



# Java Operators with Primitives and Objects

---

## Terms you'll need to understand:

- ✓ Assignment
- ✓ **instanceof**
- ✓ **equals**

## Techniques you'll need to master:

- ✓ Constructing numeric literals in base ten, hexadecimal, and octal formats
- ✓ Constructing character literals in Java's Unicode format
- ✓ Constructing string literals in the quoted format
- ✓ Understanding the effect of assignment and mathematical operators on primitives and objects
- ✓ Understanding the operation of bitwise and logical operators in expressions
- ✓ Understanding the implications of the various forms of the **AND** and **OR** logical operators
- ✓ Understanding the correct use of the **==** comparison operator with primitives and objects
- ✓ Predicting the operation of the **equals** method with combinations of various objects
- ✓ Declaring, constructing, and initializing arrays of any type

# Introduction

Java uses literals and operators in a style that will be very familiar to all C programmers. In this chapter, we review the way Java uses literals to initialize primitive variables, create objects, and pass values to methods. We then review all of the Java operators used in expressions with both primitives and objects. You should not assume that the behavior of operators is the same in Java as in C. Pay particular attention to the difference between the `==` operator (double equals sign) and the `equals` method; this seems to confuse many programmers.

## Using Literals

Literals are used to create values that are assigned to variables, used in expressions, or passed to methods. You need to know the correct ways of creating integer, floating-point, character, string, and boolean literals.

## Numeric Literals

Literal numbers can appear in Java programs in base ten, hexadecimal, and octal forms, as shown in the following sample code statements, which combine declaring a variable and initializing it to a value:

```
1. int n = 42 ;
2. long j = 4096L ; // appending L or l makes it a long
3. long k = 0xFFFFFL ;
4. byte b2 = 010 ; // an octal literal
5. double f2 = 1.023 ; // double is assumed
6. float f2 = 1.023F ; // F or f makes it a float
```

Notice that an unmodified integer value is assumed to be the 32-bit `int` primitive, but a value containing a decimal point is assumed to be the 64-bit `double`, unless you append an `F` or `f` to indicate the 32-bit `float` primitive.

In line 1, an unmodified literal integer is assumed to be in base ten format. In line 3, a number starting with a leading zero followed by an uppercase or lowercase `X` is interpreted as a hexadecimal number. In line 4, a number with a leading zero and no `X` is interpreted as an octal number. Appending an uppercase or lowercase `L` indicates a long integer.

## Tricky Literal Assignment Facts

The compiler does a variety of automatic conversions of numeric types in expressions, but in assignment statements, it gets quite picky as a defense

against common programmer errors. In the following code, lines 1 and 3 cause a compiler "possible loss of precision" error:

```
1. int n2 = 4096L ; // would require a specific (int) cast
2. short s1 = 32000 ; // ok
3. short s2 = 33000 ; // out of range for short primitive
4. int secPerDay = 24 * 60 * 60 ;
```

Although 4096 would fit in an `int` primitive, the compiler would object to line 1 because the literal is in the long format. It would require a special operator called a *cast* to allow the statement. A Java cast operator takes the form of a type enclosed in parentheses. It is an instruction to the compiler to allow conversion of one variable type to another. Casts are discussed in detail in Chapter 6, "Converting and Casting Primitives and Objects."

The compiler also pays attention to the known range of primitives, passing line 2 in the previous example but objecting to line 3. You could force the compiler to accept line 3 with a specific (`short`) cast, but the result would be a negative number due to the high bit being set.

In line 4, the compiler pre-computes the resulting value rather than writing code to perform the multiplication. This handy feature lets you write out the factors of a useful number such as `secPerDay` without any runtime penalty in memory used.

## Numeric Wrapper Classes

Each of the primitive data types has a corresponding wrapper class in the Java standard library. Java uses these classes for several purposes. The static variables of a wrapper class hold various constants, and the static methods are used for convenient conversion routines, such as the `toString` method, which returns a `String` representing a primitive value. You can also create objects that contain a primitive value, using either literals or primitive variables, as in the following examples:

```
1. Integer cmd = new Integer( 42 ) ;
2. Boolean flag = new Boolean( false ) ;
3. Character pi = new Character( '\u03c0' ) ;
4. Long lx = new Long( x ) ; // where x is a long variable
```

The values contained in a wrapper object cannot be changed, so they are not used for computation. Wrapper objects are useful when you want to store primitive values using Java's utility classes, such as `Vector`, `Stack`, and `Hashtable`, that work only with objects. The names of the wrapper classes are `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`, and `Boolean`. As you can see, the names reflect the primitive values they contain but start with a capital letter. These wrapper classes are discussed more extensively in Chapter 11, "Standard Library Utility Classes."



The most important point to remember about the wrapper class objects is that the contained value cannot be changed. These objects are said to be *immutable*.

## Character Literals

Even though typical Java code looks as if it is made up of nothing but ASCII characters, you should never forget that Java characters are, in fact, 16-bit Unicode characters. The following code shows legal ways to declare and initialize char type primitive variables:

```
1. char c1 = '\u0057' ; // the letter W as Unicode
2. char c2 = 'W' ;
3. char c3 = (char) 87 ; // the letter W
4. char cr = '\r' ; // carriage return
```

Line 1 illustrates the Unicode representation of a character indicated by the leading `\u` sequence. The numeric value is always a four-digit hexadecimal number in this format. Line 2 uses single quotation marks surrounding the literal character, and an integer is cast to the char primitive type in line 3. Line 4 shows a literal representing a nonprinting character with an escape sequence.

You can also mix in Unicode characters and special characters with ASCII strings using the `\u` escape sequence (as shown in the “String Literals” section later in this chapter). Table 3.1 summarizes Java escape sequences that can be used in strings or to initialize character primitives.

**Table 3.1 Java Escape Sequences**

Escape Sequence	Character Represented
<code>\b</code>	Backspace.
<code>\t</code>	Horizontal tab.
<code>\n</code>	New line (line feed).
<code>\f</code>	Form feed.
<code>\r</code>	Carriage return.
<code>\"</code>	Double quotation mark.
<code>\'</code>	Single quotation mark.
<code>\\</code>	Backslash.
<code>\xxx</code>	A character in octal representation; <b>xxx</b> must range from 000 through 377.
<code>\uxxxx</code>	A Unicode character, where <b>xxxx</b> is a hexadecimal-format number in the range 0000 through FFFF. Note that this is the only escape sequence that represents a 16-bit character.

