1 JAVA – BASIC SYNTAX

1.1 BASIC SYNTAX

About Java programs, it is very important to keep in mind the following points.

- **Case Sensitivity -** Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.
- **Class Names -** For all class names the first letter should be in Upper Case.

  If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

  Example: *class MyFirstJavaClass*

- **Method Names -** All method names should start with a Lower Case letter.

  If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

  Example: *public void myMethodName()*

- **Program File Name -** Name of the program file should exactly match the class name.

  When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).

  Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as *'MyFirstJavaProgram.java'*

- **public static void main(String args[]) -** Java program processing starts from the main() method which is a mandatory part of every Java program.

1.2 JAVA IDENTIFIERS

All Java components require names. Names used for classes, variables, and methods are called **identifiers**.

In Java, there are several points to remember about identifiers. They are as follows:

- All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).

- After the first character, identifiers can have any combination of characters.
- A key word cannot be used as an identifier.
- Most importantly, identifiers are case sensitive.
- Examples of legal identifiers: age, $salary, _value.
- Examples of illegal identifiers: 123abc, -salary.

1.3 JAVA MODIFIERS

Like other languages, it is possible to modify classes, methods, etc., by using modifiers. There are two categories of modifiers:

- **Access Modifiers:** default, public, protected, private
- **Non-access Modifiers:** final, abstract, strictfp

We will be looking into more details about modifiers in the next section.

1.4 JAVA VARIABLES

Following are the types of variables in Java:

- Local Variables
- Class Variables (Static Variables)
- Instance Variables (Non-static Variables)

1.5 JAVA ARRAYS

Arrays are objects that store multiple variables of the same type. However, an array itself is an object on the heap. We will look into how to declare, construct, and initialize in the upcoming chapters.

1.6 JAVA ENUMS

Enums were introduced in Java 5.0. Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.

With the use of enums it is possible to reduce the number of bugs in your code.

For example, if we consider an application for a fresh juice shop, it would be possible to restrict the glass size to small, medium, and large. This would make sure that it would not allow anyone to order any size other than small, medium, or large.

Example

```
class FreshJuice {
  enum FreshJuiceSize {SMALL, MEDIUM, LARGE}
  FreshJuiceSize size;
}

public class FreshJuiceTest {
  public static void main(String args[]) {
    FreshJuice juice = new FreshJuice();
    juice.size = FreshJuice.FreshJuiceSize.MEDIUM;
    System.out.println("Size: " + juice.size);
  }
}
```

The above example will produce the following result:

```
Size: MEDIUM
```

Note: Enums can be declared as their own or inside a class. Methods, variables, constructors can be defined inside enums as well.

1.7 JAVA KEYWORDS

The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return
short	static	strictfp	super
switch	synchronized	this	throw
throws	transient	try	void

volatile	while		

1.8 Comments in Java

Java supports single-line and multi-line comments very similar to C and C++. All characters available inside any comment are ignored by Java compiler.

```
public class MyFirstJavaProgram {
  /* This is my first java program.
  * This will print 'Hello World' as the output
  * This is an example of multi-line comments.
  */
  public static void main(String[] args) {
    // This is an example of single line comment
    System.out.println("Hello World");
  }
}
```

1.9 Using Blank Lines

A line containing only white space, possibly with a comment, is known as a blank line, and Java totally ignores it.

2 Java – Basic Datatypes

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in the memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different datatypes to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java:

- Primitive Datatypes
- Reference/Object Datatypes

2.1 Primitive Datatypes

There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the language and named by a keyword. Let us now look into the eight primitive data types in detail.

byte:

- Byte data type is an 8-bit signed two's complement integer
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive) (2^7 -1)

- Default value is 0
- Byte datatype is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer
- Example: byte a = 100, byte b = -50

short:

- Short datatype is a 16-bit signed two's complement integer
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767 (inclusive) (2^15 -1)
- Short datatype can also be used to save memory as byte data type. A short is 2 times smaller than an integer
- Default value is 0
- Example: short s = 10000, short r = -20000

int:

- Int datatype is a 32-bit signed two's complement integer
- Minimum value is - 2,147,483,648 (-2^31)
- Maximum value is 2,147,483,647(inclusive) (2^31 -1)
- Integer is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0
- Example: int a = 100000, int b = -200000

long:

- Long datatype is a 64-bit signed two's complement integer
- Minimum value is -9,223,372,036,854,775,808 (-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive) (2^63 -1)
- This type is used when a wider range than int is needed
- Default value is 0L
- Example: long a = 100000L, long b = -200000L

float:

