since its absence makes no difference.

Increment Operator ++

This operator increments the value of its operand by one. The operand must be a variable of a numeric primitive type. The operator can appear before or after the operand. If the operator appears before the operand, it is called the prefix increment operator. If it is written after the operand, it becomes the postfix increment operator.

As an example, here is a prefix increment operator in action:

```
int x = 4;
++x;
```

After **++x**, the value of **x** is 5. The preceding code is the same as

```
int x = 4;
x++;
```

After **x++**, the value of **x** is 5.

However, if the result of an increment operator is assigned to another variable in the same expression, there is a difference between the prefix operator and its postfix twin. Consider this example.

```
int x = 4;
int y = ++x;
// y = 5, x = 5
```

The prefix increment operator is applied *before* the assignment. **x** is incremented to 5, and then its value is copied to **y**.

However, check the use of the postfix increment operator here.

```
int x = 4;
int y = x++;
// y = 4, x = 5
```

With the postfix increment operator, the value of the operand **(x)** is incremented *after* the value of the operand is assigned to another variable **(y)**.

Note that the increment operator is most often applied to **int**s. However, it also works with other types of numeric primitives, such as **float** and **long**.

Decrement Operator --

This operator decrements the value of its operand by one. The operand must be a variable of a numeric primitive type. Like the increment operator, there are also the prefix decrement operator and the postfix decrement operator. For instance, the following code decrements **x** and assigns the value to **y**.

```
int x = 4;
int y = --x;
// x = 3; y = 3
```

In the following example, the postfix decrement operator is used:

```
int x = 4;
int y = x--;
// x = 3; y = 4
```

Logical Complement Operator !

This operator can only be applied to a **boolean** primitive or an instance of **java.lang.Boolean**. The value of this operator is **true** if the operand is **false**, and **false** if the operand is **true**. For example:

```
boolean x = false;
boolean y = !x;
// at this point, y is true and x is false
```

Bitwise Complement Operator ~

The operand of this operator must be an integer primitive or a variable of an integer primitive type. The result is the bitwise complement of the operand. For example:

```
int j = 2;
int k = ~j; // k = -3; j = 2
```

To understand how this operator works, you need to convert the operand to a binary number and reverse all the bits. The binary form of 2 in an integer is:

```
0000 0000 0000 0000 0000 0000 0000 0010
```

Its bitwise complement is

```
1111 1111 1111 1111 1111 1111 1111 1101
```

which is the representation of -3 in an integer.

Arithmetic Operators

There are four types of arithmetic operations: addition, subtraction, multiplication, division, and modulus. Each arithmetic operator is discussed here.

Addition Operator +

The addition operator adds two operands. The types of the operands must be convertible to a numeric primitive. For example:

```
byte x = 3;
int y = x + 5; // y = 8
```

Make sure the variable that accepts the addition result has a big enough capacity. For example, in the following code the value of **k** is -294967296 and not 4 billion.

```
int j = 2000000000; // 2 billion
int k = j + j; // not enough capacity. A bug!!!
```

On the other hand, the following works as expected:

```
long j = 2000000000; // 2 billion
long k = j + j; // the value of k is 4 billion
```

Subtraction Operator –

This operator performs subtraction between two operands. The types of the operands must be convertible to a numeric primitive type. As an example:

```
int x = 2;
int y = x - 1;    // y = 1
```

Multiplication Operator *

This operator perform multiplication between two operands. The type of the operands must be convertible to a numeric primitive type. As an example:

```
int x = 4;
int y = x * 4;      // y = 16
```

Division Operator /

This operator perform division between two operands. The left hand operand is the dividend and the right hand operand the divisor. Both the dividend and the divisor must be of a type convertible to a numeric primitive type. As an example:

```
int x = 4;
int y = x / 2;     // y = 2
```

Note that at runtime a division operation raises an error if the divisor is zero.

The result of a division using the / operator is always an integer. If the divisor does not divide the dividends equally, the remainder will be ignored. For example

```
int x = 4;
int y = x / 3;     // y = 1
```

The **java.lang.Math** class, explained in Chapter 4, "Core Classes," can perform more sophisticated division operations.

Modulus Operator %

The modulus operator perform division between two operands and returns the remainder. The left hand operand is the dividend and the right hand operand the divisor. Both the dividend and the divisor must be of a type that is convertible to a numeric primitive type. For example the result of the following operation is 2.

```
8 % 3
```

Equality Operators

There are two equality operators, **==** (equal to) and **!=** (not equal to), both operating on two operands that can be integers, floating points, characters, or **boolean**. The outcome of equality operators is a **boolean**.