## Special Precautions to Take with Unicode

Java translates Unicode characters as it reads the text, so you can't insert the code for carriage return or line feed characters using Unicode in program source code. The compiler sees the carriage return or line feed as an end-of-line character and reports an error. That is why you must use the escape sequences shown in Table 3.1 to insert these characters in string literals.

## String Literals

Because Java does not deal with strings as arrays of bytes, but as objects, the compiler has to do a lot of work behind the scenes. The following sample code shows the declaration and initialization of `String` variables with literal values enclosed in double quotation marks. Note that unlike some languages, Java does not allow single quotation marks as an alternative for delineating strings. Single quotes are only used for character literal values.

```
1. String name = "" ; // an empty string, but still an object
2. String type = ".TXT" ;
3. String longtxt = "A great long bunch of text \n"
4.           + "to illustrate how you break long lines." ;
```

At the end of line 3, notice the sequence `\n`; this is an example of an escape code sequence used to insert special characters—in this case, a line feed. Line 4 illustrates the special meaning of the `+` operator when used with strings; this is discussed in “String Objects and the `+` Operator” section later in this chapter. Here is an example of a `String` literal with the Unicode representation of a capital Greek letter delta inserted between two double quotation mark characters:

```
String tx = "Delta values are labeled \"\u0394\" on the chart.";
```



Remember that string literals create **String** objects, not byte arrays. Most of the things you are used to doing with strings in C will not work in Java.

## Boolean Literals

Fortunately, boolean literals are simple. Only the Java reserved words `true` and `false` can be used. Note that these words are always all lowercase. If you write

```
boolean flag = True ;
```

the compiler goes looking for a boolean variable named `True`.



C programmers should remember that integer variables can never be interpreted as boolean values. You must leave behind all of your tricks that depend on zero being interpreted as **false**.

## Numeric Operators

In the last section, we created and initialized some variables. Now let's look at Java's facilities for numeric operations. They will look very familiar to C programmers, but there are some differences. Operators that perform arithmetic or numeric comparison are shown in Table 3.2. The precedence gives the order in which the compiler performs operations, with 1 being the first. You can always use parentheses to control the order in which operations are performed.

**Table 3.2** Numeric Operators in Java

Precedence	Operator	Description
1	<b>++</b>	Increment by 1 (or 1.0)
1	<b>--</b>	Decrement by 1 (or 1.0)
1	<b>+</b>	Unary plus
1	<b>-</b>	Unary minus
2	<b>*</b>	Multiplication
2	<b>/</b>	Division
2	<b>%</b>	Modulo
3	<b>+</b>	Addition
3	<b>-</b>	Subtraction
5	<b>&lt;</b>	Less than test
5	<b>&gt;</b>	Greater than test
5	<b>&lt;=</b>	Less than/equal test
5	<b>&gt;=</b>	Greater than/equal test
6	<b>==</b>	Equals test (identical values)
6	<b>!=</b>	Not equals to test
13	<b>op=</b>	<b>op</b> with assignment ( <b>+=</b> , <b>-=</b> , <b>*=</b> , and so on)

## Order of Evaluation of Operands

When evaluating an expression, Java always evaluates the operand on the left first. This rule can be important if the left operand is a method call or an expression that modifies a variable that appears on the right.

## Increment and Decrement

Java follows the C convention with the increment and decrement operators, which directly modify the value in a primitive variable by adding or subtracting 1. When this operator appears in a larger expression, the order in which the modification occurs depends on the position of the operator, as shown in the following code fragment:

```
1. int x = 5 ;
2. int y = x++ ; // y gets the value 5, before incrementing x
3. int y2 = ++x ; // y2 gets the value 7, after incrementing
```

When evaluating expressions that involve increment and decrement, keep in mind that expression evaluation is always “left first.” For example, consider the following sequence:

```
1. int a = 2 ;
2. a += ++a ;
3. System.out.println( "value of a= " + a );
```

This code prints `value of a= 5` because the Java first evaluates the left side of `+=` as 2, and then evaluates `++a` as 3, and finally carries out the addition and stores the result in `a`, replacing the value created by `++a`. Remember that `++` or `--` before the variable indicates “pre” evaluation of the variable and when the operator is after the variable, it indicates “post” evaluation.



It would not be at all unusual for you to have one or more questions in which the order of increment or decrement operations is critical.

## Unary + and - Operators

Distinct from the arithmetic add and subtract operators, the unary `+` and `-` operators affect a single operand. Unary `-` changes the sign of a numeric expression to the right of the operator. Unary `+` has no effect on an expression; it is included for completeness and because some programmers like to use it to emphasize that a number is positive.

## Arithmetic Operators

In general, the arithmetic operators `+`, `-`, `/`, and `*` work as you would expect, but you will need to know the conventions that the compiler uses to convert various primitives before performing operations. As with `C`, the operator appears between its two operands.

### Arithmetic Operators with Assignment

The operators that combine an arithmetic operator with the `=` assignment operator perform an operation on the contents of the variable on the left side and store the results in the variable. For example, in the following code, line 2 is equivalent to line 3:

```
1. int x = 5 ;
2. x += 10 ; // x gets 5 + 10
3. x = x + 10 ;
```

The compiler makes some assumptions when it sees an operator with assignment. For instance, in the following sequence of statements, the compiler does not object to the fact that line 2 adds an `int` value to a `byte` because it performs an explicit cast, the equivalent of line 4; however, in line 3, which is the logical equivalent of line 2, it raises an objection:

```
1. byte b = 0 ;
2. b += 27 ;
3. b = b + 27 ;
4. b = (byte)(b + 27) ;
```

### Widening Conversions

Widening conversions of a number are those that don't lose information on the overall magnitude. For instance, the integer primitives `byte`, `char`, and `short` can all be converted to an `int` primitive, and an `int` primitive can be converted to a `long` integer without loss of information. You may see this sort of widening conversion referred to as *numeric promotion*.

