Chapter 2 - Operators and Statements

Bob

May 12, 2022

1 Java Operators

Java Operator : A Java operator is a special symbol that can be applied to a set of variables, values, or literals - referred to as operands - and that returns a result.

Types of Java Operators : There are three flavors of operators available in Java:

- unary
- binary
- ternary

1.1 Order of Operation

Unless overridden with parentheses, Java operators follow *order of operation*, listed in this table by decreasing order of *operator precedence*. If two operators have the same level of precedence, then Java guarantees left-to-right evaluation.

Operator	Symbols and examples
Post-unary operators	expression++, expression–
Pre-unary operators	++expression, –expression
Other unary operators	,+,-,!
Multiplication/Division/Modulus	*,/,%
Addition/Subtraction	+,-
Shift operators	¡¡,¿¿,¿¿¿
Relational operators	¡,¿,¡=,¿=, instanceof
Equal to/not equal to	==, !=
Logical operators	&,ˆ,—
Short-circuit logical operators	&&,——
Ternary operators	boolean expression ? expression1 : expression2
Assignment operators	=,+=,-=,*=,/=,%=,&=, ˆ=, —=, ¡¡=, ¿¿=, ¿¿¿=

1.2 Binary Operators

1.2.1 Arithmetic Operators

All of the arithmetic operators may be applied to any Java primitives, except *boolean* and *String*. Furthermore, only the addition operators + and += may be applied to String values, whcih results in String concatenation.

1.2.2 Numeric Promotion

1. If two values have different data types, Java will automatically promote one of the values to the larger of the two data types.
2. If one of the values is integral and the other is floating-point, Java will automatically promote the integral value to the floating-point value's data type.
3. Smaller data types, namely *byte*, *short* and *char*, are first promoted to *int* any time they're used with a Java binary arithmetic operator, even if neither of the operands is *int*.
4. After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.

Note: Unary operators are excluded from this rule.

1.3 Unary Operators

Unary Operator : A unary operator is onen that requires exactly one operand, or variable, to function.

Unary Operator	Description
+	Indicates a number is positive, although numbers are assumed to be positive in Java
-	Indicates a literal number is negative or negate an expression
++	Increments a value by 1
–	Decrements a value by 1
!	Inverts a Boolean's logical value

Negation Operator (-) : Reverses the sign of a numeric expression. **Logical Complement Operator (!)** : Flips the value of a boolean expression.

Note: Unlike some other programming languages, in Java *1* and *true* are not related in any way, just as *0* and *false* are not related.

Increment and Decrement Operators (++, –) : Increment and decrement operators, ++ and –, respectively, can be appied to numeric operands and have the higher order or precedence, as compared to binary operators. In other words, they often get applied first to an expression.

1.4 Additional Binary Operators

1.4.1 Assignment Operator

An *assignment operator* is a binary operator that modifies, or assigns, the variable on the left-hand side of the operator, with the result of the value on the right-hand side of the equation.

Note: **Java will automatically promote from smaller to larger data types**, but it will throw a compiler exception if it detects you are trying to convert from larger to smaller data types.

1.4.2 Casting Primitive Values

Casting primitives is required any time you are going from a larger numerical data type to a smaller numerical data type, or converting from a floating-point number to an integral value. By performing this **explicit cast** of a larger value into a smaller dat type, you are instructing the compiler to ignore its default behavior. You are telling the compiler that you have taken additional steps to prevent overflow or underflow.

Overflow and Underflow : *Overflow* is when a number is so large that it will no longer fit within the data type, so the system "wraps around" to the next lowest value and counts up from there. There's also an analogous *underflow*, when the number is too low to fit in the data type.

1.4.3 Compound Assignment Operators

Besides the simple assignment operator, =, there are also numerous *compound assignment operators*. Complex operators are really just glorified forms of the simple assignment operator, with a built-in arithmetic or logical operation that applies the left- and right-hand sides of the statement and stores the resulting value in a variable in the left-hand side of the statement. The lef-hand side of the compound operator can only be applied to a variable that is already defined and cannot be used to declare a new variable.

1.4.4 Relational Operators

Relational operators compare two expressions and return a *boolean* value. If the two numeric operands are not of the same data type, the smaller one is promoted in the manner as previously discussed.

Operator	Description
¡	Strictly less than
¡=	Less than or equal to
¿	Strictly greater than
¿=	Greater than or equal to

1.4.5 Logical Operators (&, — and ^)

The *logical operators*, (&), (—), and (^), may be applied to both *numeric* and *boolean* data types, they're referred to as *logical operators*. Alternatively, when they're applied to *boolean* data types, they're referred to as *bitwise operators*, as they perform bitwise comparisons of the bits that compose the number.

