# OCA Java SE 8 Programmer I Exam Guide (Exams 1Z0-808)

by Kathy Sierra and Bert Bates
Oracle Press. (c) 2017. Copying Prohibited.

# Chapter 4: Operators

## CERTIFICATION OBJECTIVES

- Using Java Operators
- Use Parentheses to Override Operator Precedence
- Test Equality Between Strings and Other Objects Using == and equals( )
- Two-Minute Drill
- Self Test

If you've got variables, you're going to modify them. (Unless you're one of those new-fangled "FP" programmers.) You'll increment them, add them together, and compare one to another (in about a dozen different ways). In this chapter, you'll learn how to do all that in Java. As an added bonus, you'll learn how to do things that you'll probably never use in the real world, but that will almost certainly be on the exam.

## CERTIFICATION OBJECTIVE: JAVA OPERATORS (OCA OBJECTIVES 3.1, 3.2, AND 3.3)

*3.1 Use Java operators; including parentheses to override operator precedence*.

*3.2 Test equality between Strings and other objects using == and equals()*.

*3.3 Create if and if/else and ternary constructs*

Java operators produce new values from one or more operands. (Just so we're all clear, remember that operands are the things on the right or left side of the operator.) The result of most operations is either a `boolean` or numeric value. Because you know by now that Java is not C++, you won't be surprised that Java operators aren't typically overloaded. There are, however, a few exceptional operators that come overloaded:

- The + operator can be used to add two numeric primitives together or to perform a concatenation operation if either operand is a `String`.
- The &, |, and ^ operators can all be used in two different ways, although on this version of the exam, their bit-twiddling capabilities won't be tested.

Stay awake. Operators are often the section of the exam where candidates see their lowest scores. Additionally, operators and assignments are a part of many questions dealing with other topics—it would be a shame to nail a really tricky lambdas question only to blow it on a pre-increment statement.

### Assignment Operators

We covered most of the functionality of the equal (=) assignment operator in Chapter 3. To summarize:

- When assigning a value to a primitive, *size* matters. Be sure you know when implicit casting will occur, when explicit casting is necessary, and when truncation might occur.
- Remember that a reference variable isn't an object; it's a way to *get* to an object. (We know all you C++ programmers are just dying for us to say, "it's a pointer," but we're not going to.)
- When assigning a value to a reference variable, *type* matters. Remember the rules for supertypes, subtypes, and arrays.

Next we'll cover a few more details about the assignment operators that are on the exam, and when we get to the next chapter, we'll take a look at how the assignment operator = works with `Strings` (which are immutable).

**Exam Watch**

Don't spend time preparing for topics that are no longer on the exam! The following topics have NOT been on the exam since Java 1.4:

- Bit-shifting operators
- Bitwise operators
- Two's complement
- Divide-by-zero stuff

It's not that these aren't important topics; it's just that they're not on the exam anymore, and we're really focused on the exam. (Note: The reason we bring this up at all is because you might encounter mock exam questions on these topics—you can ignore those questions!)

### Compound Assignment Operators

There are actually 11 or so compound assignment operators, but only the 4 most commonly used (`+=`, `-=`, `*=`, and `/=`) are on the exam. The compound assignment operators let lazy typists shave a few keystrokes off their workload.

Here are several example assignments, first without using a compound operator:

```
y = y - 6;
x = x + 2 * 5;
```

Now, with compound operators:

```
y -= 6;
x += 2 * 5;
```

The last two assignments give the same result as the first two.

## Relational Operators

The exam covers six relational operators (<, <=, >, >=, ==, and `!=`). Relational operators always result in a `boolean` (`true` or `false`) value. This `boolean` value is most often used in an `if` test, as follows:

```
int x = 8;
if (x < 9) {
  // do something
}
```

But the resulting value can also be assigned directly to a `boolean` primitive:

```
class CompareTest {
  public static void main(String [] args) {
    boolean b = 100 > 99;
    System.out.println("The value of b is " + b);
  }
```

Java has four relational operators that can be used to compare any combination of integers, floating-point numbers, or characters:

- **>** Greater than
- **>=** Greater than or equal to
- **<** Less than
- **<=** Less than or equal to

Let's look at some legal comparisons:

```
class GuessAnimal {
  public static void main(String[] args) {
    String animal = "unknown";
    int weight = 700;
    char sex = 'm';
    double colorWaveLength = 1.630;
    if (weight >= 500) { animal = "elephant"; }
    if (colorWaveLength > 1.621) { animal = "gray " + animal; }
    if (sex <= 'f') { animal = "female " + animal; }
    System.out.println("The animal is a " + animal);
  }
}
```

In the preceding code, we are using a comparison between characters. It's also legal to compare a character primitive with any number (although it isn't great programming style). Running the preceding class will output the following:

```
The animal is a gray elephant
```

We mentioned that characters can be used in comparison operators. When comparing a character with a character or a character with a number, Java will use the Unicode value of the character as the numerical value for comparison.

### "Equality" Operators

Java also has two relational operators (sometimes called "equality operators") that compare two similar "things" and return a `boolean` (`true` or `false`) that represents what's true about the two "things" being equal. These operators are

- **==** Equal (also known as equal to)

- **!=** Not equal (also known as not equal to)

Each individual comparison can involve two numbers (including `char`), two `boolean` values, or two object reference variables. You can't compare incompatible types, however. What would it mean to ask if a `boolean` is equal to a `char`? Or if a `Button` is equal to a `String` array? (This is nonsense, which is why you can't do it.) There are four different types of things that can be tested:

- Numbers
- Characters
- Boolean primitives
- Object reference variables

So what does == look at? The value in the variable—in other words, the bit pattern.

**Equality for Primitives**

Most programmers are familiar with comparing primitive values. The following code shows some equality tests on primitive variables:

```
class ComparePrimitives {
  public static void main(String[] args) {
    System.out.println("char 'a' == 'a'? " + ('a' == 'a'));
    System.out.println("char 'a' == 'b'? " + ('a' == 'b'));
    System.out.println("5 != 6? " + (5 != 6));
    System.out.println("5.0 == 5L? " + (5.0 == 5L));
    System.out.println("true == false? " + (true == false));
  }
}
```

This program produces the following output:

```
char 'a' == 'a'? true
char 'a' == 'b'? false
5 != 6? true
5.0 == 5L? true
true == false? false
```

As you can see, if a floating-point number is compared with an integer and the values are the same, the == operator usually returns `true` as expected.