An `int` can be converted to a `float` primitive, but there may be some loss of precision in the least significant bits. This conversion is carried out according to the Institute of Electrical and Electronics Engineers (IEEE) standard.

When evaluating an arithmetic expression with two operands, the compiler converts primitives by widening according to these rules:

1. If either is of type `double`, the other is converted to `double`.
2. Otherwise, if either is a `float`, the other is converted to `float`.
3. Otherwise, if either is of type `long`, the other is converted to `long`.
4. Otherwise, both operands are converted to `int`.

These automatic conversions can have significant consequences, particularly when you are trying to store the results of an expression in a primitive variable that has a smaller capacity than one of the operands. Consider the following code:

```
1. int a = 2 ;
2. float x = 1.5f ;
3. a = x * a ;
```

By rule 2, both sides of the expression in line 3 are converted to `float`. However, the compiler knows that `float` variables have a much wider range of magnitude than `int` variables. Therefore, if you try to compile the code, you get an error message. To avoid this error, you have to use a cast.

## Conversion with Casting

You can always direct the order and direction of number conversions with specific casts. As an example, consider the following code fragment:

```
1. float x = 123 ;
2. byte b = 23 ;
3. float y = x + b ;
4. b = (byte) y ;
```

In line 3, the compiler converts `b` to a `float` before performing the addition. You have to include the specific cast operation to get the compiler to accept line 4 because converting a `float` to an 8-bit byte involves potential loss of magnitude and precision.

## The Modulo Operator

You can think of the `%` (modulo) operator as yielding the remainder from an implied division of the left operand (dividend) by the right operand (divisor). The result is negative only when the dividend is negative. Note that if the operands are integers, the `ArithmeticException` can be thrown if the divisor is zero, just as in integer division.

Using `%` with floating-point primitives produces results similar to the integer operation, but note that the special floating-point values, such as `NaN` and `POSITIVE_INFINITY`, can result.

## Numeric Comparisons

The numeric comparisons `<`, `>`, `<=`, `>=`, `!=`, and `==` work pretty much as expected with Java primitives. If the operands are of two different types, the compiler promotes one or both according to the rules for arithmetic

operators. Remember that the result of a numeric comparison is a boolean primitive.

The `<`, `>`, `<=`, and `>=` operators are meaningless for objects, but the `==` and `!=` operators can be used. When used with object references, `==` results in `true` only if the references are identical. We return to this subject later in this chapter in the “Testing Object Equality” section because it is very important.



Be sure you master the differences between the `==` comparison with primitives and with objects. In our experience, this difference has been one of the most frequent sources of errors (on the exam and in programming).

## Arithmetic Errors

In general, Java lets you make a variety of arithmetic errors without warning you. If your code conducts operations that overflow the bounds of 32-bit or 64-bit integer arithmetic, that is your problem. Division by zero in integer arithmetic is the only error that produces a runtime exception, namely, an `ArithmeticException`.

On the other hand, floating-point operations meet the requirements of the IEEE standard for representing values that are out of the normal range. These special values are defined for `float` primitives as constants in the `Float` class, as shown in Table 3.3. The string representation is what you get from the `Float.toString` method. The `Double` class defines similar constants for double primitive values.

**Table 3.3 Special Floating-Point Values**

Constant	Interpretation	Corresponding String
<code>Float.MAX_VALUE</code>	The largest number representable	<code>3.4028235E38</code>
<code>Float.MIN_VALUE</code>	The smallest number representable	<code>1.4E-45</code>
<code>Float.NEGATIVE_INFINITY</code>	Negative divided by zero	<code>-Infinity</code>
<code>Float.POSITIVE_INFINITY</code>	Positive divided by zero	<code>Infinity</code>
<code>Float.NaN</code>	Not a number	<code>NaN</code>

### Not a Number

The special `NaN` value is particularly tricky to handle. `NaN` can result from mathematical functions that are undefined, such as taking the square root of a negative number.

You cannot directly compare the NaN value with anything. You must detect it with the special `Float.isNaN` or `Double.isNaN` methods, as in the following example:

```
1. float x = (float) Math.sqrt( y ) ; // where y may be neg
2. if( x == Float.NaN ) x = 0.0 ; // WRONG, always false
3. if( Float.isNaN( x ) ) x = 0.0 ;
    // the right way to detect NaN
```

This example shows the right way (line 3) and one of the many wrong ways (line 2) to detect the NaN value.

## Floating-Point Math and `strictfp`

The `strictfp` modifier is related to the way floating-point calculations are carried out, as affected by specialized math coprocessors. Recall that `float` and `double` primitives use 32 and 64 bits, respectively, to store values. However, some floating-point coprocessors can use internal representations of numbers that use more bits for the intermediate results of calculation. These processors produce results that are more accurate but differ slightly from what you would get if every intermediate calculation result were forced back to a 32- or 64-bit representation.

Normally, you would want to use the most accurate results possible, but this means that a calculation on one Java Virtual Machine (JVM) could produce a result that is slightly different from the same calculation on another JVM. Of course, this is contrary to the spirit of Java. Starting in Java 1.2, the `strictfp` modifier has been available so you can force floating-point math to reduce all intermediate results to the standard 32- or 64-bit representation, ensuring that calculations produce the same results on all JVMs.

When used as a method modifier, `strictfp` ensures that all calculations in the method follow the strict calculation rules. When used as a class modifier, `strictfp` forces all methods in a class to follow strict calculation rules.

## String Objects and the `+` Operator

For convenience in working with strings, Java also uses the `+` and `+=` operators to indicate concatenation. When the compiler finds an expression in which a string appears in association with a `+` operator, it turns all items in the expression into strings, concatenates the strings, and creates a new `String` object. However, remember that expressions are evaluated left to right, so if the compiler sees a numeric operation before the `String`, it will carry out that operation before the conversion to a `String`. You can see this in action in the following code, which produces "101 is the result", not "1001 is the result".

```
int n = 1 ;
System.out.println( 100 + n + " is the result " );
```

The compiler has a complete set of conventions used to turn primitives and objects into strings, but the only operators that can be used are the `+` and `+=` operators.