For example, the value of **c** is **true** after the comparison.

```
int a = 5;
int b = 5;
boolean c = a == b;
```

As another example,

```
boolean x = true;
boolean y = true;
boolean z = x != y;
```

The value of **z** is **false** after comparison because **x** is equal to **y**.

Relational Operators

There are five relational operators: **<**, **>**, **<=**, and **>=** and **instanceof**. The first four operators are explained in this section.

instanceof is discussed in Chapter 6, "Inheritance."

The **<**, **>**, **<=**, and **>=** operators operate on two operands whose types must be convertible to a numeric primitive type. Relational operations return a **boolean**.

The **<** operator evaluates if the value of the left-hand operand is less than the value of the right-hand operand. For example, the following operation returns **false:**

```
9 < 6
```

The **>** operator evaluates if the value of the left-hand operand is greater than the value of the right-hand operand. For example, this operation returns **true:**

```
9 > 6
```

The **<=** operator tests if the value of the left-hand operand is less than or equal to the value of the right-hand operand. For example, the following operation evaluates to **false:**

```
9 <= 6
```

The **>=** operator tests if the value of the left-hand operand is greater than or equal to the value of the right-hand operand. For example, this operation returns **true:**

```
9 >= 9
```

Conditional Operators

There are three conditional operators: the AND operator **&&**, the OR operator **||**, and the **? :** operator. Each of these is detailed below.

The && operator

This operator takes two expressions as operands and both expressions must return a value that must be convertible to **boolean**. It returns **true** if both operands evaluate to **true**. Otherwise, it returns **false**. If the left-hand operand evaluates to **false**, the right-hand operand will not be evaluated. For example, the following returns **false**.

```
(5 < 3) && (6 < 9)
```

The || Operator

This operator takes two expressions as operands and both expressions must return a value that must be convertible to **boolean**. || returns **true** if one of the operands evaluates to **true**. If the left-hand operand evaluates to **true**, the right-hand operand will not be evaluated. For instance, the following returns **true**.

```
(5 < 3) || (6 < 9)
```

The ? : Operator

This operator operates on three operands. The syntax is

```
expression1 ? expression2 : expression3
```

Here, *expression1* must return a value convertible to **boolean**. If *expression1* evaluates to **true**, *expression2* is returned. Otherwise, *expression3* is returned.

For example, the following expression returns 4.

```
(8 < 4) ? 2 : 4
```

Shift Operators

A shift operator takes two operands whose type must be convertible to an integer primitive. The left-hand operand is the value

to be shifted, the right-hand operand indicates the shift distance. There are three types of shift operators:

- the left shift operator **<<**
- the right shift operator **>>**
- the unsigned right shift operator **>>>**

The Left Shift Operator <<

The left shift operator bit-shifts a number to the left, padding the right bits with 0. The value of **n << s** is **n** left-shifted **s** bit positions. This is the same as multiplication by two to the power of s.

For example, left-shifting an **int** whose value is 1 with a shift distance of 3 (1 << 3) results in 8. Again, to figure this out, you convert the operand to a binary number.

```
0000 0000 0000 0000 0000 0000 0000 0001
```

Shifting to the left 3 shift units results in:

```
0000 0000 0000 0000 0000 0000 0000 1000
```

which is equivalent to 8 (the same as $1 * 2^3$).

Another rule is this. If the left-hand operand is an **int**, only the first five bits of the shift distance will be used. In other words, the shift distance must be within the range 0 and 31. If you pass an number greater than 31, only the first five bits will be used. This is to say, if **x** is an **int, x << 32** is the same as **x << 0; x << 33** is the same as **x << 1**.

If the left-hand operand is a **long**, only the first six bits of the shift distance will be used. In other words, the shift distance actually used is within the range 0 and 63.

The Right Shift Operator >>

The right shift operator **>>** bit-shifts the left-hand operand to the right. The value of **n >> s** is **n** right-shifted **s** bit positions. The resulting value is $\mathbf{n/2^s}$.

As an example, **16 >> 1** is equal to 8. To prove this, write the binary representation of 16.

```
0000 0000 0000 0000 0000 0000 0001 0000
```

Then, shifting it to the right by 1 bit results in.

```
0000 0000 0000 0000 0000 0000 0000 1000
```

which is equal to 8.

The Unsigned Right Shift Operator >>>

The value of **n >>> s** depends on whether **n** is positive or negative. For a positive **n**, the value is the same as **n >> s**.

If **n** is negative, the value depends on the type of **n**. If **n** is an **int**, the value is **(n>>s) +(2<<~s)**. If **n** is a **long**, the value is **(n>>s)+(2L<<~s)**.

