# Chapter 2: Operators and Statements.

Operator: a special symbol that can be applied to a set of variables, values or literals – referred as operands – and that returns a result. Operators can be: unary, binary and ternary. Unless overwritten by parenthesis, Java follows order of operation, then Java guarantees left-to-right evaluation. Order of operator precedence:

| | |
|---|---|
| Post-unary operators | *expression++, expression--* |
| Pre-unary operators | *++expression, --expression* |
| Other unary operators | *~, +, -, !* |
| Multiplication / Division / Modulus | **, /, %* |
| Addition / Substraction | *+, -* |
| Shift operators | *<<, >>, >>>* |
| Relational operators | *==, !=* |
| Logical operators | *&, ^, \|* |
| Short-circuit logical operators | *&&, \|\|* |
| Ternary operators | *boolean expression ? expression1 : expression2* |
| Assignment operators | *=, +=, -=, *=, /=, %=, &=, ^=, \|=, <<=, >>=, >>>=* |

Arithmetic Operators.

Addition (+), subtraction (-), multiplication (*), division (/) and modulus (%). They include ++ and -- – unary operators. *, /, % have a higher precedence than + and -. They can be applied to all primitives type except for boolean and String. When +, += is applied to String, it's concatenation.

Numeric Promotion.

Rules:

- If two values have different data types, Java will automatically promote one of the values to the larger of the two.
  *int x = 1; long y = 33; x * y data type will be long.*
- 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.
  *double x = 39.21; float y = 2.1f; x + y data type will be double.*
- Smaller data types (byte, short, char) are first promoted to int any time they're used with a Java binary arithmetic operator, even if neither of the operand is int.
  *short x = 10; short y =3; x / y data type will be int.*
- 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.
  *short x = 14; float y = 13; double z = 30;*
  *x * y / z data type result will be double. X is first promoted to int and then to float to be multiplied with y. The result will be promoted to double to be divided by z.*

OCA Chapter 2 Notes

Unary Operators.
Only requires one operand / variable to function.

| | |
|---|---|
| ***+*** | *Positive number* |
| ***-*** | *Negative number* |
| ***++*** | *Increments value by 1* |
| ***- -*** | *Decrements value by 1* |
| ***!*** | *Inverts a Boolean's logic value* |

Logical Complement: !, flips the logical value of a Boolean.
Cannot be applied to a numeric variables.
Negation Operator: -, reverses the sign in a numeric expression.
Increment and Decrement Operators: ++, - -, to be applied to numeric operands and have higher precedence or order, as compared to binary operators. These can be:

- Pre-decrement: --variable
- Pre-increment: ++variable
- Post-decrement: variable- -
- Post-increment: variable++

In the scenario in which variable is 5, all the operations above produce a different output. Pre operators will increment or decrement the value first, then return the result. Post operators will return the result first, then increment or decrement the value.

Binary Operators.

- Assignment operators. Modifies the value of the variable with the value on the right-hand side.
  E.g.: *int x = 1;*
  Java will automatically promote from smaller to larger data type. Error will be thrown if it's attempted to assign larger to smaller data type.
  E.g.:
  *int x = 1.0;*
  *short y = 1921222;*
  *int z = 9f;*
  *long t = 192301398193810323;*

  Casting primate values. Examples above can be fixed by casting the large data types to smaller ones.
  E.g.:
  *int x = (int)1.0;*
  *short y = (short)1921222; // value overflows to 20,678*
  *int z = (int)9f;*
  *long t = 192301398193810323L;*

  *Example of underflow: System.out.println(2147483647+1); // prints -2147483648*

  *Another casting example to override JVM behavior:*

*short x = 10; short y = 3;*
*short z = (short)(x * y);*

- Compound Assignment Operators. Includes +=, -=, /=, *=.
  E.g.:
  *int x = 2, z = 3;*
  *x = x * z; // simple assignment.*
  *x *= z; // Compound assignment.*

  Compound operators can apply a cast of larger values to small automatically.
  E.g.:
  *long x = 10; int y = 5;*
  *y *= x;*

  Another valid and not common compound assignment:
  *long x = 5;*
  *long y = (x=3);*
  *System.out.println(x);*
  *System.out.println(y);*
  Both output 3.

- Rational Operators. Compares two expressions and returns a Boolean value. This applies only to numeric primitive data types. As other operators, the smaller operand will be promoted the larger data type.
  E.g:
  *int x = 10, y = 20, z = 10;*
  *System.out.println(x < y); // true*
  *System.out.println(x <= y); // true*
  *System.out.println(x >= z); // true*
  *System.out.println(x > z); // false*

- Logical Operators. These &, | and ^ are applied to numeric and Boolean primitive data types. When applied to Boolean is referred as logical operators. When applied to numeric values is referred as bitwise operators, because they perform bitwise comparisons of the bits that compose the number.