The methods used to turn primitives into strings are found in the wrapper classes that Java has for each primitive. (Wrapper classes are discussed in detail in Chapter 11, “Standard Library Utility Classes.”) For instance, in the following code fragment, the compiler knows to use the `toString` method in the `Float` class to create a `String` representation of the `pi` primitive. It also knows how to add the Unicode character for the Greek letter `pi` to the `String`:

```
1. float pi = 3.14159f ;
2. String tmp = "Pi = " + pi + " or " + '\u03c0' ;
```

## Objects and toString()

The root of the Java object hierarchy, the `Object` class, has a `toString()` method that returns a descriptive string. Therefore, every object has a `toString` method by inheritance. This default `toString` method produces a rather cryptic result, so many of the standard library classes implement a `toString` method that is more appropriate for the particular class. The net result is that the compiler can always use the `+` operator in any combination of strings and objects.

## Strings Are Immutable

The contents of a `String` object cannot be changed. Take the following code:

```
1. String filename = new String( "mystuff" ) ;
2. filename += ".txt" ;
```

It looks as if we are changing a `String` object. What is actually happening is that there is a `String` object created in line 1 with a reference in the variable named `filename`. In line 2, the contents of that `String` are concatenated with the literal `".txt"` and a reference to the new `String` object is stored in the variable `filename`.



Questions involving the immutability of `String` objects frequently cause trouble for beginning Java programmers. Chapter 11 contains more examples and practice questions on this subject.

## The null Value and Strings

The Java mechanism that adds various items to create strings can recognize that a reference variable contains the special value `null`, instead of an object reference. In that case, the string `"null"` is added.

## Bitwise and Logical Operators

Table 3.4 summarizes the operators that can be used on individual bits in integer primitives and in logical expressions. Bitwise operators are used in expressions with integer values and apply an operation separately to each bit in an integer. The term *logical expression* refers to an expression in which all of the operands can be reduced to boolean primitives. Logical operators produce a boolean primitive result.

Precedence	Operator	Operator Type	Description
1	<code>~</code>	Integral	Unary bitwise complement
1	<code>!</code>	Logical	Unary logical complement
4	<code>&lt;&lt;</code>	Integral	Left shift
4	<code>&gt;&gt;</code>	Integral	Right shift (keep sign)
4	<code>&gt;&gt;&gt;</code>	Integral	Right shift (zero fill)
5	<code>instanceof</code>	Object, type	Tests class membership
6	<code>==</code>	Object	Equals (same object)
6	<code>!=</code>	Object	Unequal (different object)
7	<code>&amp;</code>	Integral	Bitwise <b>AND</b>
7	<code>&amp;</code>	Logical	Logical <b>AND</b>
8	<code>^</code>	Integral	Bitwise <b>XOR</b>
8	<code>^</code>	Logical	Logical <b>XOR</b>
9	<code> </code>	Integral	Bitwise <b>OR</b>
9	<code> </code>	Logical	Logical <b>OR</b>
10	<code>&amp;&amp;</code>	Logical	Logical <b>AND</b> (conditional)

(continued)

**Table 3.4 Bitwise and Logical Operators (continued)**

Precedence	Operator	Operator Type	Description
11		Logical	Logical <b>OR</b> (conditional)
12	?:	Logical	Conditional (ternary)
13	=	Variable, any	Assignment
13	<<=	Binary	Left shift with assignment
13	>>=	Binary	Right shift with assignment
13	>>>=	Binary	Right shift, zero fill, assignment
13	&=	Binary	Bitwise <b>AND</b> with assignment
13	&=	Logical	Logical <b>AND</b> with assignment
13	=	Binary	Bitwise <b>OR</b> with assignment
13	=	Logical	Logical <b>OR</b> with assignment
13	^=	Binary	Bitwise <b>XOR</b> with assignment
13	^=	Logical	Logical <b>XOR</b> with assignment

## Bitwise Operations with Integers

Bitwise operators change the individual bits of an integer primitive according to the familiar rules for AND, OR, and XOR (Exclusive OR) operations (as summarized in Table 3.5). The operands of the &, |, and ^ operators are promoted to int or long types, as discussed earlier under “Widening Conversions,” and the result is an int or long primitive, not a boolean. Because each bit in an integer primitive can be modified and examined independently with these operators, they are frequently used to pack a lot of information into a small space.

**Table 3.5 Bitwise Logic Rules**

Operand	Operator	Operand	Result
1	& (AND)	1	1
1	& (AND)	0	0
0	& (AND)	1	0
0	& (AND)	0	0
1	(OR)	1	1
1	(OR)	0	1
0	(OR)	1	1
0	(OR)	0	0
1	^ (XOR)	1	0

(continued)

**Table 3.5 Bitwise Logic Rules (continued)**

Operand	Operator	Operand	Result
1	^ (XOR)	0	1
0	^ (XOR)	1	1
0	^ (XOR)	0	0

In thinking about the action of bitwise operators, you may want to draw out the bit pattern for various values. To keep them straight, it helps to draw groups of four bits so the groups correspond to hexadecimal digits. Questions on the test will not require you to remember all of the powers of two, but being able to recognize the first few helps. Here is an example of the use of the & or AND operator:

```
short flags = 20 ; // 0000 0000 0001 0100 or 0x0014
short mask = 4 ; // 0000 0000 0000 0100
short rs1t = (short)( flags & mask ) ;
```

Note that because operands are promoted to `int` or `long`, the cast to `short` is necessary to assign the value to a `short` primitive variable `rs1t`. Applying the rule for the AND operator, you can see that the bit pattern in `rs1t` will be "0000 0000 0000 0100", or a value of 4. C programmers who are used to checking the result of a bitwise operation in an `if` statement, such as line 1 in the following code, should remember that Java can use boolean values in logic statements only, as shown in line 2:

```
1. if( rs1t ) doSomething() ; // ok in C, wrong in Java
2. if( rs1t != 0 ) doSomething() ; // this is ok in both
```

## Practicing Bitwise Operations

Unless you are very familiar with bitwise operations and binary representation of integers, we think you should get in some practice. Listing 3.1 shows a simple practice program. Type it in, compile it, and run it with some example numbers. This is important; do it now!

