# Subsection 1: Use Java operators and understand operator precedence

## What is a Java operator?

A Java operator is a symbol that when applied to a set of values, variables or literals returns a result. Look at the following example.

5 + 3

The operator in the above arithmetic expression is the plus sign (+) and it operates on the values 5 and 3 to produce the result 7.

There are three types of operators: unary, binary and ternary. As their name suggest these operators apply to one, two or three operands. The above example uses the plus sign which is a binary operator because it requires two operands, the 5 and the 3. Later on we will look deeper into these three operator flavours.

## Operator precedence

The order in which an expression is evaluated is very important in determining the correct outcome. Take the following example.

5 + 2 \* 10

What is the result of this expression? (Note that the asterisk (\*) denotes multiplication.) Is it 70? (5 plus 2 then multiplied by 10) or is it 25? (2 multiplied by 10 then add 5)? The answer is 25 and we arrive at this value by applying the rules of operator precedence. These rules tell us which operator to evaluate first.

The following table shows a list of Java operators from those with the highest precedence shown at the top of the list and reducing in precedence order to the lowest at the bottom. It also shows weather the operator is left, right or non-associative.

Table 1: Operator precedence

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Symbol | Associativity | Description |
|  | () [] . | left | Parentheses, Array Subscript, Member selection |
| Unary | x++ x-- ~x | left | postfix increment/decrement, bitwise negation |
| Unary | ++x --x +x -x !x | right | prefix increment/decrement, unary positive/negative, logical negation |
|  | (X) | right | typecasting |
|  | new X | non | object creation |
| Arithmetic | \* / % | left | multiplication, division, modulus |
| Arithmetic and concatenation | x+y x-y x+"x" | left | addition, subtraction, string concatenation |
| Shift | << >> >>> | left | Bitwise shift |
| Relational | < <= > >=  instanceof | left | comparison  runtime type compatibility |
| Equality | == != | left | equality and inequality |
| Logical | & | left | bitwise AND |
| Logical | ^ | left | bitwise XOR |
| Logical | | | left | bitwise OR |
| Short-circuit Logical | && | left | logical AND |
| Short-circuit Logical | || | left | logical OR |
| Ternary | x ? y : z | right | ternary (conditional) |
| Assignment | =  += -= \*= /= %=  <<= >>= >>>=  &= ^= |= | right | assignment and compound assignment |

The order of precedence is applied from the top of the table to the bottom. Let’s examine some examples using operators from this table. Consider the following equation.

10 + (20-5) \* 5

In this expression the value of the expression in the brackets is evaluated first, which gives a new expression 10 + 15 \* 5, then the multiplication of 15 with 5 giving 75, then then addition of 10 which results in 85.

So the order is:

1. (20 – 5) = 15
2. 15 \* 5 = 75
3. 10 + 75 = 85

You will notice that the parentheses are evaluated first, the multiplication and then the addition. We have applied the order of evaluation stated in the operator precedence table.

Let’s look at one more example:

40 / 4 + 1 \* ((100 / 10) \* 2)

Here we have nested parentheses and in cases of nested parentheses the inner-most expression is evaluated first. The order of evaluation is as follows:

1. 100 / 10 = 10
2. 10 \* 2 = 20
3. 20 \* 1 = 20
4. 40 / 4 = 10
5. 10 + 20 = 30

Let’s look at examples that involve Strings that highlight operator precedence. What is the value of the variable *message* after the evaluation of the expression?

String message = "hello" + 2 \* 10;

The result is *hello20.* This is because the multiplication is evaluated first and a String is constructed from the result and concatenation to the *“hello”* String. Refer back to the table and you can see that multiplication has a higher precedence than concatenation. Consider another example:

String message = "hello" + 2 + 10;

The result is *hello210* because the concatenation of the String *“hello”* to the number 2 is done first, then number 10 is concatenation to the String *“hello2”*. This is an example of how associativity determines the order of evaluation when the operators are of equal precedence. Refer back to the table and review the associativity of the concatenation operators and you will see that it is left associative, which means the expression is evaluated from left to right.