- Float datatype is a single-precision 32-bit IEEE 754 floating point
- Float is mainly used to save memory in large arrays of floating point numbers
- Default value is 0.0f
- Float datatype is never used for precise values such as currency
- Example: float f1 = 234.5f

double:

- double datatype is a double-precision 64-bit IEEE 754 floating point
- This datatype is generally used as the default data type for decimal values, generally the default choice
- Double datatype should never be used for precise values such as currency
- Default value is 0.0d
- Example: double d1 = 123.4

boolean:

- boolean datatype represents one bit of information
- There are only two possible values: true and false
- This datatype is used for simple flags that track true/false conditions
- Default value is false
- Example: boolean one = true

char:

- char datatype is a single 16-bit Unicode character
- Minimum value is '\u0000' (or 0)
- Maximum value is '\uffff' (or 65,535 inclusive)
- Char datatype is used to store any character
- Example: char letterA ='A'

2.2 Reference Datatypes

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.
- Class objects and various type of array variables come under reference datatype.
- Default value of any reference variable is null.
- A reference variable can be used to refer any object of the declared type or any compatible type.
- Example: Animal animal = new Animal("giraffe");

2.3 Java Literals

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example:

```
byte a = 68;
char a = 'A'
```

byte, int, long, and short can be expressed in decimal (base 10), hexadecimal (base 16) or octal (base 8) number systems as well.

Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using these number systems for literals. For example:

```
int decimal = 100;
int octal = 0144;
int hexa =  0x64;
```

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

String and char types of literals can contain any Unicode characters. For example:

```
char a = '\u0001';
String a = "\u0001";
```

Java language supports few special escape sequences for String and char literals as well. They are:

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)
\t	Tab
\"	Double quote

\'	Single quote
\\	backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

3 JAVA – VARIABLE TYPES

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

You must declare all variables before they can be used. Following is the basic form of a variable declaration:

```
data type variable [ = value][, variable [= value] ...] ;
```

Here *data type* is one of Java's datatypes and *variable* is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Following are valid examples of variable declaration and initialization in Java:

```
int a, b, c;         // Declares three ints, a, b, and c.
int a = 10, b = 10;  // Example of initialization

byte B = 22;         // initializes a byte type variable B.
double pi = 3.14159; // declares and assigns a value of PI.

char a = 'a';        // the char variable a iis initialized with value 'a'
```

This chapter will explain various variable types available in Java Language. There are three kinds of variables in Java:

- Local variables
- Instance variables
- Class/Static variables

3.1 LOCAL VARIABLES

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor, or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

Example

Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to only this method.

```
public class Test {
  public static void main(String args[]) {
    Test test = new Test();
    test.pupAge();
  }

  public void pupAge() {
    int age = 0;
    age = age + 7;
    System.out.println("Puppy age is : " + age);
  }
}
```

This will produce the following result:

```
Puppy age is: 7
```

Example

Following example uses *age* without initializing it, so it would give an error at the time of compilation.

```
public class Test {
  public static void main(String args[]) {
    Test test = new Test();
    test.pupAge();
  }

  public void pupAge() {
    int age;
    age = age + 7;
    System.out.println("Puppy age is : " + age);
  }
}
```

This will produce the following error while compiling it:

```
Test.java:4:variable number might not have been initialized
age = age + 7;
         ^
1 error
```

3.2 Instance Variables

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.

- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name *ObjectReference.VariableName*.

Example

```
import java.io.*;

public class Employee {
  // this instance variable is visible for any child class.
  public String name;
  // salary variable is visible in Employee class only.
  private double salary;
  // The name variable is assigned in the constructor.
  public Employee(String empName) {
    name = empName;
  }

  public static void main(String args[]) {
    Employee empOne = new Employee("Ransika");
    empOne.setSalary(1000);
    empOne.printEmp();
  }

  // The salary variable is assigned a value.
  public void setSalary(double empSal) {
    salary = empSal;
  }

  // This method prints the employee details.
  public void printEmp() {
    System.out.println("name  : " + name);
    System.out.println("salary :" + salary);
  }
}
```

This will produce the following result:

```
name  : Ransika
salary :1000.0
```

3.3 CLASS/STATIC VARIABLES

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.
- Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name *ClassName.VariableName*.
- When declaring class variables as public static final, then variable names (constants) are all in upper case. If the static variables are not public and final, the naming syntax is the same as instance and local variables.