**Listing 3.1 A Program to Experiment with Bitwise Operators**

```
public class BitwiseTest {
    public static void main(String[] args){
        if( args.length < 2 ){
            System.out.println("expects two numbers");
            System.exit(1);
        }
        int a = Integer.parseInt( args[0] );
        int b = Integer.parseInt( args[1] );
```

(continued)

**Listing 3.1 A Program to Experiment with Bitwise Operators (continued)**

```

System.out.println( "a as binary " +
                    Integer.toBinaryString( a ));
System.out.println( "b as binary " +
                    Integer.toBinaryString( b ));
System.out.println( "NOT a " +
                    Integer.toBinaryString( ~a ));
System.out.println( "NOT b " +
                    Integer.toBinaryString( ~b ));
System.out.println( "a AND b " +
                    Integer.toBinaryString( a & b ));
System.out.println( "a OR b " +
                    Integer.toBinaryString( a | b ));
System.out.println( "a XOR b " +
                    Integer.toBinaryString( a ^ b ));
}
}

```

Wasn't that fun? As a variation on the program in Listing 3.1, you might try using the `toHexString` and `toOctalString` methods in the `Integer` class to show what base 10 numbers look like in hexadecimal (base 16) and octal (base 8) formats.

Table 3.6 shows the result of applying the various bitwise operators to the sample operands `op1` and `op2`. Note that in the last line of the table, the `~`, or complement, operator sets the highest order bit, which causes the integer to be interpreted as a negative number. We are using short primitives here to simplify the table, but the same principles apply to all of the integer primitives.

**Table 3.6 Illustrating Bitwise Operations on Some Short Primitives (16-bit Integers)**

Binary	Operation	Decimal	Hex
0000 0000 0101 0100	<code>op1</code>	84	0x0054
0000 0001 0100 0111	<code>op2</code>	327	0x0147
0000 0000 0100 0100	<code>op1 &amp; op2</code>	68	0x0044
0000 0001 0101 0111	<code>op1   op2</code>	343	0x0157
0000 0001 0001 0011	<code>op1 ^ op2</code>	275	0x0113
1111 1110 1011 1000	<code>~op2</code>	-238	0xFE88

## The Unary Complement Operators

The `~` operator takes an integer type primitive. If smaller than `int`, the primitive value will be converted to an `int`. The result simply switches the sense of every bit. The `!` operator is used with boolean primitives and changes `false` to `true` or `true` to `false`.

## The Shift Operators: <<, >>, and >>>

The shift operators work with integer primitives only; they shift the left operand by the number of bits specified by the right operand. The important point to note with these operators is the value of the new bit that is shifted into the number. For << (left shift), the new bit that appears in the low order position is always zero.

Sun had to define two types of right shift because the high order bit in integer primitives indicates the sign. The >> right shift propagates the existing sign bit, which means a negative number will still be negative after being shifted. The >>> right shift inserts a zero as the most significant bit. Table 3.7 should help you visualize what is going on. Again, if you are not comfortable with these bit manipulations, stop and write some test programs. You can modify the program from Listing 3.1 to include expressions such as `a << b`.



There is a good chance you will get at least one question that involves shift operators. Be sure you have mastered them.

**Table 3.7 The Results of Some Bit-Shifting Operations on Sample 32-bit Integers**

Bit Pattern	Operation	Decimal Equivalent
0000 0000 0000 0000 0000 0000 0110 0011	starting x bits	99
0000 0000 0000 0000 0000 0011 0001 1000	after <code>x &lt;&lt; 3</code>	792
0000 0000 0000 0000 0000 0000 0001 1000	after <code>x &gt;&gt; 2</code>	24
1111 1111 1111 1111 1111 1111 1001 1101	starting y bits	-99
1111 1111 1111 1111 1111 1111 1111 1001	after <code>y &gt;&gt; 4</code>	-7
0000 0000 0000 1111 1111 1111 1111 1111	after <code>y &gt;&gt;&gt; 12</code>	1048575



When performing bit manipulations on primitives shorter than 32 bits, remember that the compiler promotes all operands to 32 bits before performing the operation.

Two final notes on the (right shift) shift operators: If the right operand is larger than 31 for operations on 32-bit integers, the compiler uses only the

five lowest order bits—values of 0 through 31, the remainder after division by 32—to control the number of bits shifted. With a 64-bit integer as the right operand, only the six lowest order bits (that is, values of 0 through 63, the remainder after division by 64) are used. Therefore, in line 1 of the following code fragment, `y` is shifted by only one bit; this also means that the sign bit is ignored, so in line 2, the right shift is not turned into a left shift by the minus sign:

```
1. x = y << 4097 ;
2. x = z >> -1 ;
```

## Shift and Bitwise Operations with Assignment

Note that the assignment operator `=` can be combined with the shift and bitwise operators, just as with the arithmetic operators. The result is as you would expect.

## Operators with Logical Expressions

The `&`, `|`, `^` (AND, OR, and XOR) operators used with integers also work with boolean values as expected. The compiler generates an error if both operands are not boolean. The tricky part (which you are almost guaranteed to run into) has to do with the `&&` and `||` “conditional AND” and “conditional OR” operators.

When the `&` and `|` operators are used in an expression, both operands are evaluated. For example, in the following code fragment, both the `( x >= 0 )` test and the call to the `testY` method will be executed:

```
if( ( x >= 0 ) & testY( y ) )
```

However, the conditional operators check the value of the left operand first and do not evaluate the right operand if it is not needed to determine the logical result. For instance, in the following code fragment, if `x` is `-1`, the result must be `false`. This means the `testY` method is never called:

```
if( ( x >= 0 ) && testY( y ) )
```

Similarly, if the `||` operator finds the left operand to be `true`, the result must be `true`, so the right operand is not evaluated.

These conditional logical operators, also known as *short circuit logical operators*, are used frequently in Java programming. For example, if it is possible that a `String` object reference has not been initialized, you might use the following code, where the test versus `null` ensures that the `equalsIgnoreCase` method will never be called with a `null` reference:

```
// ans is declared to be a String reference
if( ans != null && ans.equalsIgnoreCase( "yes" ) {}
```



There is a very good chance you will get one or more questions that require understanding the conditional operators. These questions frequently involve the need to determine whether a reference is `null`.