1.4.6 Truth Tables

x & y (AND)

—	y = true	y = false
x = true	true	false
x = false	false	false

x — y (INCLUSIVE OR)

—	y = true	y = false
x = true	true	true
x = false	true	false

x ŷ (EXCLUSIVE OR)

—	y = true	y = false
x = true	false	true
x = false	true	false

1.5 Shortcut Operators (&& and ——)

The short-circuit operators are nearly identical to the logical operators, & and —, respectively, except that **the right-hand side of the expression may never be evaluated if the final result can be determined by the left-hand side of the expression.**

1.6 Equality Operators

Determining equality in Java can be a nontrivial endeavor as ther's a semantic difference between "two objects are the same" and "two objects are equivalent". It is further complicated by the fact that for numeric and boolean primitives, there is no such distinction.

The *equals* operator == and *not equals* operator *!=*. Like the relational operators, they compare two operands and return a boolean value about whether the expressions or values are equal, or not equal, respectively. The equality operators are used in one of three scenarios:

1. Comparing two numeric primitive types. If the numeric values are of different data types, the values are automatically promoted.

2. Comparing two *boolean* values.

3. Comparing two objects, including *null* and *String* values.

Note: Pay close attention to the data types when you see an equality operator on the exam. (= vs ==)

For object comparison, the equality operator is applied to the references to the objects, not the objects they point to. Two references are equal if and only

if they point to the same object, or both point to *null*. (objectA == objectB vs objectA.equals(objectB).

2 Java Statements

Java Statement : is a complete unit of execution in Java, terminated with a semicolon (;).

Control Flow Statement : Break up the flow of execution byusing decision making, looping, and branching, allowing the application to selectively execute particular segments of code.

Block of Code : is a group of zero or more statements between balanced braces (), and can be used anywhere a single statement is allowed.

Decision making control structures:

- if-then
- if-then-else
- switch

Repetition control structures:

- for
- for-each
- while
- do-while

2.1 The if-then statement

The *if-then* statement execute a block of code under certain circumstances, by allowing our application to execute a particular block of code if and only if a *boolean* expression evalueates to true at runtime.

```
if(booleanExpression) {
   // Branch if true
}
```

For readability, it is considered good coding practice to put blocks around the execution component of *if-then* statements, as well as many other control flow statements, although it is not required.

Note: In Java tabs are just whitespace and are not evaluated as part of the execution.

2.2 The if-then-else statemnt

```
if(booleanExpression) {
   // Branch if true
} else {
   // Branch if false
}
```

Note: In Java *0* and *1* are not considered *boolean* values.

2.2.1 Ternary Operator

The conditional operator **? :** otherwise known as *ternary operator*, is the only operator that takes three operands and is of the form:

```
booleanExpression ? expression1 : expression2
```

The first operand must be a *boolean* expression, and the second and third can be any expression that returns a value. The ternary operation is really a condensed form of an *if-then-else* statement that returns a value. These 2 snippets of code are identical: With **if-then-else**:

```
int y = 10;
int x;
if(y > 5) {
   x = 2 * y;
} else {
   x = 3 * y;
}
```

With **Ternary Operator**:

```
int y = 10;
int x = (y > 5) ? (2 * y) : (3 * y);
```

Note: add parentheses around the expressions in ternary operations is helpful for readability, although not required. There is no requirement that second and third expressions in ternary operations have the same data types, although it may come into play when combined with the assignment operator. **Ternary Expression Evaluation**: Only one of the right-hand expressions of the ternary operator will be evaluated at runtime. In a manner similar to the short-circuit operators, if one of the two right-hand expressions in a ternary operator performs a side effect, then it may not be applied at runtime.

```
int y = 1;
int z = 1;
final int i = y<10 ? y++ : z++;
System.out.println(y + "," + z); // Outputs 2,1
```

2.3 The switch statement

A *switch* statement is a complex decision-making structure in which a single value is evaluated and flow is redirected to the first matching branch, known as a *case* statement. If no such case statement is found that matches the value, an optional *default* statement will be called. If no such *default* option is available, the entire switch statement will be skipped.

2.3.1 Supported Data Types

A *switch* statement has a target variable that is not evaluated until runtime. Prior to Java 5.0, this variable could only be *int* values or those values that could be promoted to *int*, specifically *byte*, *short*, or *int*. In Java 7, *switch* statements were further updated to allow matching on *String* values. Finally, the switch statement also supports any of the primitve numeric wrapper classes, such as *Byte*, *Short*, *Character*, or *Integer*.

```
switch(variableToTest) {
   case constantExpression1:
      // Branch for case1;
      break;

   case constantExpression2:
      // Branch for case2;
      break;

   ...

   default:
      // Branch for default;
}
```

Data Types supported by *switch* statements include the following:

- *byte* and *Byte*
- *short* and *Short*
- *char* and *Character*
- *int* and *Integer*
- *String*
- *enum* values

Note that *boolean* and *long*, and their associated wrapper classes, are not supported by *switch* statements. **RULE SWITCH: No boolean, No long**

2.3.2 Compile-time Constant Values

The values in each *case* statement must be compile-time constant values of the same data type as the *switch* value. This means that you can use only:

- literals
- *enum* constants
- *final* constant variables of the same data type. By final constant, we mean that the variable must be marked with the *final* modifier and initialized with a literal value in the same expression in which it is declared.