Example

```
import java.io.*;
public class Employee{

   // salary  variable is a private static variable
   private static double salary;


   // DEPARTMENT is a constant
   public static final String DEPARTMENT = "Development ";
   public static void main(String args[]){

      salary = 1000;
      System.out.println(DEPARTMENT + "average salary:" + salary);
   }
}
```

This will produce the following result:

```
Development average salary:1000
```

Note: If the variables are accessed from an outside class, the constant should be accessed as Employee.DEPARTMENT

4 JAVA – MODIFIER TYPES

Modifiers are keywords that you add to those definitions to change their meanings. Java language has a wide variety of modifiers, including the following:

- Java Access Modifiers
- Non Access Modifiers

4.1 JAVA ACCESS MODIFIERS

Java provides a number of access modifiers to set access levels for classes, variables, methods, and constructors. The four access levels are:

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

4.2 Default Access Modifier - No Keyword

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

Example

Variables and methods can be declared without any modifiers, as in the following examples:

```
String version = "1.5.1";
boolean processOrder() {
   return true;
}
```

Private Access Modifier - Private

- Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.
- Private access modifier is the most restrictive access level. Class and interfaces cannot be private.
- Variables that are declared private can be accessed outside the class, if public getter methods are present in the class.
- Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

Example

The following class uses private access control:

```
public class Logger {
  private String format;

  public String getFormat() {
    return this.format;
  }

  public void setFormat(String format) {
    this.format = format;
  }
}
```

Here, the *format* variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly.

So, to make this variable available to the outside world, we defined two public methods: *getFormat()*, which returns the value of format, and *setFormat(String)*, which sets its value.

Public Access Modifier - Public

A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Example

The following function uses public access control:

```
public static void main(String[] arguments) {
   // ...
}
```

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

Protected Access Modifier - Protected

Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Example

The following parent class uses protected access control, to allow its child class override *openSpeaker()* method:

```
class AudioPlayer {
  protected boolean openSpeaker(Speaker sp) {
    // implementation details
```

```
    return;
  }
}

class StreamingAudioPlayer {
  boolean openSpeaker(Speaker sp) {
    // implementation details
    return;
  }
}
```

Here, if we define *openSpeaker()* method as private, then it would not be accessible from any other class other than *AudioPlayer*. If we define it as public, then it would become accessible to all the outside world. But our intention is to expose this method to its subclass only, that's why we have used *protected* modifier.

Access Control and Inheritance

The following rules for inherited methods are enforced:

- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
- Methods declared private are not inherited at all, so there is no rule for them.

4.3 Java Non-Access Modifiers

Java provides a number of non-access modifiers to achieve many other functionalities.

- The *static* modifier for creating class methods and variables.
- The final modifier for finalizing the implementations of classes, methods, and variables.
- The abstract modifier for creating abstract classes and methods.
- The synchronized and *volatile* modifiers, which are used for threads.

4.4 The Static Modifier

Static Variables

The *static* keyword is used to create variables that will exist independently of any instances created for the class. Only one copy of the static variable exists regardless of the number of instances of the class.

Static variables are also known as class variables. Local variables cannot be declared static.

Static Methods

The *static* keyword is used to create methods that will exist independently of any instances created for the class.

Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.

Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.

Example

The static modifier is used to create class methods and variables, as in the following example:

```
public class InstanceCounter {
  private static int numInstances = 0;

  InstanceCounter() {
    InstanceCounter.addInstance();
  }

  protected static int getCount() {
    return numInstances;
  }

  private static void addInstance() {
    numInstances++;
  }

  public static void main(String[] arguments) {
    System.out.println("Starting with " + InstanceCounter.getCount() + "
instances");
    for (int i = 0; i < 500; ++i) {
      new InstanceCounter();
    }
    System.out.println("Created " + InstanceCounter.getCount() + "
instances");
  }
}
```

This will produce the following result:

```
Started with 0 instances
Created 500 instances
```

4.5 The Final Modifier

Final Variables

A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to a different object.

However, the data within the object can be changed. So, the state of the object can be changed but not the reference.

With variables, the *final* modifier often is used with *static* to make the constant a class variable.

Example

```
public class Test {
  // The following are examples of declaring constants:
  public static final int BOXWIDTH = 6;
  static final String TITLE = "Manager";
  final int value = 10;

  public void changeValue() {
    value = 12; //will give an error
  }
}
```

Final Methods

A final method cannot be overridden by any subclasses. As mentioned previously, the final modifier prevents a method from being modified in a subclass.

The main intention of making a method final would be that the content of the method should not be changed by any outsider.