## Logical Operators with Assignment

Only the `&`, `|`, and `^` logical operators can be combined with `=`, producing the `&=`, `|=`, and `^=` logical operators. Naturally, in a logical expression with these combined operators, the left operand must be a `boolean` primitive variable. Note that there are bitwise operators that look the same but that work with integer primitives.

## The instanceof Operator

The `instanceof` operator tests the type of object the left operand refers to versus the type named by the right operand. The value returned is `true` if the object is of that type or if it inherits that type from a super type or interface implementation.

The right operand must be the name of a reference type, such as a class, an interface, or an array reference. Expressions using `instanceof` are unusual because the right operand cannot be an object. It must be the name of a reference type.

As an example of the use of `instanceof`, when the following code runs, both `"List"` and `"AbstractList"` are printed because the `Vector` class implements the `List` interface and descends from the `AbstractList` class.

```
Vector v = new Vector();
if( v instanceof List ){
    System.out.println("List") ;
}
if( v instanceof AbstractList ){
    System.out.println("AbstractList") ;
}
```



Remember that the **instanceof** operator can be used with interfaces and arrays as well as classes.

## The Conditional Assignment Operator

The conditional assignment operator is the only Java operator that takes three operands. It is essentially a shortcut for a structure that takes at least two statements, as shown in the following code fragment (assume that *x*, *y*, and *z* are `int` primitive variables that have been initialized):

```
1. // long way
2. if( x > y ) z = x ;
3. else z = y ;
4. //
5. // short way
6. z = x > y ? x : y ;
```

The operand to the left of the `?` must evaluate as boolean, and the other two operands must be of the same type, or convertible to the same type, which will be the type of the result. The result will be the operand to the left of the colon if the boolean is true; otherwise, it will be the right operand.

## More About Assignment

You have seen the use of the `=` operator in simple assignments such as `x = y + 3`. You should also note that the value assigned can be used by a further assignment operator, such as in the following statement, which assigns the calculated value to all four variables:

```
a = b = c = x = y + 3 ;
```

## Testing Object Equality

It is a source of great confusion to novice programmers that Java has two ways of thinking about the equality of objects. When used with object references, the `==` operator returns `true` only if both references are to the same object. This is illustrated in the following code fragment in which we create and compare some `Integer` object references:

```
1. Integer x1 = new Integer( 5 ) ;
2. Integer x2 = x1 ;
3. Integer x3 = new Integer( 5 ) ;
4. if( x1 == x2 ) System.out.println("x1 eq x2" );
5. if( x2 == x3 ) System.out.println("x2 eq x3" );
```

Executing this code will print only "x1 eq x2" because both variables refer to the same object. To test for equality of content, you have to use a method that can compare the *content* of the objects.

## The equals Method

In the Java standard library classes, the method that compares content is always named `equals` and takes an `Object` reference as input. For example, the `equals` method of the `Integer` class (paraphrased from the original for clarity) works like this:

```
1. public boolean equals(Object obj){
2.     if( obj == null ) return false ;
3.     if( !( obj instanceof Integer ) ) return false ;
4.     return this.value == ((Integer)obj).intValue() ;
5. }
```

Note that the `equals` method does not even look at the value of the other object until it has been determined that the other object reference is not `null` and that it refers to an `Integer` object.

The `equals` method in the `Object` class (the root of the entire Java hierarchy of classes) returns `true` only if

```
this == obj
```

Therefore, in the absence of an overriding `equals` method, the `==` operator and `equals` methods inherited from `Object` are equivalent.



Remember that the **equals** method compares content only if the two objects are of the identical type. For example, an **equals** test by an **Integer** object on a **Long** object always returns **false**, regardless of the numeric values. Also note that the signature of the **equals** method expects an **Object** reference as input. The compiler reports an error if you try to call **equals** with a primitive value. It is extremely likely that you will get one or more questions involving the **equals** method.

## The == with Strings Trap

One reason that it is easy to fall into the error of using `==` when you want `equals` is the behavior of `String` literals. The compiler optimizes storage of `String` literals by reusing them. In a large program, this can save a considerable amount of memory. Take the following code fragment:

```
1. String s1 = "YES" ;
2. String s2 = "YES" ;
3. if( s1 == s2 ) System.out.println("equal");
4. String s3 = new String( "YES" );
5. String s4 = new String( "YES" );
6. if( s3 == s4 ) System.out.println("s3 eq s4");
```

The `String` literal "YES" appears in both lines 1 and 2, but the compiler creates only one `String` object, referred to by both `s1` and `s2`. Thus, line 3 prints out "equal" and it appears to have been tested for equality of content with the `==` operator. However, the test in line 6 is always `false` because two distinct objects are involved.



One of the most common mistakes that new programmers make is using the `==` operator to compare `String` objects instead of the `equals` method. At least one question involving understanding the difference is likely to appear on the test.

## Array Initialization

An *array* in Java is a type of object that can contain a number of variables. These variables can be referenced only by the array index—a nonnegative integer. The first element in an array has an index of 0.

All of these contained variables, or elements, must be the same type, which is the type of the array. Every array has an associated `length` variable, established when the array is created, which you can access directly. If you try to address an element with an index that is outside the range of the array, an exception is generated. Java arrays are one dimensional, but an array can contain other arrays, which gives the effect of multiple dimensions.

You can have arrays of any of the Java primitives or reference variables. The important point to remember is that when created, primitive arrays will have default values assigned, but object references will all be `null`.