**Equality for Reference Variables**

As you saw earlier, two reference variables can refer to the same object, as the following code snippet demonstrates:

```
JButton a = new JButton("Exit");
JButton b = a;
```

---

**Exam Watch**

Don't mistake = for == in a `boolean` expression. The following is legal:

```
11. boolean b = false;
12. if (b = true) { System.out.println("b is true");
13. } else { System.out.println("b is false");  }
```

Look carefully! You might be tempted to think the output is `b is false`, but look at the `boolean` test in line 12. The `boolean` variable b is not being compared to `true`; it's being `set` to `true`. Once b is set to `true`, the `println` executes and we get `b is true`. The result of any assignment expression is the value of the variable following the assignment. This substitution of = for == works only with `boolean` variables because the `if` test can be done only on `boolean` expressions. Thus, this does not compile:

```
7. int x = 1;
8. if (x = 0) { }
```

Because x is an integer (and not a `boolean`), the result of (x = 0) is 0 (the result of the assignment). Primitive `ints` cannot be used where a `boolean` value is expected, so the code in line 8 won't work unless it's changed from an assignment (=) to an equality test (==) as follows:

```
8. if (x == 0) { }
```

---

After running this code, both variable a and variable b will refer to the same object (a `JButton` with the label `Exit`). Reference variables can be tested to see if they refer to the same object by using the == operator. Remember, the == operator is looking at the bits in the variable, so for reference variables, this means that if the bits in both reference variables are identical, then both refer to the same object. Look at the following code:

```
import javax.swing.JButton;
class CompareReference {
  public static void main(String[] args) {
    JButton a = new JButton("Exit");
    JButton b = new JButton("Exit");
    JButton c = a;
    System.out.println("Is reference a == b? " + (a == b));
    System.out.println("Is reference a == c? " + (a == c));
  }
}
```

This code creates three reference variables. The first two, a and b, are separate `JButton` objects that happen to have the same label. The third reference variable, c, is initialized to refer to the same object that a refers to. When this program runs, the following output is produced:

```
Is reference a == b? false
Is reference a == c? true
```

This shows us that a and c reference the same instance of a `JButton`. The == operator will not test whether two objects are "meaningfully equivalent," a concept we'll cover in much more detail in Chapter 6, when we look at the `equals()` *method* (as opposed to the equals *operator* we're looking at here).

**Equality for Strings and java.lang.Object.equals()**

We just used == to determine whether two reference variables refer to the same object. Because objects are so central to Java, every class in Java inherits a method from class `Object` that tests to see if two objects of the class are "equal." Not surprisingly, this method is called `equals()`. In this case of the `equals()` method, the phrase "meaningfully equivalent" should be used instead of the word "equal." So the `equals()` method is used to determine if two objects of the same class are "meaningfully equivalent." For classes that you create, you have the option of overriding the `equals()` method that your class inherited from class `Object` and creating your own definition of "meaningfully equivalent" for instances of your class.

In terms of understanding the `equals()` method for the OCA exam, you need to understand two aspects of the `equals()` method:

- What `equals()` means in class `Object`
- What `equals()` means in class `String`

**The equals() Method in Class Object** The `equals()` method in class `Object` works the same way that the == operator works. If two references point to the same object, the `equals()` method will return true. If two references point to different objects, even if they have the same values, the method will return false.

**The equals() Method in Class String** The `equals()` method in class `String` has been overridden. When the `equals()` method is used to compare two strings, it will return true if the strings have the same value, and it will return false if the strings have different values. For `String`'s `equals()` method, values ARE case sensitive.

Let's take a look at how the `equals()` method works in action (notice that the `Budgie` class did NOT override `Object.equals()`):

```
   class Budgie {
     public static void main(String[] args) {
     Budgie b1 = new Budgie();
     Budgie b2 = new Budgie();
     Budgie b3 = b1;

     String s1 = "Bob";
     String s2 = "Bob";
     String s3 = "bob";                  // lower case "b"

     System.out.println(b1.equals(b2));   // false, different objects
     System.out.println(b1.equals(b3));   // true, same objects
     System.out.println(s1.equals(s2));   // true, same values
     System.out.println(s1.equals(s3));   // false, values are case sensitive
   }
}
```

which produces the output:

```
false
true
true
false
```

**Equality for enums**

Once you've declared an `enum`, it's not expandable. At runtime, there's no way to make additional `enum` constants. Of course, you can have as many variables as you'd like refer to a given `enum` constant, so it's important to be able to compare two `enum` reference variables to see if they're "equal"—that is, do they refer to the same `enum` constant? You can use either the `==` operator or the `equals()` method to determine whether two variables are referring to the same `enum` constant:

```
class EnumEqual {
  enum Color {RED, BLUE}                   // ; is optional
  public static void main(String[] args) {
    Color c1 = Color.RED;  Color c2 = Color.RED;
    if(c1 == c2) { System.out.println("=="); }
    if(c1.equals(c2)) { System.out.println("dot equals"); }
} }
```

(We know `} }` is ugly; we're prepping you.) This produces the output:

```
==
dot equals
```

## instanceof Comparison

The `instanceof` operator is used for object reference variables only, and you can use it to check whether an object is of a particular type. By "type," we mean class or interface type—in other words, whether the object referred to by the variable on the left side of the operator passes the IS-A test for the class or interface type on the right side. (Chapter 2 covered IS-A relationships in detail.) The following simple example,

```
public static void main(String[] args) {
  String s = new String("foo");
  if (s instanceof String) {
    System.out.print("s is a String");
  }
}
```

prints this:

```
s is a String
```

Even if the object being tested is not an actual instantiation of the class type on the right side of the operator, `instanceof` will still return `true` if the object being compared is *assignment compatible* with the type on the right.

The following example demonstrates a common use for `instanceof`: testing an object to see if it's an instance of one of its subtypes before attempting a downcast:

```
class A { }
class B extends A {
  public static void main (String [] args) {
    A myA = new B();
    m2(myA);
  }
  public static void m2(A a) {
    if (a instanceof B)
      ((B)a).doBstuff();     // downcasting an A reference
                             // to a B reference
  }
  public static void doBstuff() {
    System.out.println("'a' refers to a B");
  }
}
```

The code compiles and produces this output:

```
'a' refers to a B
```

In examples like this, the use of the `instanceof` operator protects the program from attempting an illegal downcast.