The following an example that involves a method call:

int j = 10;  
int i = doIt() / (j = 2);

The *doIt()* method has the highest precedence and will be executed first. In this case, the *doIt()* method will throw an exception so the code *j = 2* will not be evaluated. If the exception is caught, the value of *j* will continue to be 10. We will look at exception handling later on in this course in **Lesson 2 Section 3: Understand Exception Handling**.

## Associativity

An operator can be left-associative, right-associative, or non-associative. So far we have seen expressions evaluated from left to right. This is correct for arithmetic operators. However this is not true for all operators as can be seen from the table above. The **associativity** column identifies the type of associativity that applies to each operator.

The way associativity applies is as follows:

**Left-associative** operators of the same precedence are evaluated in order from left to right. For example, addition and subtraction have the same precedence and they are left-associative. In the expression 10-4+2, the subtraction is done first because it is to the left of the addition, outputting a value of 8.

**Right-associative** operators of the same precedence are evaluated in order from right to left. For example, assignment is right-associative. Consider the following code snippet:

int a = 3;  
int b = 4;  
a = b = 5;

After the code has been evaluated, both *a* and *b* evaluate to 5 because the assignments are evaluated from right to left.

A **non-associative** operator cannot be combined with other operators of the same precedence.

## Binary Arithmetic Operators

You will use binary operators more frequently that most others and you will combine them together to produce complex expressions and therefore operator precedence becomes very important.

### Assignment Operators

The simplest assignment operator is = and we have met this operator already when assigning values to variables. The following expression int x = 10 + 5 assigns the result of 15 to the variable x. This is referred to as a simple assignment operator.

### Compound Assignment Operators

Java also offers compound assignment operators that combine simple arithmetic operations with assignment. It’s a short-hand way to perform an arithmetic operation and assign the result.

Assignment Result

The result of an assignment is the result of the expression. Consider the following example:

long x = 20;  
System.*out*.println(x = x \* 2);

The output to the console window is 40, which is the result of the expression. Now consider this example:

System.*out*.println(a = (b = 2));

Variable b is assigned the integer 2 and the result of the assignment is 2, which in turn is assigned to variable a, and the result of this assignment is 2, so the output is 2.

Watch out for questions that test this knowledge. Even though it is not common code in practice it does not mean that you will not meet tricky questions on this topic.

### Numeric promotion rules

If you remember from **lesson 1 section 2** we talked about casting variables to larger and smaller data types. This was done either implicitly or explicitly and you learnt the rules associated with this Java language feature.

Related to this feature is the way that data types are dealt with when applying operators.

The rules that are applied are as follows and are applied in the following given order:

1. When two values of different data types are encountered Java will promote the smaller value to the larger value automatically.
2. An integral value will be promoted to a floating point type if one of the values is a floating-point data type.
3. Byte, short and char data types are always promoted to an integer even if none of the expressions types are integers. This rule only applies to binary operators.
4. The resulting data type will be the data type that types are promoted to.

Let’s have a look at some examples of these rules in play:

**Rule 1:**

int a = 10;  
long b = 200;

What is the resulting data type of a + b? Applying the first rule the integer is promoted to a long data type.

**Rule 2:**

int a = 3;  
double b = 1.69;

What is the resulting data type of a \* b? Applying the second rule the integer is promoted to a double data type.

**Rule 3 and 4:**

byte a = 3;  
byte b = 1;

What is the resulting data type of a - b? Applying the third rule both bytes are promoted to integer data types and then applying the fourth rule to get the resulting data type of an integer.

## Unary Operators

Unary operators are sign indicator operators which are positive (+) and negative (-), incremental (++) and decremental (--) and logical negation (!). As you will notice from the table the postfix incremental and decremented operators (x++ x--) have greater precedence that the other operators (+ - ! ++x --x). By definition unary operators require only one operand to operate on.