## Declaration

Like other variables, arrays must be declared before you use them. Declaration can be separate from the actual creation of the array. Here are some examples of declaring variables that are arrays of primitives (lines 1 through 3) and objects (lines 4 and 5):

```
1. int counts[] ;
2. int[] counts ; // 1 and 2 are equivalent
3. boolean flags [   ] ; // extra spaces are not significant
4. String names[] ;
5. MyClass[][] things ; // a two-dimensional array of objects
```

If the following lines were in a method and the method was executed, line 2 would print "counts = null" because the array object has not yet been constructed.

```
1. int counts[] ;
2. System.out.println("counts = " + counts ) ;
```

## Construction

You cannot do anything with an array variable until the array has been constructed with the `new` operator. The statement that constructs an array must give the size, as shown in the following code fragment, assumed to follow lines 1 through 6 in the previous code (the code in line 9 assumes that an integer primitive `nStrings` has been initialized):

```
7. counts = new int[20] ;
8. flags = new boolean[ 5 ] ;
9. names = new String[ nStrings ] ;
```

After this code executes, memory for the arrays will be reserved and initialized. The array reference variables will have references to array objects of known types. In other words, the type of an array object controls what can be stored in the indexed locations. You can test the type of an array variable with the `instanceof` operator, using a name for the reference type that looks like an array declaration. For example, using the `flags` variable initialized in line 8, the following test would result in `true`:

```
10. if( flags instanceof boolean[] )
```

Exactly what is in the array locations after construction depends on the type. Integer and floating-point primitive arrays have elements initialized to zero values. Arrays of boolean types have elements of `false` values. Arrays of object types have `null` references.

You can combine declaration of an array variable with construction, as shown in the following code examples:

```
1. float rates[] = new float[33] ;
2. String files[] = new String[ 1000 ] ;
```



You must remember the distinction between the status of arrays of primitives and the status of arrays of object references after the array is constructed. Arrays of primitives have elements that are initialized to default values. Arrays of objects have the value `null` in each element. You are practically guaranteed to have a related question on the exam.

## Combined Declaration, Construction, and Initialization

Java allows a statement format for combined declaration, construction, and initialization of arrays with literal values, as shown in the following code

examples (note that the `String` array defined in lines 2, 3, and 4 is two dimensional):

```
1. int[] fontSize = { 9, 11, 13, 15, 17 } ;
2. String[][] fontDesc = {
3.     {"TimesRoman", "bold"}, {"Courier", "italic"},
4.     {"ZapfDingBats", "normal"} } ;
```

## Initialization

To provide initial object references or primitive values other than the default, you have to address each element in the array. In the following code, we declare and create an array of `Rectangle` objects, and then create the `Rectangle` objects for each element:

```
1. Rectangle hotSpots[] = new Rectangle[10];
2. for( int i = 0 ; i < hotSpots.length ; i++ ){
3.     hotSpots[i] = new Rectangle(10 * i, 0, 10, 10);
4. }
```

The Java compiler checks the assignment of values to array positions just like it checks assignment to single variables. For example, the following code would not compile because the compiler knows that `1024` is outside the range of byte variables.

```
byte[] x = new byte[ 200 ];
x[0] = 1024 ;
```

At runtime, Java checks every array index against the known size of the array so that it is impossible to place data outside the reserved memory space. An attempt to use a bad index in an array operation results in an `ArrayIndexOutOfBoundsException` being thrown. We talk more about exceptions in Chapter 8, “Exceptions and Assertions.”

## Object Array Sample Question

The following sample question is related to object arrays. This example caused many errors in mock exam tests. We feel this difficulty illustrates two important points about taking the exam:

- ▶ Read the question carefully.
- ▶ Remember that the wrapper class names, although spelled like the primitives, always start with a capital letter.

1. What will happen when you try to compile and run the following application?

```
1. public class Example {  
2.     public Boolean flags[] = new Boolean[4] ;  
3.     public static void main(String[] args){  
4.         Example E = new Example();  
5.         System.out.println( "Flag 1 is " + E.flags[1] );  
6.     }  
7. }
```

- A. The text "Flag 1 is true" will be written to standard output.
- B. The text "Flag 1 is false" will be written to standard output.
- C. The text "Flag 1 is null" will be written to standard output.
- D. The compiler will object to line 2.

Answer C is correct. Most people forget that `Boolean` is a wrapper class for boolean values and thus the array creation statement in line 2 merely created the array. All of the references in that array are initialized to `null`.

# Exam Prep Practice Questions

## Question 1

What will be the result of calling the following method with an input of 2?

```
1. public int adder( int N ){  
2.     return 0x100 + N++ ;  
3. }
```

- A. The method will return **258**.
- B. The method will return **102**.
- C. The method will return **259**.
- D. The method will return **103**.

Answer A is correct. The hexadecimal constant `0x100` is 256 in decimal so adding 2 results in 258. The post increment of `N` will have no effect on the returned value. The method would return 102 if the literal constant were decimal, but it is not. Therefore, answer B is incorrect. Answers C and D represent incorrect arithmetic.

## Question 2

What happens when you attempt to compile and run the following code?

```
1. public class Logic {  
2.     static int minusOne = -1 ;  
3.     static public void main(String args[] ){  
4.         int N = minusOne >> 31 ;  
5.         System.out.println("N = " + N );  
6.     }  
7. }
```

- A. The program will compile and run, producing the output "**N = -1**".
- B. The program will compile and run, producing the output "**N = 1**".
- C. A runtime **ArithmeticException** will be thrown.
- D. The program will compile and run, producing the output "**N = 0**".

Answer A is correct. The `>>` operator extends the sign as the shift operation is performed. The program would have compiled and run, producing the output "`N = 1`" if the `>>>` operator, which shifts in a zero bit, had been specified, but it was not. Therefore, answer B is incorrect. An `ArithmeticException` is typically thrown due to integer division by zero, not

by a shift operation. Therefore, answer C is incorrect. Answer D does not occur because the `>>` operator extends the sign as the shift is performed.

## Question 3

---

What would be the result of running the following method with an input of **67**?

```
1. public int MaskOff( int n ){  
2.     return n | 3 ;  
3. }
```

- A. The method would return **3**.
- B. The method would return **64**.
- C. The method would return **67**.
- D. The method would return **0**.

Answer C is correct. The bit pattern of 67 is `1000011`, so the bitwise OR with 3 would not change the number. The method would have returned 3 if the bitwise AND operator `&` had been used, but this is the OR operator. Therefore, answer A is incorrect. The method would have returned 64 if the XOR operator `^` had been used, but it was not. Therefore, answer B is incorrect. Answer D cannot result from the OR of 67 with 3.

## Question 4

---

How many **String** objects are created in the following code?

```
1. String A, B, C ;  
2. A = new String( "1234" ) ;  
3. B = A ;  
4. C = A + B ;
```

- A. One
- B. Two
- C. Three
- D. Four

The correct answer is B. Both A and B refer to the same String object, whereas C refers to a String created by concatenating two copies of A. Therefore, only two String objects have been created, and all other answers are incorrect.

