Operators

Friday, April 5, 2024

10:51 AM

**Assignments**

Assignment of primitives is quite straightforward. Since the primitive holds the actual value and not a reference to an object, when you assign primitives, you copy the contents from one place to another. For example, if you say a = b for primitives, then the contents of b are copied into a. If you then go on to modify a, b is naturally unaffected by this modification.

When you assign objects, however, things change. Whenever you manipulate an object, what you’re manipulating is the reference, so when you assign “from one object to another,” you’re actually copying a reference from one place to another. This means that if you say c = d for objects, you end up with both c and d pointing to the object that, originally, only d pointed to.

This phenomenon is often called **aliasing**, and it’s a fundamental way that Java works with objects.

**Testing object equivalence**

The relational operators == and != also work with all objects, but their meaning often confuses the first-time Java programmer.

*//: operators/Equivalence.java*

*public class Equivalence {*

*public static void main(String[] args) {*

*Integer n1 = new Integer(47);*

*Integer n2 = new Integer(47);*

*System.out.println(n1 == n2);*

*System.out.println(n1 != n2);*

*}*

*}*

*/\* Output:*

*false*

*true \*///:*

The statement **System.out.println(n1 == n2)** will print the result of the boolean comparison within it. Surely the output should be “true” and then “false,” since both Integer objects are the same. But while the contents of the objects are the same, the references are not the same. The operators == and != compare object references, so the output is actually “false” and then “true.”

What if you want to compare the actual contents of an object for equivalence? You must use the special method **equals( )** that exists for all objects (not primitives, which work fine with == and !=).

*public class EqualsMethod {*

*public static void main(String[] args) {*

*Integer n1 = new Integer(47);*

*Integer n2 = new Integer(47);*

*System.out.println(n1.equals(n2));*

*}*

*}*

*/\* Output:*

*true*

*\*///:*

**Logical operators**

Each of the logical operators **AND (&&), OR (||) and NOT (!)** produces a **boolean** value of true or false based on the logical relationship of its arguments.

You can apply AND, OR, or NOT to boolean values only. You can’t use a non-boolean as if it were a boolean in a logical expression as you can in C and C++.

*//! print("i || j is " + (i || j)); Treating an int as a boolean is not legal Java:*

**Short-circuiting**

When dealing with logical operators, you run into a phenomenon called “short-circuiting.” This means that the expression will be evaluated only until the truth or falsehood of the entire expression can be unambiguously determined. As a result, the latter parts of a logical expression might not be evaluated.

**Literals**

A trailing character after a literal value establishes its type. Uppercase or lowercase L means long (however, using a lowercase l is confusing because it can look like the number one). Uppercase or lowercase F means float. Uppercase or lowercase D means double.

Hexadecimal (base 16), which works with all the integral data types, is denoted by a leading 0x or 0X followed by 0-9 or a-f either in uppercase or lowercase. If you try to initialize a variable with a value bigger than it can hold (regardless of the numerical form of the value), the compiler will give you an error message.

Octal (base 8) is denoted by a leading zero in the number and digits from 0-7.

**Exponential notation**

**//: operators/Exponents.java // "e" means "10 to the power."**

**Ternary if-else operator**

The expression is of the form:

***boolean-exp ? value0 : value1***

If boolean-exp evaluates to true, value0 is evaluated, and its result becomes the value produced by the operator. If boolean-exp is false, value1 is evaluated and its result becomes the value produced by the operator.

**String operator + and +=**

*//: operators/StringOperators.java*

*import static net.mindview.util.Print.\*;*

*public class StringOperators {*

*public static void main(String[] args) {*

*int x = 0, y = 1, z = 2;*

*String s = "x, y, z ";*

*print(s + x + y + z);*

*print(x + " " + s); // Converts x to a String*

*s += "(summed) = "; // Concatenation operator*

*print(s + (x + y + z));*

*print("" + x); // Shorthand for Integer.toString()*

*}*

*} /\* Output:*

*x, y, z 012*

*0 x, y, z*

*x, y, z (summed) = 3*

*0*

You see the use of the += operator to append a string to s, and the use of parentheses to control the order of evaluation of the expression so that the ints are actually summed before they are displayed.

Notice the last example in main( ): you will sometimes see an empty String followed by a + and a primitive as a way to perform the conversion without calling the more cumbersome explicit method (Integer.toString( ), in this case).

**Casting operators**

The word cast is used in the sense of “casting into a mold.” Java will automatically change one type of data into another when appropriate. For instance, if you assign an integral value to a floating point variable, the compiler will automatically convert the **int** to a **float**. Casting allows you to make this type conversion explicit, or to force it when it wouldn’t normally happen.

**To perform a cast, put the desired data type inside parentheses to the left of any value.**

*//: operators/Casting.java*

*public class Casting {*

*public static void main(String[] args) {*

*int i = 200;*

*long lng = (long)i;*

*lng = i; // "Widening," so cast not really required*

*long lng2 = (long)200;*

*lng2 = 200; // A "narrowing conversion":*

*i = (int)lng2; // Cast required*

*}*

*} ///:~*

when you perform a so-called narrowing conversion (that is, when you go from a data type that can hold more information to one that doesn’t hold as much), you run the risk of losing information. Here the compiler forces you make the cast explicit.

Java allows you to cast any primitive type to any other primitive type, except for boolean, which doesn’t allow any casting at all.

**Java has no "sizeof"**

Java does not need a sizeof( ) operator for this purpose, because all the data types are the same size on all machines.