You can test an object reference against its own class type or any of its superclasses. This means that *any* object reference will evaluate to `true` if you use the `instanceof` operator against type `Object`, as follows:

```
B b = new B();
if (b instanceof Object) {
   System.out.print("b is definitely an Object");
}
```

This prints

```
b is definitely an Object
```

### Exam Watch

Look for `instanceof` questions that test whether an object is an instance of an interface when the object's class implements the interface indirectly. An indirect implementation occurs when one of an object's superclasses implements an interface, but the actual class of the instance does not. In this example,

```
interface Foo { }
class A implements Foo { }
class B extends A { }
...
A a = new A();
B b = new B();
```

the following are true:

```
a instanceof Foo
b instanceof A
b instanceof Foo  // implemented indirectly
```

An object is said to be of a particular interface type (meaning it will pass the `instanceof` test) if any of the object's superclasses implement the interface.

In addition, it is legal to test whether the `null` reference is an instance of a class. This will always result in `false`, of course. This example,

```
class InstanceTest {
  public static void main(String [] args) {
     String a = null;
     boolean b = null instanceof String;
     boolean c = a instanceof String;
     System.out.println(b + " " + c);
  }}
```

prints this:

```
false false
```

**instanceof Compiler Error**

You can't use the `instanceof` operator to test across two different class hierarchies. For instance, the following will NOT compile:

```
class Cat { }
class Dog {
  public static void main(String [] args) {
    Dog d = new Dog();
    System.out.println(d instanceof Cat);
  }
}
```

Compilation fails—there's no way d could ever refer to a `Cat` or a subtype of `Cat`.

### Exam Watch

Remember that arrays are objects, even if the array is an array of primitives. Watch for questions that look something like this:

```
int [] nums = new int[3];
if (nums instanceof Object) { } // result is true
```

An array is always an instance of `Object`. Any array.

Table 4-1 summarizes the use of the `instanceof` operator given the following:

```
interface Face { }
class Bar implements Face{ }
class Foo extends Bar { }
```

## Arithmetic Operators

We're sure you're familiar with the basic arithmetic operators:

- **+** addition
- **–** subtraction

- ***** multiplication
- **/** division

**Table 4-1: Operands and Results Using `instanceof` Operator**

| First Operand (Reference Being Tested) | instanceof Operand (Type We're Comparing the Reference Against) | Result |
|---|---|---|
| `null` | Any class or interface type | `false` |
| `Foo` instance | `Foo, Bar, Face, Object` | `true` |
| `Bar` instance | `Bar, Face, Object` | `true` |
| `Bar` instance | `Foo` | `false` |
| `Foo [ ]` | `Foo, Bar, Face` | `compiler error` |
| `Foo [ ]` | `Object` | `true` |
| `Foo [ 1 ]` | `Foo, Bar, Face, Object` | `true` |

These can be used in the standard way:

```
int x = 5 * 3;
int y = x - 4;
System.out.println("x - 4 is " + y);  // Prints 11
```

### The Remainder (%) Operator (a.k.a. the Modulus Operator)

One operator you might not be as familiar with is the remainder operator: `%`. The remainder operator divides the left operand by the right operand, and the result is the remainder, as the following code demonstrates:

```
class MathTest {
  public static void main (String [] args) {
    int x = 15;
    int y = x % 4;
    System.out.println("The result of 15 % 4 is the "
      + "remainder of 15 divided by 4. The remainder is " + y);
  }
}
```

Running class `MathTest` prints the following:

```
The result of 15 % 4 is the remainder of 15 divided by 4. The remainder
is 3
```

(Remember: Expressions are evaluated from left to right by default. You can change this sequence, or *precedence*, by adding parentheses. Also remember that the `*`, `/`, and `%` operators have a higher precedence than the `+` and `-` operators.)

---

**Exam Watch**

When working with `ints`, the remainder operator (a.k.a. the modulus operator) and the division operator relate to each other in an interesting way:

- The modulus operator throws out `everything but` the remainder.
- The division operator throws out the remainder.

---

### String Concatenation Operator

The plus sign can also be used to concatenate two strings together, as we saw earlier (and as we'll definitely see again):

```
String animal = "Gray " + "elephant";
```

String concatenation gets interesting when you combine numbers with `Strings`. Check out the following:

```
String a = "String";
int b = 3;
int c = 7;
System.out.println(a + b + c);
```

Will the + operator act as a plus sign when adding the `int` variables b and c? Or will the + operator treat 3 and 7 as characters and concatenate them individually? Will the result be `String10` or `String37`? Okay, you've had long enough to think about it.

The `int` values were simply treated as characters and glued on to the right side of the `String`, giving the result:

```
String37
```

So we could read the previous code as

"Start with the value `String`, and concatenate the character 3 (the value of b) to it, to produce a new string `String3`, and then concatenate the character 7 (the value of c) to that, to produce a new string `String37`. Then print it out."

However, if you put parentheses around the two `int` variables, as follows,

```
System.out.println(a + (b + c));
```

you'll get this:

```
String10
```

Using parentheses causes the `(b + c)` to evaluate first, so the rightmost + operator functions as the addition operator, given that both operands are `int` values. The key point here is that within the parentheses, the left-hand operand is not a `String`. If it were, then the + operator would perform `String` concatenation. The previous code can be read as

> "Add the values of `b and c` together, and then take the sum and convert it to a `String` and concatenate it with the `String` from variable a."
>
> The rule to remember is this:
>
> *If either operand is a `String`, the + operator becomes a `String` concatenation operator. If both operands are numbers, the + operator is the addition operator*.

You'll find that sometimes you might have trouble deciding whether, say, the left-hand operator is a `String` or not. On the exam, don't expect it always to be obvious. (Actually, now that we think about it, don't expect it *ever* to be obvious.) Look at the following code:

```
System.out.println(x.foo() + 7);
```

You can't know how the + operator is being used until you find out what the `foo()` method returns! If `foo()` returns a `String`, then 7 is concatenated to the returned `String`. But if `foo()` returns a number, then the + operator is used to add 7 to the return value of `foo()`.