### Increment and decrement operators

Incremental and decremental operators increase the operand by one and come in two flavours, prefix and postfix.

Prefix increment and decrement operators apply before the operand is used in an expression and postfix increment and decrement operators apply after the operand is used in an expression. Let’s see an example to clarify what this means in practice.

int visitors = 100;  
System.*out*.println(++visitors);

In this example we are applying a prefix incremental operator to the variable *visitors*. The result of this operation is to increase the value of the variable *visitors* by one. The output is 101. The reverse operations would use the prefix decremental operator and would reduce the value of *visitors* by one.

Let’s look at an example of a postfix

1. int visitors = 100;  
2. System.*out*.println(visitors++);  
3. System.*out*.println(visitors);

The output might be a little surprising and is 100 then 101. The postfix increment operator adds one to the value after the operand is used, in this case being used means that the value is printed to the console before being increased by one. This can be seen in the output of line 3 which is 101. A postfix decrement operator works by reducing the operand by one after it has been used.

Let’s have a look at how to use these operators in an expression. Consider the following example.

int x = 10;  
int y = 10;  
int z = ++x + y--;  
System.*out*.println(z);

Can you figure out the value output to the console?

The prefix increment operator is applied to the variable *x* and a postfix decrement is applied to *y*. This expression is evaluated from left to right so firstly the value of *x* is increased by one to 11 then the value of *y*, which is10, is used and added to 11, then *y* is decreased by one, this results in an output of 21. After expression is evaluated the value of *y* will be 9.

Let’s look at another example:

int z = ++x + (y--) + y;  
System.*out*.println(z);

This is a tricky one, so you can expect to see questions like this on the exam.

Going from left to right we evaluate first by adding one to *x* which gives us 11, then we use the value of *y*, which is 10 and then it is decreased by one so that by the time we get to evaluate the final *y* it is 9. So the expression is evaluated by adding 11 to 10 to 9 which results in 30.

I strongly advise you to write code that tests variations of the above expressions in order to become proficient at using these operators.

### Logical negation and unary negative operators

There are two operators that reverse the value of the operand. For boolean values we use the exclamation mark (!) to reverse the value. So a *true* value becomes *false* and vice versa.

### Logical and Short-circuit operators

The logical operators (& ^ |) are applied to boolean values and mean: AND, OR and EXCLUSIVE OR respectively.

All permutations of *true* and *false* and the outcomes for each logical operator can be summarized in truth tables as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **x & y (LOGICAL AND)** | | | |
|  | | **y =** | |
| **true** | **false** |
| **x =** | **true** | true | false |
| **false** | false | false |

|  |  |  |  |
| --- | --- | --- | --- |
| **x | y (LOGICAL OR)** | | | |
|  | | **y =** | |
| **true** | **false** |
| **x =** | **true** | true | true |
| **false** | true | false |

|  |  |  |  |
| --- | --- | --- | --- |
| **x ^ y (EXCLUSIVE OR)** | | | |
|  | | **y =** | |
| **true** | **false** |
| **x =** | **true** | false | true |
| **false** | true | false |

I suggest that you study these tables carefully and understand how the three logical operators work. You will be using them extensively later on in the course, especially in subsection 3.

Exercises:

Which of the following statements are true?

1. binary operators require two operands
2. unary operator require one or two operands
3. ternary operator require three operands
4. unary operators require one operator
5. binary operators require one or two operands

Answers: 1, 3, 4

Which of the following statements are true?

1. Operator precedence determine order of operand evaluation
2. Operator precedence states all operands are evaluated left to right
3. Associativity determines the direction of operator evaluation
4. Associativity overrides operator precedence
5. There are three type of associativity

Answers: 1, 3, 5

Which of the following expressions compile?