Assignment Operators

There are twelve assignment operators:

```
=  +=  -=  *=  /=  %=  <<=  >>=  >>>=  &=  ^=  |=
```

Assignment operators take two operands whose type must be of an integral primitive. The left-hand operand must be a variable. For instance:

```
int x = 5;
```

Except for the assignment operator **=**, the rest work the same way and you should see each of them as consisting of two operators. For example, **+=** is actually **+** and **=**. The assignment operator **<<=** has two operators, **<<** and **=**.

The two-part assignment operators work by applying the first operator to both operands and then assign the result to the left-hand operand. For example **x += 5** is the same as **x = x + 5**.

x -= 5 is the same as **x = x - 5**.

x <<= 5 is equivalent to **x = x << 5**.

x &= 5 produces the same result as **x = x &= 5**.

Integer Bitwise Operators & | ^

The bitwise operators **& | ^** perform a bit to bit operation on two operands whose types must be convertible to **int. &** indicates an AND operation, **|** an OR operation, and **^** an exclusive OR operation. For example,

```
0xFFFF & 0x0000 = 0x0000
0xF0F0 & 0xFFFF = 0xF0F0
0xFFFF | 0x000F = 0xFFFF
0xFFF0 ^ 0x00FF = 0xFF0F
```

Logical Operators & |^

The logical operators **& | ^** perform a logical operation on two operands that are convertible to **boolean. &** indicates an AND operation, **|** an OR operation, and ^ an exclusive OR operation. For example,

```
true & true  = true
true & false = false
true | false = true
false | false = false
true ^ true = false
false ^ false = false
false ^ true = true
```

Operator Precedence

In most programs, multiple operators often appear in an expression, such as.

```
int a = 1;
int b = 2;
int c = 3;
int d = a + b * c;
```

What is the value of **d** after the code is executed? If you say 9, you're wrong. It's actually 7.

Multiplication operator ***** takes precedence over addition operator **+**. As a result, multiplication will be performed before addition. However, if you want the addition to be executed first, you can use parentheses.

```
int d = (a + b) * c;
```

The latter will assign 9 to **d**.

Table 1.5 lists all the operators in the order of precedence. Operators in the same column have equal precedence.

Table 1.5: Operator precedence

Operator	
postfix operators	[ ] . (params) expr++ expr--
unary operators	++expr --expr +expr -expr ~ !
creation or cast	new (type)expr
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof

Table 1.5: Operator precedence

Operator	
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
conditional	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Note that parentheses have the highest precedence. Parentheses can also make expressions clearer. For example, consider the following code:

```
int x = 5;
int y = 5;
boolean z = x * 5 == y + 20;
```

The value of **z** after comparison is **true**. However, the expression is far from clear.

You can rewrite the last line using parentheses.

```
boolean z = (x * 5) == (y + 20);
```

which does not change the result because * and **+** have higher precedence than **==**, but this makes the expression much clearer.

Promotion

Some unary operators (such as **+**, **-**, and **~**) and binary operators (such as +, -, *, /) cause automatic promotion, i.e. elevation to a wider type such as from **byte** to **int**. Consider the following code:

```
byte x = 5;
byte y = -x; // error
```

The second line surprisingly causes an error even though a byte can accommodate -5. The reason for this is the unary operator - causes the result of **-x** to be promoted to **int**. To rectify the problem, either change **y** to **int** or perform an explicit narrowing conversion like this.

```
byte x = 5;
byte y = (byte) -x;
```

For unary operators, if the type of the operand is **byte, short**, or **char**, the outcome is promoted to **int**.

For binary operators, the promotion rules are as follows.

- If any of the operands is of type **byte** or **short**, then both operands will be converted to **int** and the outcome will be an **int**.
- If any of the operands is of type **double**, then the other operand is converted to **double** and the outcome will be a **double**.
- If any of the operands is of type **float**, then the other operand is converted to **float** and the outcome will be a **float**.
- If any of the operands is of type **long**, then the other operand is converted to **long** and the outcome will be a **long**.

For example, the following code causes a compile error:

```
short x = 200;
short y = 400;
short z = x + y;
```

You can fix this by changing **z** to **int** or perform an explicit narrowing conversion of **x + y**, such as

```
short z = (short) (x + y);
```

Note that the parentheses around **x + y** is required, otherwise only **x** would be converted to **int** and the result of addition of a **short** and an **int** will be an **int**.

1.9 Comments

It is good practice to write comments throughout your code, sufficiently explaining what functionality a class provides, what a method does, what a field contains, and so forth.

There are two types of comments in Java, both with syntax similar to comments in C and C++.