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 ^ y (Exclusive OR)

| | y = true | y = false |
|---|---|---|
| x = true | False | True |
| x = false | True | False |

Short-circuit operators (&&) and (||) are very similar to the previous ones, expect that the right hand of the expression may never be evaluated.
E.g.: *boolean x = true || (y < 4);*

Other examples:

```
If(x != null %% x.getValue() < 5) {
        // Do something
}
```

- Equality Operators.
  Equals ( == ) and not Equals ( != ) operators are used in the following scenarios:
  1. Comparing two numeric primitives types. If data types are different, the smaller will be promoted to the larger type.
  2. Comparing two boolean values.
  3. Comparing two objects, including null and String values.

## Java Statements.

A statement in Java is a complete unit of execution, terminated with a semicolon ( ; ). Control flow statements break up the flow of executing by using decision making, looping and branching, allowing the application to selectively execute specific segments of code.

The if-then Statement.
We only want to execute a block of code under certain circumstances. This statement allows the code to execute if the boolean expression evaluates to true at runtime.

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

The if-then-else Statement.
We want to execute different blocks of code when our condition may have different outputs.

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

---

Ternary Operator. It's the only operator in java that takes 3 operands and is of the form:
`booleanExpression ? expression1 : expression2`
The first one must be a boolean expression, and the second and third can be any expression that returns a value.

---

OCA Chapter 2 Notes

The Switch Statement.

It's 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 ***case*** is found that matches the value, an optional ***default*** statement will be called. If no ***default*** option is available, the whole block will be skipped.

Supported data types: byte and Byte (wrapper class), short and Short (w. class), char and Character (w. class), int and Integer (w. class), String and enum values. Does not support boolean and long.

```
switch(variableToTest) {
      case constantExpression1:
            // Branch for case1;
            break;
      case constantExpression2:
            // Branch for case2;
            break;
      default:
            // branch for default;
      }
```

Compile-time Constant values.

The values in each case statement must be compile-time contact values of the same data type as the switch value. This means only literals, enum constants or final constant variables of the same data type can be used.

The While Statement.

A repetition control structure / loop. It's the simplest loop in Java.

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

The body will execute as long as the boolean expression is true.

The do-while Statement.

Do-while loop is guaranteed that the body will execute at least once, since the boolean expression is evaluated at the end of the block:

```
do { // Body } while(booleanExpression);
```

The Basic for Statement.

This contains an initialization block, a boolean expression block as the previous loops, and an update statement block.

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

Variables declared in the initialization block has a scope limited to the loop.

The boolean expression is evaluated in ever iteration before the loop executes.

5 variations of this for statement to familiarize with:

1. Creating an infinite loop.
   ```
   for ( ; ; ) { System.out.println("Hello World");
   ```
2. Adding multiple terms to the for statement.
   ```
   int x = 0;
   for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {
   System.out.print(y + " "); }
   ```
3. Redeclaring a variable in the initialization block.
   ```
   int x = 0;
   for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) { // Does not compile
   System.out.print(x + " "); }
   ```
4. Using incompatible data types in the initialization block.
   ```
   for (long y = 0, int x = 4; x < 5 && y < 10; x++, y++) { // Does not
   compile
   System.out.print(x + " "); }
   ```
5. Using loop variables outside the loop.
   ```
   for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {
   System.out.print(y + " "); }
   System.out.print(x); // Does not compile
   ```

The For-each Statement / Enhanced loop.

It was specifically design for iterating over arrays and Collections objects. It's composed of an initialization section and an object to be iterated over.

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

The right-hand side of the loop 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 must include a declaration of an instance of a variable, whose type matched the type of a member of the array or collection in the right-hand side. For each iteration, the instance will have a different value from the collection in the right-hand side.

Nested Loops.

Loops can be nested, one inside of the other. And optional labels can be applied in order to make the code more understandable.

```
OUTER_LOOP: for(datatype instance : collection) {
INNER_LOOP:       for(initialization; booleanExpression; updateStatement) {
      // Body
}}

while(booleanExpression) {
      do { // Body } while(booleanExpression);
// Body }
```

The Break Statement.

It transfers the flow of control out to the enclosing statement. It applies to all kinds of loop seen previously.

```
optionalLabel : while (booleanExpression) {
```

```
        // Body
        break optionalLabel;
}
```

Without the label, break will terminate the nearest inner loop. The optional label allows us to break out of a higher-level outer loop.

The Continue Statement.

It causes the flow to finish the execution of the current loop.

```
optionalLabel : while (booleanExpression) {
        // Body
        continue optionalLabel;
}
```

The continue statement transfers the control to the boolean expression that determines if the loop should continue. It only ends the current iteration of the loop. As the break statement, it is applied to the nearest inner loop, unless optional label is used.

| | Allows optional label | 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 |