5-operators-and-statements

Contents

- Operators
  - Arithmetic
  - Bitwise
  - Caveats
  - Representation
- Statements
  - If
  - While
  - Switch
  - Jumps

Operators

Arithmetic

Operator	Result
+	Addition (also unary plus)
-	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
- =	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
–	Decrement

Operator	Result

Bitwise

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
»	Shift right
»>	Shift right zero fill
«	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
»=	Shift right assignment
»>=	Shift right zero fill assignment
«=	Shift left assignment

Caveats

Besides the usual Arithmetic and Bitwise operators which work pretty much as one would expect from other languages, there are two special types of operators which java exposes which are `>>>` and `<<<`. The regular shift operators in java work a little bit different than what one might expect, since java has no unsigned type, the usual shift operators in java `>>` and `<<` actually account for the sign part of the number, making sure that each time you use `<<` or `>>` the sign bit of the number is kept. As we all know shifting right or left a number by one means we effectively divide or multiply it by 2.

```
00100011 35
>> 2
00001000 8 // the two least significant bits are lost, we have 2^3 + 0^2 +
   0^1 + 0^0 = 8
```

```
11111000 -8
>> 1
11111100 -4 // the -8 is perfect power of two, shifting right divides it
   perfectly by 2
```

However note that the top bits are filled up with the sign bit of the number, 1 for negative, 0 for positive, this is done to keep the 2s complement representation of the number correct. To actually avoid this java introduced the `>>>` and `<<<` operators, which are meant to perform shifts on signed numbers without filling up, or touching the higher order bits. These triple shift operators are what is available in most other languages by default as shift operations

Representation

`All of the integer types (except char)are signed integers`. This means that they can represent negative values as well as positive ones. Java uses an encoding known as two's complement, which means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result. For example, -42 is represented by inverting all of the bits in 42, or 00101010,

which yields 11010101, then adding 1, which results in 11010110, or -42. To decode a negative number, first invert all of the bits, then add 1.

For example, -42, or 11010110 inverted, yields 00101001, or 41, so when you add 1 you get 42. The reason Java (and most other computer languages) uses two's complement is easy to see when you consider the issue of zero crossing. Assuming a byte value, zero is represented by 00000000. In one's complement, simply inverting all of the bits creates 11111111, which creates negative zero. The trouble is that negative zero is invalid in integer math. This problem is solved by using two's complement to represent negative values. When using two's complement, 1 is added to the complement, producing 100000000. This produces a 1 bit too far to the left to fit back into the byte value, resulting in the desired behavior, where -0 is the same as 0, and 11111111 is the encoding for -1. Although we used a byte value in the preceding example, the same basic principle applies to all of Java's integer types.

Statements

If

It can be used to route program execution through two different paths. Nothing really unusual about it compared to other languages

While

Similarly to the if statement there is nothing special about the `while` or `do while`, compared to other languages.

Switch

The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. The switch statement as of very recently did not take anything else other than a constant or literal value to switch on, unlike if statements, however that has changed in recent java versions where the switch statement has improved dramatically

```
switch (expression) {
    case value1:
        // statement sequence
        break;
    case value2:
        // statement sequence
        break;
    .
    .
    .
    case valueN :
        // statement sequence
        break;
    default:
        // default statement sequence
}
```

`Note that the break is optional, if one does not include a break after the case, the flow of the program will continue, meaning that the default statement will also be invoked`. This is an often done mistake which is very seldom intentional

A switch statement is usually more efficient than a set of nested ifs. That point is particularly interesting because it gives insight into how the Java compiler works. When it compiles a switch statement, the Java compiler will inspect each of the case constants and create a "jump table" that it will use for selecting the path of execution depending on the value of the expression.

Jumps

Break In Java, the break statement has three uses. First, as you have seen, it terminates a statement sequence in a switch statement. Second, it can be used to exit a loop. Third, it can be used as a "civilized" form of goto.

```
boolean t = true;
first: {
    second: {
        third: {
            System.out.println("Before the break.");
            if(t) break second; // break out of second block
            System.out.println("This won't execute");
        }
    System.out.println("This won't execute");
    }
System.out.println("This is after second block.");
}
```

The most notable feature of break is that it can use labels, similarly to how labels work in other languages like C or assembly, we can mark a block of code with a label, which is enclosed in square brackets, which can then be used from inside nested loops, such as the example above, which will break out of the two loops and directly exit the outer label block when the conditional is met

Continue This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an action. In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop.

```
outer: for (int i=0; i<10; i++) {
    for(int j=0; j<10; j++) {
        if(j > i) {
            System.out.println();
            continue outer;
        }
        System.out.print(" " + j);
    }
}
System.out.println();
```

Similarly to break continue can also use labels, however instead of exiting early what it is going to do is return control to an earlier statement instead, in the example above, when j becomes greater than i, the control flow is returned back to the outer label loop, continuing the increment of i in the outer for loop, meaning that at the end, the code above will print a step like structure

```
0
0 1
0 1 2
0 1 2 3
```

```
0 1 2 3 4
.........
```

Return The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement. The general workings of return are not much different than those in other languages, there are no hidden special cases, we can either return without a value, `void`, or return a specific return type from the control block