Finally, you need to know that it's legal to mush together the compound additive operator (+=) and `Strings`, like so:

```
String s = "123";
s += "45";
s += 67;
System.out.println(s);
```

Since both times the += operator was used and the left operand was a `String`, both operations were concatenations, resulting in

```
1234567
```

---

**Exam Watch**

If you don't understand how `String` concatenation works, especially within a `print` statement, you could actually fail the exam even if you know the rest of the answers to the questions! Because so many questions ask "What is the result?", you need to know not only the result of the code running but also how that result is printed. Although at least a few questions will directly test your `String` knowledge, `String` concatenation shows up in other questions on virtually every objective. Experiment! For example, you might see a line such as this:

```
int b = 2;
System.out.println("" + b + 3);
```

It prints this:

```
23
```

But if the `print` statement changes to this:

```
System.out.println(b + 3);
```

The printed result becomes

```
5
```

---

### Increment and Decrement Operators

Java has two operators that will increment or decrement a variable by exactly one. These operators are either two plus signs (++) or two minus signs (--):

- **++** Increment (prefix and postfix)
- **--** Decrement (prefix and postfix)

The operator is placed either before (prefix) or after (postfix) a variable to change its value. Whether the operator comes before or after the operand can change the outcome of an expression. Examine the following:

```
1. class MathTest {
2.   static int players = 0;
3.     public static void main (String [] args) {
4.       System.out.println("players online: " + players++);
5.       System.out.println("The value of players is "
                            + players);
6.       System.out.println("The value of players is now "
                            + ++players);
7.     }
8. }
```

Notice that in the fourth line of the program the increment operator is *after* the variable `players`. That means we're using the postfix increment operator, which causes `players` to be incremented by one but only *after* the value of `players` is used in the expression. When we run this program, it outputs the following:

```
%java MathTest
players online: 0
The value of players is 1
The value of players is now 2
```

Notice that when the variable is written to the screen, at first it says the value is `0`. Because we used the postfix increment operator, the increment doesn't happen until after the `players` variable is used in the `print` statement. Get it? The "post" in postfix means *after*. Line 5 doesn't increment `players`; it just outputs its value to the screen, so the newly incremented value displayed is 1. Line 6 applies the prefix increment operator to `players`, which means the increment happens *before* the value of the variable is used, so the output is `2`.

Expect to see questions mixing the increment and decrement operators with other operators, as in the following example:

```
int x = 2; int y = 3;
if ((y == x++) | (x < ++y)) {
  System.out.println("x = " + x + " y = " + y);
 }
```

The preceding code prints this:

```
x = 3 y = 4
```

You can read the code as follows: "If 3 is equal to 2 OR 3 < 4"

The first expression compares `x` and `y`, and the result is `false`, because the increment on `x` doesn't happen until *after* the `==` test is made. Next, we increment `x`, so now `x` is `3`. Then we check to see if `x` is less than `y`, but we increment `y` *before* comparing it with `x`! So the second logical test is `(3 < 4)`. The result is `true`, so the `print` statement runs.

As with `String` concatenation, the increment and decrement operators are used throughout the exam, even on questions that aren't trying to test your knowledge of how those operators work. You might see them in questions on `for` loops, exceptions, or even threads. Be ready.

---

**Exam Watch**

Look out for questions that use the increment or decrement operators on a `final` variable. Because `final` variables can't be changed, the increment and decrement operators can't be used with them, and any attempt to do so will result in a compiler error. The following code won't compile:

```
final int x = 5;
int y = x++;
```

It produces this error:

```
Test.java:4: cannot assign a value to final variable x
int y = x++;
        ^
```

You can expect a violation like this to be buried deep in a complex piece of code. If you spot it, you know the code won't compile, and you can move on without working through the rest of the code.

This question might seem to be testing you on some complex arithmetic operator trivia, when, in fact, it's testing you on your knowledge of the `final` modifier.

---

## Conditional Operator

The conditional operator is a *ternary* operator (it has *three* operands) and is used to evaluate `boolean` expressions—much like an `if` statement, except instead of executing a block of code if the test is `true`, a conditional operator will assign a value to a variable. In other words, the goal of the conditional operator is to decide which of two values to assign to a variable. This operator is constructed using a ? (question mark) and a `:` (colon). The parentheses are optional. Here is its structure:

```
x = (boolean expression) ? value to assign if true : value to assign if false
```

Let's take a look at a conditional operator in code:

```
class Salary {
  public static void main(String [] args) {
    int numOfPets = 3;
    String status = (numOfPets<4) ? "Pet limit not exceeded"
                    : "too many pets";
    System.out.println("This pet status is " + status);
  }
}
```

You can read the preceding code as "Set `numOfPets` equal to 3".

Next we're going to assign a `String` to the status variable. If `numOfPets` is less than `4`, assign `"Pet limit not exceeded"` to the `status` variable; otherwise, assign `"too many pets"` to the `status` variable.

A conditional operator starts with a `boolean` operation, followed by two possible values for the variable to the left of the assignment (=) operator. The first value (the one to the left of the colon) is assigned if the conditional (`boolean`) test is `true`, and the second value is assigned if the conditional test is `false`. You can even nest conditional operators into one statement:

```
class AssignmentOps {
  public static void main(String [] args) {
   int sizeOfYard = 10;
   int numOfPets = 3;
   String status = (numOfPets<4)?"Pet count OK"
       :(sizeOfYard > 8)? "Pet limit on the edge"
         :"too many pets";
    System.out.println("Pet status is " + status);
  }
}
```

Don't expect many questions using conditional operators, but you might get one.

## Logical Operators

The exam objectives specify six "logical" operators (&, `|`, ^, `!`, &&, and `||`). Some Oracle documentation uses other terminology for these operators, but for our purposes and in the exam objectives, these six are the logical operators.

### Bitwise Operators (Not an Exam Topic!)

Okay, this is going to be confusing. Of the six logical operators just listed, three of them (&, `|`, and ^) can also be used as "bitwise" operators. Bitwise operators were included in previous versions of the exam, but they're NOT on the Java 6, Java 7, or Java 8 exam. We bring them up here just so you have a more complete picture of the logical operators.