1. "hello" + 2 + 10
2. ("hello" + 2) \* 10
3. 100 \* "hello" + 2 + 10
4. 2 + "hello" + 10
5. "hello" + (2 + 10)

Answer: 1, 4, 5

Which of the following statements are true?

1. Left-associative operators of the same precedence are evaluated in order from left to right.
2. Right-associative operators of the same precedence are evaluated in order from right to left.
3. Assignment are right-associative
4. Arithmetic operators are left associative
5. A non-associative operator cannot be combined with other operators of the same precedence.

Answers: 1, 2, 3, 4, 5

Which of the following expressions are correct?

1. 20 + 40 / 4 - 5 \* 7 = -5
2. 8 \* (10 - 2) + 5 = 69
3. 2 + (10/6) = 4
4. 20 + 40 / 4 - 5 \* 7 = 2
5. 8 \* (10 - 2) + 5 = 83
6. 2 + (10/6) = 3

Answers: 1, 2, 6

Given int x = 5 and int y = 10; what is the result of the expression int z = ++x+y+++y-- ?

1. 27
2. 26
3. 37
4. 36

Answer: 1

Given int x = 10 and int y = 1; what is the result of the expression int z = (x--)+--y-x ?

1. -9
2. 0
3. 1
4. 10

Answer: 3

Which of the following does not compile?

1. -true
2. !true
3. -1
4. !1
5. true \* -1

Answers: 1, 4, 5

Given boolean x = true; int a = 8; int b = 4; What is the result of the expression x && ((a > ++b) && !false) and the value of b?

1. false, 5
2. true, 4
3. false, 4
4. true, 5
5. Does not compile

Answers: 4

# Subsection 2: Test equality using == and the equals method

### Equality Operators

Determining the equality or inequality of two numerical values is relatively simple. There are two operators == and != . The result of evaluating two operands of an expression using either of these operators is a boolean.

The equality operators are used to compare two primitive values or two boolean values for equality, as shown in the following code snippet.

int x = 5;  
 char y = 5;  
 System.*out*.println(x == y);  
  
 boolean yes = true;  
 boolean no = false;  
 System.*out*.println(yes != no);

Equality becomes tricky when comparing object instances. An object instance can be *null* or an instance of a class.

When we use the equality operators to compare two object instances we are asking if the memory addresses of the two objects are the same and if they are the same than the two objects are the same object and, of course, equal. Review the following example.

1. BankAccount ABC = new BankAccount();  
2. BankAccount XYZ = ABC;  
3. System.*out*.println(ABC == XYZ);

In this code snippet there is one *BankAccount* object (line 1) and two references to the object *ABC* and *XYZ*. When the two instance variables are tested for equality in line 3 the memory addresses the two reference point to are compared and found to be the same. Therefore the output of line 3 is *true*.

In the following example the output is *false*.

BankAccount a = new BankAccount();  
 BankAccount b = new BankAccount();  
 System.*out*.println(a == b);

It is possible to test to see if a reference variable actually points to an object memory location or not. Remember that the default value of an object reference is always *null* and remains *null* until an object memory address it assigned to it. Consider the following code example:

1. private BankAccount *account*;   
2. public void testAccount(){  
3. System.*out*.println(*account* == null);  
4. }

In this code snippet the account variable is defined as type *BankAccount* but it is not assigned a new object or an existing object, therefore its value is the default object reference value *null*. This is shown in line 3 which print the value *true* to the console.

### Object Equality: Using the .equals method

Every object has an *equals()* method that it inherits from the parent object *Object.class*. We have not discussed the topic of inheritance and the *Object* class yet this will come in **Lesson 3: Object orientation and class structure Section 2: Understand Inheritance,** but all you need to know now is that all classes automatically get an *equals()* method which when called compares two objects for equality.

### String equality

Strings are a special case in Java and testing their equality results in behaviour you shouldn’t expect from all other Java objects and this is why we have this section.