- Traditional comments. Enclose a traditional comment in /* and */.
- End-of-line comments. Use double slashes (//) which causes the rest of the line after // to be ignored by the compiler.

For example, here is a comment that describes a method

```
/*
  toUpperCase capitalizes the characters of in a String object
*/
public void toUpperCase(String s) {
```

Here is an end-of-line comment:

```
public int rowCount; //the number of rows from the database
```

Traditional comments do not nest, which means

```
/*
  /* comment 1 */
  comment 2 */
```

is invalid because the first ***/** after the first **/*** will terminate the comment. As such, the comment above will have the extra **comment 2 */**, which will generate a compiler error.

On the other hand, end-of-line comments can contain anything, including the sequences of characters **/*** and ***/**, such as this:

```
// /* this comment is okay */
```

Self Test

1 Question ?

Which of the following are Java primitives? (Choose all that apply)

A. int

B. Long

C. short

D. boolean

E. String

2 Question ?

Which of the following are Java legal identifiers? (Choose all that apply)

A. tempCounter

B. long

C. $

D. _rows

E. game_users

3 Question ?

Which of the following are valid variable declarations? (Choose all that apply)

A. int int;

B. long Long;

C. boolean $true;

D. short #x2;

E. int a, b, c;

4 Question ?

Which of the following are valid value assignments? (Choose all that apply)

A. float f = 123;

B. float g = 123.45;

C. double h = 123.45;

D. long i = 123;

E. int j = 12345_12345_12345;

F. long k = 12345_12345_12345;

5 Question ?

Which of the following are valid value assignments? (Choose all that apply)

A. float f = 123;

B. float g = 123.45F;

C. double h = 123.45D;

D. long i = 123L;

E. int j = 12345_12345_12345;

F. long k = 12345_12345_12345L;

6 Question ?

Given

```
int temp = 1 + 2 * 3;
```

What is the value of temp?

A. 9

B. 7

C. 2

D. None of the above

7 Question ?

Given

```
int temp = (1 + 2) * 3;
```

What is the value of temp?

A. 9

B. 7

C. 2

D. 6

8 Question ?

Which of these statements create a variable whose value cannot be changed?

A. public static final int TEMP = 5;

B. public static int TEMP = 5;

C. private static final int TEMP = 5;

D. final int TEMP = 5;

9 Question ?

Given the following code

```
1.  long a = 1000_000_000L;
2.  int b = a - 999_999_999L;
3.  int c = (int) a - 999_999_999L;
4.  int d = (int) (a - 999_999_999L);
```

What will happen if you try to compile the code?

A. The code will compile with no reported errors;

B. Compile error at line 2;

C. Compile error at line 3;

D. Compile error at line 4;

10 Question ?

Which of the following are valid comments?

A. // This is a comment

B. /* This is a comment */

C. /* This is a /* good */ comment */

D. // /* This is a /* good */ comment */

Answers

1 **A, C, D**.

int, short and **boolean** are some of Java primitives, and so is **long**, but not **Long.String** is a Java class and is not a primitive.

2 **A, C, D, E**.

B is not a valid identifier because **long** is a Java primitive. Note that **$** can be used in a Java identifier but its use is normally reserved for machine-generated code.

3 **B, C, E**.

A is invalid because **int** is a primitive and cannot be used as an identifier. B is valid because **Long** can be used as a variable name, even though it happens to be the name of a popular Java class. C is legal because $ can be used in an identifier. D is invalid because it contains the character #. E is also valid and declares three **int**s.

4 **A, C, D**.

B is invalid because 123.45 is a **double** and cannot be assigned to a float. E is invalid because the value is too big for an **int**. F is invalid because literal 12345_12345_12345 is considered an **int** and its value is too big for an **int**.

5 **A, B, C, D, F**.

E is invalid because the value is too big for an int. All the other options are correct.

6 **B**.

2 * 3 is evaluated first because * has precedence over +. The result (6) is then added to 1.

7 **A**.

The bracket has precedence over *, so 1 + 2 is evaluated first. The result (3) is then multiplied by 3.

8 **A, C, D**.

The **final** keyword makes a variable a constant.

9 **B, C**.

The expression at line 2 returns a **long** and assigning a **long** to an **int** results in a compile error. At line 3 variable a is converted to an int, which is legal. However, subtracting a **long** from an **int** results in a **long**, so the expression at line 3 returns a **long**. Attempting to assign a **long** value to an **int** causes a compile error. Line 4 performs the subtraction first and then the result is converted to an **int** and assigned to an **int** variable. Therefore, line 4 does not cause a compile error.

10 **A, B, D**.

C is invalid because the comment is nested. D is valid because the statement comments out everything to the right of //.