Here are several legal statements that use bitwise operators:

```
byte b1 = 6 & 8;
byte b2 = 7 | 9;
byte b3 = 5 ^ 4;
System.out.println(b1 + " " + b2 + " " + b3);
```

Bitwise operators compare two variables bit by bit and return a variable whose bits have been set based on whether the two variables being compared had respective bits that were either both "on" (&), one or the other "on" (`|`), or exactly one "on" (^). By the way, when we run the preceding code, we get

```
0 15 1
```

---

**Exam Watch**

Having said all this about bitwise operators, the key thing to remember is this:

BITWISE OPERATORS ARE NOT ON THE Java 6, Java 7, or Java 8 EXAM!

---

### Short-Circuit Logical Operators

Five logical operators on the exam are used to evaluate statements that contain more than one `boolean` expression. The most commonly used of the five are the two *short-circuit* logical operators:

n && Short-circuit AND

n `||` Short-circuit OR

They are used to link little `boolean` expressions together to form bigger `boolean` expressions. The `&&` and `||` operators evaluate only `boolean` values. For an AND (&&) expression to be `true`, both operands must be `true`. For example:

```
if ((2 < 3) && (3 < 4)) { }
```

The preceding expression evaluates to `true` because *both* operand one `(2 < 3)` and operand two `(3 < 4)` evaluate to `true`.

*The short-circuit feature of the && operator is so named because it doesn't waste its time on pointless evaluations*. A short-circuit `&&` evaluates the left side of the operation first (operand one), and if it resolves to `false`, the `&&` operator doesn't bother looking at the right side of the expression (operand two) since the `&&` operator already *knows* that the complete expression can't possibly be `true`.

```
class Logical {
  public static void main(String [] args) {
    boolean b1 = false, b2 = false;
    boolean b3 = (b1 == true) && (b2 = true);  // will b2 be set to true?
    System.out.println(b3 + " " + b2);
  }
}
```

When we run the preceding code, the **assignment** `(b2 = true)` never runs because of the short-circuit operator, so the output is

```
%java Logical
false false
```

The `||` operator is similar to the `&&` operator, except that it evaluates to `true` if EITHER of the operands is true. If the first operand in an OR operation is `true`, the result will be `true`, so the short-circuit `||` doesn't waste time looking at the right side of the equation. If the first operand is `false`, however, the short-circuit `||` has to evaluate the second operand to see if the result of the OR operation will be `true` or `false`. Pay close attention to the following example; you'll see quite a few questions like this on the exam:

```
 1. class TestOR {
 2.   public static void main(String[] args) {
 3.     if ((isItSmall(3)) || (isItSmall(7))) {
 4.       System.out.println("Result is true");
 5.     }
 6.     if ((isItSmall(6)) || (isItSmall(9))) {
 7.       System.out.println("Result is true");
 8.     }
 9.   }
10.
11.   public static boolean isItSmall(int i) {
12.     if (i < 5) {
13.       System.out.println("i < 5");
14.       return true;
15.     } else {
16.       System.out.println("i >= 5");
17.       return false;
18.     }
19.   }
20. }
```

What is the result?

```
% java TestOR
i < 5
Result is true
i >= 5
i >= 5
```

Here's what happened when the `main()` method ran:

1. When we hit line 3, the first operand in the `||` expression (in other words, the *left* side of the `||` operation) is evaluated.
2. The `isItSmall(3)` method is invoked, prints `"i < 5"`, and returns `true`.
3. Because the *first* operand in the `||` expression on line 3 is `true`, the `||` operator doesn't bother evaluating the second operand. So we never see the `"i >= 5"` that would have printed had the *second* operand been evaluated (which would have invoked `isItSmall (7)`).

4. Line 6 is evaluated, beginning with the *first* operand in the `||` expression.
5. The `isItSmall(6)` method is called, prints `"i >= 5"`, and returns `false`.
6. Because the *first* operand in the `||` expression on line 6 is `false`, the `||` operator can't skip the *second* operand; there's still a chance the expression can be `true`, if the *second* operand evaluates to `true`.
7. The `isItSmall(9)` method is invoked and prints `"i >= 5"`.
8. The `isItSmall(9)` method returns `false`, so the expression on line 6 is `false`, and thus line 7 never executes.

**Exam Watch**

The `||` and `&&` operators work only with boolean operands. The exam may try to fool you by using integers with these operators:

```
if (5 && 6) { }
```

It looks as though we're trying to do a bitwise AND on the bits representing the integers 5 and 6, but the code won't even compile.

### Logical Operators (not Short-Circuit)

There are two *non-short-circuit* logical operators:

- & Non-short-circuit AND
- | Non-short-circuit OR

These operators are used in logical expressions just like the && and `||` operators are used, but because they aren't the short-circuit operators, they evaluate both sides of the expression—always! They're inefficient. For example, even if the *first* operand (left side) in an & expression is `false`, the *second* operand will still be evaluated—even though it's now impossible for the result to be `true`! And the `|` is just as inefficient: if the *first* operand is `true`, the Java Virtual Machine (JVM) still plows ahead and evaluates the *second* operand even when it knows the expression will be `true` regardless.

You'll find a lot of questions on the exam that use both the short-circuit and non-short-circuit logical operators. You'll have to know exactly which operands are evaluated and which are not, because the result will vary depending on whether the second operand in the expression is evaluated. Consider this,

```
int z = 5;
if(++z > 5 || ++z > 6) z++;   // z = 7 after this code
```

versus this:

```
int z = 5;
if(++z > 5 | ++z > 6) z++;   // z = 8 after this code
```

### Logical Operators ^ and !

The last two logical operators on the exam are

- **^** Exclusive-OR (XOR)
- **!** Boolean invert

The ^ (exclusive-OR) operator evaluates only `boolean` values. The ^ operator is related to the non-short-circuit operators we just reviewed, in that it always evaluates *both* the left and right operands in an expression. For an exclusive-OR (^) expression to be `true`, EXACTLY one operand must be `true`. This example,

```
System.out.println("xor " + ((2 < 3) ^ (4 > 3)));
```

produces this output:

```
xor false
```

The preceding expression evaluates to `false` because BOTH operand one `(2 < 3)` and operand two `(4 > 3)` evaluate to `true`.