Exercises:

Which of the following expressions are true, given int x = 10 and int y = 5 ?

1. x == y
2. x == x
3. x == (y + y)
4. y == true
5. y != x

Answers: 2, 3, 5

Which of the following compare two BankAccount instances? Assume all instances variables are BankAccount types.

1. account1 == account2
2. accountequals(account2)
3. accountisEquals(account2)
4. accountisEqualsTo(account2)
5. account1 == account1

Answers: 1, 2, 5

Given the following code snippets, select the options that result in equality when compared like this message1 == message2.

1.  
String message1 = "Hello";  
String message2 = "Hello";

2.  
String message1 = "Hello";  
String message2 = " Hello ".trim();

3.  
String message1 = "Hello";  
String message2 = "Hello" + " World";

4.  
String hello1 = new String("Hello");  
String hello2 = "Hello";

5.  
String hello1 = new String("Hello");  
String hello2 = hello1;

Answers: 1, 5

# Subsection 3: Use if, if/else and ternary constructs

Application code is executed in textual order which means that line 1 is executed before line 2 which is executed before line 3 and so on. This is guaranteed but is not always desirable. You will often want to take control of the execution flow and only execute code based on a condition being *true* or *false* and you may want to repeatedly execute the same block of code a certain number of times. This is possible via the use of control flow statements.

In this course we will be looking at all the flow control statements available in Java and will start with the *if-then* family of statements.

### The if-then statement

An *if-then* statement wraps code that we only want to execute if a condition is *true*.

### Tricky examples

Watch out for tricky examples on the exam like the following one.

Consider the following example.

1. if (age >= 13) {  
2. *account* = new BankAccount();  
3. } else if(age > 200){  
4. System.*out*.println("Sorry too old");  
5. }

The *else-if* statement will never get executed as the if condition will be *true* for ages over 200 years. If you want this logic to work you must reorder the conditional logic so the higher value is tested first. See this in the following code snippet.

1. if (age > 200) {  
2. System.*out*.println("Sorry too old");  
3. } else if(age >= 13){  
4. *account* = new BankAccount();  
5. }

Now if the customer’s age is between 13 and 200 a new *BankAccount* will be created otherwise a message is printed to the console.

### The non-use of curly braces and indentations

It is not required to use curly braces to indicate a code block branch. Consider the following example:

1. if (true)  
2. System.*out*.println("Hello World");

The code on line 2 is interpreted by the compiler as the code branch associated with the if condition and will be executed if the condition evaluates to true. There is no requirement for it to be indented in order to be valid, as it is in the example. The same code can be written on a single line, as in the following example:

if (true) System.*out*.println("True");

There can even be a few lines between the if condition and the code branch and it will still compile:

if (true)  
   
   
 System.*out*.println("True");

Remember that comments are striped out by the compiler so the following example compiles:

if (true)  
 // Some comments here  
 System.*out*.println("True");

adding an *else* clause without curly braces is still valid:

if (true)  
 System.*out*.println("True");  
 else   
 System.*out*.println("False");

and combining the lines together is also allowed:

if (true) System.*out*.println("True");  
 else System.*out*.println("False");

Be careful of tricky questions about indentations. Consider the following example:

1. if (false)  
2. System.*out*.println("False");  
3. System.*out*.println("Not True");

What is the outcome of this code? You might be inclined to think that because of the indentations lines 2 and 3 form the code branch associated with the if condition. You’d be mistaken. Only line 2 is associated with the if condition and the outcome of this code is to print “Not True” to the console.

Take care that even though these constructions are legal they are not examples of good coding practice. You should always use braces to indicate clearly to your fellow developers where the code branch start and end. Remember that you want to be the best Java citizen you can be.