Break Statement: There is a *break* statement at the end of each *case* and *default* section. Break statements terminate the *switch* statement and return flow control to the enclosing statement. If you leave out the *break* statement, flow will continue to the next proceeding *case* or *default* block automatically. Note: There is no requirement that the *case* or *default* statements be in a particular order. Note: *The exam creators are fond of switch examples that are missing break statements.* The data type for *case* statements must all match the data type of the *switch* variable.

Exit switch:

- break
- return

2.4 The while statement

A repetition control structure, which we refer to as a *loop*, executes a statement of code multiple times in succession.

```
while{booleanExpression) {
   // Body
}
```

During execution, the *boolean* expression is evaluated before each iteration of the loop and exits if the evaluation returns false.

2.5 The do-while statement

Unlike a *while* loop, though, a *do-while* loop guarantees that the statement or block will be executed at least once. Note: Any *while* loop can be converted to a *do-while* loop and viceversa. Java recommends you sue a *while* loop when a loop might not be executed at all an a *do-while* loop when the llop is executed at least once.

2.6 The for statement

A basic *for loop* has the same conditional *boolean* expression and statement, or block of statements, as the other loops you have seen, as well as two new sections: an *initialization block* and an *update* statement.

```
for(initialization; booleanExpression; updateStatement) {
   // Body
}
```

Note that each section is separated by a semicolon. The initialization and update sections may contain multiple statements, separated by commas. **Infinite for loop**:

```
for( ; ; ) {
   // Body
}
```

The components of the *for* loop are each optional. The semicolons separating the three sections are required. The variables in the initialization block must all be of the same type.

2.7 The for-each statement

Starting with Java 5.0, Java developers have had a new type of enhanced for loop at their disposal, one specifically designed for iterating over arrays and *Collection* objects.

```
for(datatype instance : collection) {
   // Body
}
```

The right-hand side of the *for-each* loop statement must be a built-in Java array or an object whose class implements *java.lang.Iterable*, which includes most of the Java *Collections* framework. The left-hand side of the *for-each* loop must include a declaration for an instance of a variable, whose type matches the type of a member of the array or collection in the right-hand side of the statement.

Collections for OCA Exam: The only member of the *Collections* framework that you need to be aware of are:

- *List*
- *ArrayList*

Note: When you see a *for-each* loop on the exam, make sure the right-hand side is an array or *Iterable* object and the left-hand side has a matching type.

Problem for-each: While the *for-each* statement is convenient for working with lists in many cases, it does hide access to the loop iterator variable.

As a developer you can always revert to a standard *for* loop if you need fine-grain control.

3 Advanced Flow Control

Advanced flow controls:

- Nested Loops
- Optional Labels
- The *break* statement
- The *continue* statement

3.1 Nested Loops

Loops can contain other loops.

3.2 Optional Labels

A *label* is an optional pointer to the head of a statement that allows the application flow to jump to it or break from it. It is a single word that is proceeded by a colon (:). Example:

```
int[][] array = {{1,2},{3,4}};
OUTER_LOOP: for(int[] simpleArray : array) {
   INNER_LOOP: for(int i = 0; i<simpleArray.length; i++) {
      System.out.print(simpleArray[i]);
   }
   System.out.println();
}
```

Optional labels are often only used in loop structures. It is rarely considered a good coding practice. For readability, they are commnly expressed in uppercase, with underscores between words, to distinguish them from regular variables.

3.3 The *break* statement

A *break* statement transfers the flow of control out to the enclosing statement.

```
OPTIONAL_LABEL: while(booleanExpression) {
   // Body

   // Somewhere in loop
   break OPTIONAL_LABEL;
}
```

Notice that the *break* statement can take an optional label parameter. Without a label parameter, the *break* statement will terminate the nearest inner loop it is currently in the process of executing. The optional label parameter allows us to break out of a higher level outer loop.

3.4 The *continue* statement

The *continue* statement causes flow to finish the execution of the current loop.

```
OPTIONAL_LABEL: while(booleanExpression) {
   // Body

   // Somewhere in loop
   continue OPTIONAL_LABEL;
}
```

The syntax of the *continue* statement mirrors that of the *break* statement. While the *break* statement transfers control to the enclosing statement, the *continue* statement transfers control to the *boolean* expression that determines if the loop should continue. In other words, it ends the current iteration of the loop.

—	Allows optional labels	Allows unlabeled break	Allows continue statement
if	Yes	No	No
while	Yes	Yes	Yes
do-while	Yes	Yes	Yes
for	Yes	Yes	Yes
switch	Yes	Yes	No