## Question 5

Which of the following versions of initializing a **char** variable would cause a compiler error? [Check all correct answers.]

- A. `char c = - 1 ;`
- B. `char c = '\u00FF' ;`
- C. `char c = (char) 4096 ;`
- D. `char c = 4096L ;`
- E. `char c = 'c' ;`
- F. `char c = "c" ;`

Answers A, D, and F are correct. In answer A, the literal creates a negative `int` that the compiler recognizes as being outside the normal range of `char` (the only unsigned integer primitive). In answer D, an explicit cast would be required to convert the literal `long` into a `char`. In answer F, the string literal could be used only to initialize a `String` object. The other options are legal assignments to a `char` primitive. Therefore, answers B, C, and E are incorrect. Note that questions that ask you to identify statements that will *not* compile are likely to appear on the exam.

## Question 6

What happens when you try to compile and run the following code?

```

1. public class EqualsTest{
2.     public static void main(String args[]){
3.         Long LA = new Long( 7 ) ;
4.         Long LB = new Long( 7 ) ;
5.         if( LA == LB )
6.             System.out.println("Equal");
7.         else System.out.println("Not Equal");
8.     }

```

- A. The program compiles but throws a runtime exception in line 5.
- B. The program compiles and prints **"Equal"**.
- C. The program compiles and prints **"Not Equal"**.
- D. The compiler objects to line 5.

Answer C is correct. When used with objects, the `==` operator tests for identity. Because `LA` and `LB` are different objects, the test fails. All other answers are incorrect.

## Question 7

What happens when you try to compile and run the following code?

```
1. public class EqualsTest{
2.     public static void main(String args[]){
3.         char A = '\u0005' ;
4.         if( A == 0x0005L ) {
5.             System.out.println("Equal");
6.         }
7.         else {
8.             System.out.println("Not Equal");
9.         }
10.    }
11. }
```

- A. The compiler reports “**Invalid character in input**” in line 3.
- B. The program compiles and prints “**Not Equal**”.
- C. The program compiles and prints “**Equal**”.
- D. The compiler objects to the use of **==** to compare a **char** and a **long**.

Answer C is correct. The compiler promotes variable A to a long before the comparison so answer D does not occur. The compiler does not report “Invalid character in input” in line 3 because this is the correct form for initializing a char primitive. Therefore, answer A is incorrect. Because answer C is correct, answer B cannot possibly be the correct answer.

## Question 8

In the following code fragment, you know that the **getParameter** call may return a **null** if there is no parameter named **size**:

```
1. int sz ;
2. public void init(){
3.     sz = 10 ;
4.     String tmp = getParameter("size");
5.     if( tmp != null X tmp.equals("BIG"))
6.         sz = 20 ;
7. }
```

Which logical operator should replace **X** in line 5 to ensure that a **NullPointerException** is not generated if **tmp** is **null**?

- A. **&**
- B. **&&**
- C. **|**
- D. **||**

The correct answer is B, the “short-circuited” AND operator. All of the other operators would attempt to run the equals method on the tmp variable, even if it were null, causing a NullPointerException. Therefore, answers A, C, and D are incorrect.

## Question 9

---

What would happen if you tried to compile and run the following code?

```
1. public class EqualsTest{
2.     public static void main(String args[]){
3.         Long L = new Long( 7 );
4.         if( L.equals( 7L ))
5.             System.out.println("Equal");
6.         else System.out.println("Not Equal");
7.     }
8. }
```

- A. The program would compile and print “Equal”.
- B. The program would compile and print “Not Equal”.
- C. The compiler would object to line 4.
- D. A runtime cast error would occur at line 4.

Answer C is correct. The compiler knows that the equals method takes an Object rather than a primitive as input. Because the program does not compile, answers A, B, and D are incorrect.

## Question 10

---

What would happen if you tried to compile and run the following code?

```
1. public class EqualsTest{
2.     public static void main(String args[]){
3.         Object A = new Long( 7 );
4.         Long L = new Long( 7 );
5.         if( A.equals( L ))
6.             System.out.println("Equal");
7.         else System.out.println("Not Equal");
8.     }
9. }
```

- A. The program would compile and print “Equal”.
- B. The program would compile and print “Not Equal”.
- C. The compiler would object to line 5.
- D. A runtime cast error would occur at line 5.

Answer A is correct. The `Long` object created in line 3 does not lose its identity when cast to `Object A`, so the `equals` method knows the class is correct and compares the values. Because answer A is correct, answer B is obviously incorrect. Answers C and D do not occur because this is the correct form for comparing objects with the `equals` method. Therefore, they are incorrect.

## Need to Know More?



Campione, Mary, Kathy Walrath, Alison Huml. *The Java Tutorial: A Short Course on the Basics, Third Edition*. Addison-Wesley, Boston, MA, 2000. ISBN 0201703939. A convenient bound version of Sun's online tutorial.



<http://java.sun.com/docs/books/jls/> is where the definitive Java language specification document is maintained in HTML form. This document has also been published as ISBN 0201310082, but most programmers will find the online documentation to be sufficient.



<http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html> is where the definitive JVM specification in the most current edition is maintained. It has detailed sections on the representation of primitive values such as interpretation of the `strictfp` modifier.



# Creating Java Classes

---

## Terms you'll need to understand:

- ✓ Access modifier
- ✓ **extends**
- ✓ **implements**
- ✓ Local or “automatic” variable
- ✓ Scope of variables
- ✓ Default constructor

## Techniques you'll need to master:

- ✓ Constructing a class definition using the modifiers **public**, **abstract**, and **final**
- ✓ Constructing definitions of classes that implement interfaces
- ✓ Declaring methods using the modifiers **public**, **private**, **protected**, **static**, **final**, **native**, and **synchronized**, as well as understanding the consequences of using them
- ✓ Declaring variables using the modifiers **public**, **private**, **protected**, **static**, **final**, **volatile**, and **transient**
- ✓ Using variables declared inside code blocks
- ✓ Differentiating between static, instance, and local variables
- ✓ Understanding the circumstances governing the use of default constructors