The `!` (boolean invert) operator returns the opposite of a boolean's current value. The following statement,

```
if(!(7 == 5)) { System.out.println("not equal"); }
```

can be read "If it's not true that 7 == 5," and the statement produces this output:

```
not equal
```

Here's another example using booleans:

```
boolean t = true;
boolean f = false;
System.out.println("! " + (t & !f) + " " + f);
```

It produces this output:

```
! true false
```

In the preceding example, notice that the & test succeeded (printing `true`) and that the value of the `boolean` variable `f` did not change, so it printed `false`.

## Operator Precedence

The OCA 8 exam has reintroduced the topic of operator precedence. As you probably already know but will definitely see demonstrated in this section, when several operators are used in combination, the order in which they are evaluated can alter the result of the expression.

### Operator Precedence Rant

Allow us to rant for a minute here. Memorizing operator precedence was on the old SCJP 1.2 exam about 15 years ago. Starting with the SCJP 1.4 exam, and for all the exams until the OCA 8, operator precedence has not been on the exam. For a glorious 15 years, candidates didn't have to do this bit of memorization. Sadly, this topic snuck its way back into the exam for OCA 8. Why do we care so much about this? Take a look at this code:

```
System.out.println(true & false == false | true);
```

What result would you expect? Imagine a more realistic version, evaluating some booleans:

```
System.out.println(b1 & b2 == b3 | b4);
```

What would you guess the programmer's intention was here? There are two likely scenarios:

**Scenario 1: (b1 & b2) == (b3 | b4)** If this was the programmer's intention, then he just created a bug.

**Scenario 2: b1 & (b2 == b3) | b4** If this was the programmer's intention, then the code will work as intended, but his boss and fellow workers will want to strangle him.

This is a long-winded way to say that when you're writing code, you shouldn't rely on everyone's memory of operator precedence. You should just use parentheses like civilized people do.

### The Actual Beginning of the Operator Precedence Section

Table 4-2 lists the most commonly used operators and their relative precedence, starting at the top with the highest precedence operators and ending at the bottom with the lowest. (Note, not all of Java's operators are in this table!)

**Table 4-2: Precedence Hierarchy of Common Operators (from Highest to Lowest)**

| Types of Operators | Symbols | Example Uses |
|---|---|---|
| Unary operators | `-, !, ++, --` | `-7 * 4, !myBoolean` |
| Multiplication, division, modulus | `*, /, %` | `7 % 4` |
| Addition, subtraction | `+, -` | `7 + 4` |
| Relational operators | `<, >, <=, >=` | `y > x` |
| Equality operators | `==, !=` | `y != x` |
| Logical operators (& beats \|) | `&, \|` | `myBool & yourBool` |
| Short-circuit (&& beats \|\|) | `&&, \|\|` | `myBool \|\| yourBool` |
| Assignment operators | `=, +=, -=` | `X += 5;` |

JavaRanch (we know, we know, "Coderanch") moderator Fritz Walraven shared this tip with us. We like it, and we're passing it along to you: for the table above, you might make up a word like "UMARELSA," or a sentence using those first letters, to help you remember the precedence rules!

There are three important general rules for determining how Java will evaluate expressions with operators:

- When two operators of the same precedence are in the same expression, Java evaluates the expression from left to right.
- When parts of an expression are placed in parentheses, those parts are evaluated first.
- When parentheses are nested, the innermost parentheses are evaluated first.

A good way to burn these precedence rules into your brain is to—as always—write some test code and play around with it. We've added an example of some test code that demonstrates several of the precedence hierarchy rules listed here. As you can see, we often compared parentheses-free expressions with their parentheses-rich counterparts to prove the rules:

```
System.out.println((-7 - 4) + " " + (-(7 - 4)));         // unary (-7), beats minus
                                                         // output: -11 -3

System.out.println((2 + 3 * 4) + " " + ((2 + 3) * 4));   // * beats +
                                                         // output: 14 20

System.out.println(7 > 5 && 2 > 3);                      // > beats &&
                                                         // output: false

System.out.print((true & false == false | true) + " ");  // == beats & System.out.
print(((true & false) == (false | true)));    // output: true
```

And to repeat, the output is:

```
-11 -3
14 20
false
true false
```

We're so sorry that you need to memorize this stuff, but if you master what's in this short section, you should be able to handle whatever weird precedence-related questions the exam throws at you.

## CERTIFICATION SUMMARY

If you've studied this chapter diligently, you should have a firm grasp on Java operators, and you should understand what equality means when tested with the == operator. Let's review the highlights of what you've learned in this chapter.

The logical operators (&&, ||, &, |, and ^) can be used only to evaluate two `boolean` expressions. The difference between && and & is that the && operator won't bother testing the right operand if the left evaluates to `false`, because the result of the && expression can never be `true`. The difference between || and | is that the || operator won't bother testing the right operand if the left evaluates to `true` because the result is already known to be `true` at that point.

The == operator can be used to compare values of primitives, but it can also be used to determine whether two reference variables refer to the same object.

The `instanceof` operator is used to determine whether the object referred to by a reference variable passes the IS-A test for a specified type.

The + operator is overloaded to perform `String` concatenation tasks and can also concatenate `Strings` and primitives, but be careful—concatenation can be tricky.

The conditional operator (a.k.a. the "ternary operator") has an unusual, three-operand syntax—don't mistake it for a complex assert statement.

The ++ and -- operators will be used throughout the exam, and you must pay attention to whether they are prefixed or postfixed to the variable being updated.

Even though you should use parentheses in real life, for the exam you should memorize Table 4-2 so you can determine how code that doesn't use parentheses for complex expressions will be evaluated, based on Java's operator-precedence hierarchy.

Be prepared for a lot of exam questions involving the topics from this chapter. Even within questions testing your knowledge of another objective, the code will frequently use operators, assignments, object and primitive passing, and so on.

## TWO-MINUTE DRILL

Here are some of the key points from each section in this chapter.

**Relational Operators (OCA Objectives 3.1 and 3.2)**

- Relational operators always result in a `boolean` value (`true` or `false`).
- There are six relational operators: >, >=, <, <=, ==, and !=. The last two (== and !=) are sometimes referred to as *equality operators*.
- When comparing characters, Java uses the Unicode value of the character as the numerical value.
- Equality operators
    - There are two equality operators: == and !=.
    - Four types of things can be tested: numbers, characters, booleans, and reference variables.
    - When comparing reference variables, == returns `true` only if both references refer to the same object.