Example

You declare methods using the *final* modifier in the class declaration, as in the following example:

Final Classes

```
public class Test {
  public final void changeName() {
// body of method
  }
}
```

The main purpose of using a class being declared as *final* is to prevent the class from being sub classed. If a class is marked as final, then no class can inherit any feature from the final class.

Example

```
public final class Test {
// body of class
}
```

4.6 The Abstract Modifier

Abstract Class

An abstract class can never be instantiated. If a class is declared as abstract, then the sole purpose is for the class to be extended.

A class cannot be both abstract and final (since a final class cannot be extended). If a class contains abstract methods, then the class should be declared abstract. Otherwise, a compile error will be thrown.

An abstract class may contain both abstract methods as well normal methods.

Example

```
abstract class Caravan {
  private double price;
  private String model;
  private String year;

  public abstract void goFast(); //an abstract method

  public abstract void changeColor();
}
```

Abstract Methods

An abstract method is a method declared without any implementation. The methods body (implementation) is provided by the subclass. Abstract methods can never be final or strict.

Any class that extends an abstract class must implement all the abstract methods of the super class, unless the subclass is also an abstract class.

If a class contains one or more abstract methods, then the class must be declared abstract. An abstract class does not need to contain abstract methods.

The abstract method ends with a semicolon. Example: public abstract sample();

Example

```
public abstract class SuperClass {
  abstract void m(); //abstract method
}

class SubClass extends SuperClass {
  // implements the abstract method
  void m() {
  .........
  }
}
```

The Synchronized Modifier

The synchronized keyword used to indicate that a method can be accessed by only one thread at a time. The synchronized modifier can be applied with any of the four access level modifiers.

Example

```
public synchronized void showDetails(){
.......
}
```

The Transient Modifier

An instance variable is marked transient to indicate the JVM to skip the particular variable when serializing the object containing it.

This modifier is included in the statement that creates the variable, preceding the class or data type of the variable.

Example

```
public transient int limit = 55;   // will not persist
public int b; // will persist
```

The Volatile Modifier

The volatile modifier is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.

Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.

Example

```
public class MyRunnable implements Runnable {
  private volatile boolean active;

  public void run() {
    active = true;
    while (active) { // line 1
// some code here
    }
  }

  public void stop() {
    active = false; // line 2
  }
}
```

Usually, run() is called in one thread (the one you start using the Runnable), and stop() is called from another thread. If in line 1, the cached value of active is used, the loop may not stop when you set active to false in line 2. That's when you want to use *volatile*.

To use a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following example.

```
public class className {
// ...
}
  protected static final int BOXWIDTH = 42;
  static final double weeks = 9.5;
  private boolean myFlag;

  public static void main(String[] arguments) {
// body of method
  }
```

4.7 Access Control Modifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

4.8 Non-Access Modifiers

Java provides a number of non-access modifiers to achieve many other functionalities.

- The static modifier for creating class methods and variables.
- The final modifier for finalizing the implementations of classes, methods, and variables.
- The abstract modifier for creating abstract classes and methods.
- The synchronized and *volatile* modifiers, which are used for threads.

5 JAVA – BASIC OPERATORS

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc. Operators

5.1 THE ARITHMETIC OPERATORS

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20, then:

Sr.No.	Operator and Example
1	**+ (Addition)** Adds values on either side of the operator **Example:** A + B will give 30
2	**- (Subtraction)** Subtracts right-hand operand from left-hand operand **Example:** A - B will give -10
3	*** (Multiplication)** Multiplies values on either side of the operator **Example:** A * B will give 200
4	**/ (Division)** Divides left-hand operand by right-hand operand **Example:** B / A will give 2

5	**% (Modulus)** Divides left-hand operand by right-hand operand and returns remainder **Example:** B % A will give 0
6	**++ (Increment)** Increases the value of operand by 1 **Example:** B++ gives 21
7	**-- (Decrement)** Decreases the value of operand by 1 **Example:** B-- gives 19

Example

The following program is a simple example which demonstrates the arithmetic operators. Copy and paste the following Java program in Test.java file, and compile and run this program:

```
public class Test {
  public static void main(String args[]) {
    int a = 10;
    int b = 20;
    int c = 25;
    int d = 25;
    System.out.println("a + b = " + (a + b));
    System.out.println("a - b = " + (a - b));
    System.out.println("a * b = " + (a * b));
    System.out.println("b / a = " + (b / a));
    System.out.println("b % a = " + (b % a));
    System.out.println("c % a = " + (c % a));
    System.out.println("a++  = " + (a++));
    System.out.println("b--  = " + (a--));
// Check the difference in d++ and ++d
    System.out.println("d++  = " + (d++));
    System.out.println("++d  = " + (++d));
  }
}
```