### Unusual constructions that are valid

There are some pretty unusual looking constructions of *if-then* statement that are perfectly legal that you might encounter in the exam and in real life. Let’s have a quick look at some of these:

if(true);

This *if* statement does nothing useful but is still legal.

if(true);  
else if(false);  
else;

This statement does nothing useful but is still legal.

if (8 == 81) {}

The result of this equality test is false and therefore it is valid.

if (bool = false) {}

If we assume that bool is declared as a boolean and knowing that the result of an assignment is the value of the assignment, in this case *false*, we can see why this is a valid if condition. It is equivalent to:

if (false) {}

If in the following example we assume that x is an integer and knowing that the result of an assignment is the value of the assignment, in this case it will be a boolean value, we can see why this is a valid if condition.

if (x == 10 ? true : false) { }

This condition contains a ternary statement which we look at next.

### Unreachable code

The Java compiler will identify any code that never runs because a condition will never be satisfied and flags it as an unreachable code error and compilation will fail.

However, in the case of if condition there is a specified exception in the Java Language Specification ([JLS 14.21 Unreachable Statement](https://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html)) that will allow unreachable code. Let’s see some examples:

if (false) {  
 x = 3;  
}

In this case the assignment of 3 to x will never occur and this code branch is unreachable. However it does not cause an unreachable code error. Although the body of the condition is unreachable the JLS explicitly defines this as an exception to the rule. It allows this construct to support optimizations through the conditional compilation. The following examples demonstrates this exception to the rule.

boolean DEBUG = true;  
if (DEBUG) {  
 System.*out*.println("beginning debug task 1");  
}

Here, the DEBUG variable can be set to *false* in the code while generating the production version of the class file, which will allow the compiler to optimize the code by removing the whole if statement entirely from the class file.

### The Ternary Operator

The ternary operator is a simplified *if-then-else* statement but is evaluated according to the rules of precedence set out in table 1 and is executed second to last before assignments.

Exercises:

Which of follow code fragments are valid? Assume all variables have been defined and declared.

1.  
if(age >= 13){  
 System.out.println("Welcome");  
}

2.  
if(!true) System.out.println("Hello World!");

3.   
if(!GetTrueOrFalse()) System.out.println("I am true");

4.  
if(true) if(false);

5.  
if(-1) {  
 System.out.println("Error");  
}

Answers: 1, 2, 3, 4

Which of the following code fragments compile? Assume all variables have been defined and declared.

1.   
if(age < 0){  
 System.out.println("Sorry too young");  
 } else {  
 System.out.println("Sorry too old");  
 }

2.  
if(age < 20){  
 System.out.println("Special account for you");  
 } else if (age < 10 ) {  
 System.out.println("Sorry too young");  
 }

3.  
if(score > 0 & score < 20){  
 System.out.println("You lose");  
 } else {  
 System.out.println("You win!");  
 } else if (score > 100 ) {  
 System.out.println("Jackpot!");  
 }

4.  
if (age >= 13) {  
 account = new BankAccount();  
 } else if(age > 200){  
 System.out.println("Sorry too old");  
 }

5.  
if (age > 100)  
 // Never too old  
 System.out.println("Welcome");

Answers: 1, 2, 4, 5

Which of the following ternary statements compile?

1. age >= 21 ? "Welcome" : "Too Young";
2. score > 100 ? "You Win" : 0.0f
3. score > 100 ? a() : b();
4. float result = score > 100 ? "You Win" : 0.0f;
5. score > 100 ? score > 200 ? "You Win Big" : "You Win Normal" : "You Lose"

Answers: 1, 2, 3, 5

# Subsection 4: Use switch/case statements

Let’s move on to another structure that allows us to control the flow of execution, the switch/case statement. The switch statement evaluates a variable and depending on the value of the variable a code block is chosen and executed. These code blocks are called case statements and the first case that matches the switch variable will be executed. If no case is found that matches the switch variable the default case is executed and if no default case is defined the execution flow jumps to the first line after the end of the switch/case statement.