### instanceof Operator (OCA Objective 3.1)

- `instanceof` is for reference variables only; it checks whether the object is of a particular type.
- The `instanceof` operator can be used only to test objects (or `null`) against class types that are in the same class hierarchy.
- For interfaces, an object passes the `instanceof` test if any of its superclasses implement the interface on the right side of the `instanceof` operator.

### Arithmetic Operators (OCA Objective 3.1)

- The four primary math operators are add (+), subtract (`–`), multiply (`*`), and divide (`/`).
- The remainder (a.k.a. modulus) operator (%) returns the remainder of a division.
- Expressions are evaluated from left to right, unless you add parentheses, or unless some operators in the expression have higher precedence than others.
- The `*`, `/`, and `%` operators have higher precedence than + and `–`.

### String Concatenation Operator (OCA Objective 3.1)

- If either operand is a `String`, the + operator concatenates the operands.
- If both operands are numeric, the + operator adds the operands.

### Increment/Decrement Operators (OCA Objective 3.1)

- Prefix operators (e.g. `--x`) run before the value is used in the expression.
- Postfix operators (e.g., `x++`) run after the value is used in the expression.
- In any expression, both operands are fully evaluated *before* the operator is applied.
- Variables marked `final` cannot be incremented or decremented.

### Ternary (Conditional) Operator (OCA Objective 3.3)

- Returns one of two values based on the state of its `boolean` expression.
  - Returns the value after the `?` if the expression is `true`.
  - Returns the value after the `:` if the expression is `false`.

### Logical Operators (OCA Objective 3.1)

- The exam covers six "logical" operators: &, `|`, `^`, `!`, &&, and `||`.
- Work with two expressions (except for `!`) that must resolve to boolean values.
- The && and & operators return `true` only if both operands are `true`.
- The `||` and | operators return `true` if either or both operands are `true`.
- The && and `||` operators are known as short-circuit operators.
- The && operator does not evaluate the right operand if the left operand is `false`.
- The `||` does not evaluate the right operand if the left operand is `true`.
- The & and `|` operators always evaluate both operands.
- The `^` operator (called the "logical XOR") returns `true` if exactly one operand is `true`.
- The `!` operator (called the "inversion" operator) returns the opposite value of the boolean operand it precedes.

### Parentheses and Operator Precedence (OCA Objective 3.1)

- In real life, use parentheses to clarify your code, and force Java to evaluate expressions as intended.
- For the exam, memorize Table 4-2 to determine how parentheses-free code will be evaluated.

## SELF TEST

**1.** Given: ?

```
class Hexy {
  public static void main(String[] args) {
    int i = 42;
    String s = (i<40)?"life":(i>50)?"universe":"everything";
    System.out.println(s);
  }
}
```

What is the result?

A. `null`

B. `life`

C. `universe`

D. `everything`

E. Compilation fails

F. An exception is thrown at runtime

**2.** Given: ?

```
public class Dog {
  String name;
  Dog(String s) { name = s; }
  public static void main(String[] args) {
    Dog d1 = new Dog("Boi");
    Dog d2 = new Dog("Tyri");
    System.out.print((d1 == d2) + " ");
    Dog d3 = new Dog("Boi");
    d2 = d1;
    System.out.print((d1 == d2) + " ");
    System.out.print((d1 == d3) + " ");
  }
}
```

What is the result?

A. `true true true`

B. `true true false`

C. `false true false`

D. `false true true`

E. `false false false`

F. An exception will be thrown at runtime

**3.** Given: ?

```
class Fork {
  public static void main(String[] args) {
    if(args.length == 1 | args[1].equals("test")) {
      System.out.println("test case");
    } else {
      System.out.println("production " + args[0]);
    }
  }
}
```

And the command-line invocation:

```
java Fork live2
```

What is the result?

A. `test case`

B. `production live2`

C. `test case live2`

D. Compilation fails

E. An exception is thrown at runtime

**4.** Given: ?

```
class Feline {
  public static void main(String[] args) {
    long x = 42L;
    long y = 44L;
    System.out.print(" " + 7 + 2 + " ");
    System.out.print(foo() + x + 5 + " ");
    System.out.println(x + y + foo());
  }
  static String foo() { return "foo"; }
}
```

What is the result?

A. `9 foo47 86foo`

B. `9 foo47 4244foo`

C. `9 foo425 86foo`

D. `9 foo425 4244foo`

E. `72 foo47 86foo`

F. `72 foo47 4244foo`

G. `72 foo425 86foo`

H. `72 foo425 4244foo`

I. Compilation fails

**5.** **Note**: Here's another old-style drag-and-drop question…just in case. ?

Place the fragments into the code to produce the output `33`. Note that you must use each fragment exactly once.

```
CODE:
class Incr {
  public static void main(String[] args) {
    Integer x = 7;
    int y = 2;

    x    ___  ___;
    ___  ___  ___;
    ___  ___  ___;
    ___  ___  ___;

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

FRAGMENTS:

```
y   y   y   y

y   x   x

-=  *=  *=  *=
```

**6.** Given: ?

```
public class Cowboys {
  public static void main(String[] args) {
    int x = 12;
    int a = 5;
    int b = 7;
    System.out.println(x/a + " " + x/b);
  }
}
```

What is the result? (Choose all that apply.)

A. `2 1`

B. `2 2`

C. `3 1`

D. `3 2`

E. An exception is thrown at runtime

**7.** Given:

```
 3. public class McGee {
 4.   public static void main(String[] args) {
 5.     Days d1 = Days.TH;
 6.     Days d2 = Days.M;
 7.     for(Days d: Days.values()) {
 8.       if(d.equals(Days.F)) break;
 9.       d2 = d;
10.     }
11.     System.out.println((d1 == d2)?"same old" : "newly new");
12.   }
13.   enum Days {M, T, W, TH, F, SA, SU};
14. }
```

What is the result?

A. `same old`

B. `newly new`

C. Compilation fails due to multiple errors

D. Compilation fails due only to an error on line 7

E. Compilation fails due only to an error on line 8

F. Compilation fails due only to an error on line 11

G. Compilation fails due only to an error on line 13

**8.** Given:

```
 4. public class SpecialOps {
 5.   public static void main(String[] args) {
 6.     String s = "";
 7.     boolean b1 = true;
 8.     boolean b2 = false;
 9.     if((b2 = false) | (21%5) > 2) s += "x";
10.     if(b1 || (b2 = true))         s += "y";
11.     if(b2 == true)                s += "z";
12.     System.out.println(s);
13.   }
14. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. `x` will be included in the output

C. `y` will be included in the output

D. `z` will be included in the output

E. An exception is thrown at runtime

**9.** Given:

```
 3. public class Spock {
 4.   public static void main(String[] args) {
 5.     int mask = 0;
 6.     int count = 0;
 7.     if( ((5<7) || (++count < 10)) | mask++ < 10 )   mask = mask + 1;
 8.     if( (6 > 8) ^ false)                            mask = mask + 10;
 9.     if( !(mask > 1) && ++count > 1)                 mask = mask + 100;
10.     System.out.println(mask + " " + count);
11.   }
12. }
```

Which two are true about the value of mask and the value of count at line 10? (Choose two.)

A. `mask` is 0

B. `mask` is 1

C. `mask` is 2

D. `mask` is 10

E. `mask` is greater than 10

F. `count` is 0

G. `count` is greater than 0

**10.** Given:

```
 3. interface Vessel { }
 4. interface Toy { }
 5. class Boat implements Vessel { }
 6. class Speedboat extends Boat implements Toy { }
 7. public class Tree {
 8.   public static void main(String[] args) {
 9.     String s = "0";
10.     Boat b = new Boat();
11.     Boat b2 = new Speedboat();
12.     Speedboat s2 = new Speedboat();
13.     if((b instanceof Vessel) && (b2 instanceof Toy))  s += "1";
14.     if((s2 instanceof Vessel) && (s2 instanceof Toy)) s += "2";
15.     System.out.println(s);
16.   }
17. }
```

What is the result?

A. `0`

B. `01`

C. `02`

D. `012`

E. Compilation fails

F. An exception is thrown at runtime

**11.** Given:

```
10. boolean b1 = false;
11. boolean b2;
12. int x = 2, y = 5;
13. b1 = 2-12/4 > 5+-7 && b1 != y++>5 == 7%4 > ++x | b1 == true;
14. b2 = (2-12/4 > 5+-7) && (b1 != y++>5) == (7%4 > ++x) | (b1 == true);
15. System.out.println(b1 + " " + b2);
```

What is the result? (This is a tricky one. If you want a hint, go take another look at the operator precedence rant in the chapter.)

A. `true true`

B. `false true`

C. `true false`

D. `false false`

E. Compilation fails

F. An exception is thrown at runtime

## Answers

**1.** ☑ **D** is correct. This is a ternary nested in a ternary. Both ternary expressions are false.

☒ **A, B, C, E**, and **F** are incorrect based on the above. (OCA Objective 3.1 and 3.3)

**2.** ☑ **C** is correct. The == operator tests for reference variable equality, not object equality.

☒ **A, B, D, E**, and **F** are incorrect based on the above. (OCA Objectives 3.1 and 3.2)

**3.** ☑ **E** is correct. Because the short-circuit (`||`) is not used, both operands are evaluated. Since `args[1]` is past the `args` array bounds, an `ArrayIndexOutOfBoundsException` is thrown.

☒ **A, B, C**, and **D** are incorrect based on the above. (OCA Objectives 3.1 and 3.2)

**4.** ☑ **G** is correct. Concatenation runs from left to right, and if either operand is a `String`, the operands are concatenated. If both operands are numbers, they are added together.

☒ **A, B, C, D, E, F, H**, and **I** are incorrect based on the above. (OCA Objective 3.1)

**5.** Answer:

```
class Incr {
  public static void main(String[] args) {
    Integer x = 7;
    int y = 2;

    x *= x;
    y *= y;
    y *= y;
    x -= y;

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

Yeah, we know it's kind of puzzle-y, but you might encounter something like it on the real exam if Oracle reinstates this type of question. (OCA Objective 3.1)

**6.** ☑ **A** is correct. When dividing `ints`, remainders are always rounded down.

☒ **B, C, D**, and **E** are incorrect based on the above. (OCA Objective 3.1)

**7.** ☑ **A** is correct. All this syntax is correct. The for-each iterates through the `enum` using the `values()` method to return an array. An `enum` can be compared using either `equals()` or `==`. An `enum` can be used in a ternary operator's boolean test.

☒ **B**, **C**, **D**, **E**, **F**, and **G** are incorrect based on the above. (OCA Objectives 3.1, 3.2, and 3.3)

**8.** ☑ **C** is correct. Line 9 uses the modulus operator, which returns the remainder of the division, which in this case is `1`. Also, line 9 sets `b2` to `false`, and it doesn't test `b2`'s value. Line 10 would set `b2` to `true`; however, the short-circuit operator keeps the expression `b2 = true` from being executed.

☒ **A, B, D**, and **E** are incorrect based on the above. (OCA Objectives 3.1, and 3.2)

**9.** ☑ **C** and **F** are correct. At line 7 the `||` keeps `count` from being incremented, but the `|` allows `mask` to be incremented. At line 8 the ^ returns true only if exactly one operand is true. At line 9 `mask` is 2 and the `&&` keeps `count` from being incremented.

☒ **A, B, D, E**, and **G** are incorrect based on the above. (OCA Objective 3.1)

**10.** ☑ **D** is correct. First, remember that `instanceof` can look up through multiple levels of an inheritance tree. Also remember that `instanceof` is commonly used before attempting a downcast; so in this case, after line 15, it would be possible to say `Speedboat s3 = (Speedboat)b2;`.

☒ **A, B, C, E**, and **F** are incorrect based on the above. (OCA Objective 3.1)

**11.** ☑ **A** is correct. We're pretty sure you won't encounter anything as horrible as this on the real exam. But if you got this one correct, pat yourself on the back! The way to tackle a problem like this is to evaluate the expression in stages. In this case you might solve it like so:

Stage 1: resolve any use of unary operators

Stage 2: resolve any use of multiplication-related operators

Stage 3: handle addition and subtraction

Stage 4: handle any relationship operators

Stage 5: deal with the equality operators

Stage 6: deal with the logical operators

Stage 7: do the short-circuit operators

Stage 8: finally, do the assignment operators

☒ **B**, **C, D, E**, and **F** are incorrect based on the above. (OCA Objective 3.1)