This will produce the following result:

```
a + b = 30
a - b = -10
a * b = 200 b / a = 2
b % a = 0 c % a = 5
a++ = 10
b-- = 11
d++ = 25
++d = 27
```

5.2 THE RELATIONAL OPERATORS

There are following relational operators supported by Java language. Assume variable A holds 10 and variable B holds 20, then:

Sr. No.	Operator and Description
1	**== (equal to)** Checks if the values of two operands are equal or not, if yes then condition becomes true. **Example:** (A == B) is not true.
2	**!= (not equal to)** Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. **Example:** (A != B) is true.
3	**> (greater than)** Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. **Example:** (A > B) is not true.
4	**< (less than)** Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. **Example:** (A < B) is true.
5	**>= (greater than or equal to)** Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. **Example:** (A >= B) is not true.
6	**<= (less than or equal to)** Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. **Example:** (A <= B) is true.

Example

The following program is a simple example that demonstrates the relational operators. Copy and paste the following Java program in Test.java file and compile and run this program.

```
public class Test {

  public static void main(String args[]) {
    int a = 10;
    int b = 20;
    System.out.println("a == b = " + (a == b));
    System.out.println("a != b = " + (a != b));
    System.out.println("a > b = " + (a > b));
    System.out.println("a < b = " + (a < b));
    System.out.println("b >= a = " + (b >= a));
    System.out.println("b <= a = " + (b <= a));
  }
}
```

This will produce the following result:

```
a == b = false
a != b = true
a > b = false
a < b = true b
>= a = true b
<= a = false
```

5.3 The Bitwise Operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b

= 13; now in binary format they will be as follows:

```
a = 0011 1100
b = 0000 1101
-----------------
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011
```

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then:

Sr. No.	Operator and Description
1	**& (bitwise and)** Binary AND Operator copies a bit to the result if it exists in both operands. **Example:** (A & B) will give 12 which is 0000 1100
2	**| (bitwise or)** **Binary OR Operator copies a bit if it exists in either operand.** **Example: (A | B) will give 61 which is 0011 1101**
3	**^ (bitwise XOR)** **Binary XOR Operator copies the bit if it is set in one operand but not both.** **Example: (A ^ B) will give 49 which is 0011 0001**
4	**~ (bitwise compliment)** **Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.** **Example: (~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.**
5	**<< (left shift)** **Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.** **Example: A << 2 will give 240 which is 1111 0000**
6	**>> (right shift)** **Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.** **Example: A >> 2 will give 15 which is 1111**

7	**>>> (zero fill right shift)** **Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.** **Example: A >>>2 will give 15 which is 0000 1111**

Example

The following program is a simple example that demonstrates the bitwise operators. Copy and paste the following Java program in Test.java file and compile and run this program:

```
public class Test {
  public static void main(String args[]) {
    int a = 60;  /* 60 = 0011 1100 */
    int b = 13;  /* 13 = 0000 1101 */
    int c = 0;

    c = a & b; /* 12 = 0000 1100 */
    System.out.println("a & b = " + c);

    c = a | b; /* 61 = 0011 1101 */
    System.out.println("a | b = " + c);

    c = a ^ b; /* 49 = 0011 0001 */
    System.out.println("a ^ b = " + c);

    c = ~a; /*-61 = 1100 0011 */
    System.out.println("~a = " + c);

    c = a << 2; /* 240 = 1111 0000 */
    System.out.println("a << 2 = " + c);

    c = a >> 2; /* 15 = 1111 */
    System.out.println("a >> 2 = " + c);

    c = a >>> 2; /* 15 = 0000 1111 */
    System.out.println("a >>> 2 = " + c);
  }
}
```

This will produce the following result:

```
a & b = 12
a | b = 61
a ^ b = 49

~a = -61
a << 2 = 240
a >> 15
a >>> 15
```

5.4 THE LOGICAL OPERATORS

The following table lists the logical operators:

Assume Boolean Variables A holds true and variable B holds false, then:

Operator	Description
1	**&& (logical and)** Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. **Example:** (A && B) is false.
2	**|| (logical or)** Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true. **Example:** (A || B) is true.
3	**! (logical not)** Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. **Example:** !(A && B) is true.

Example

The following simple example program demonstrates the logical operators. Copy and paste the following Java program in Test.java file and compile and run this program:

```
public class Test {
  public static void main(String args[]) {
    boolean a = true;
    boolean b = false;

    System.out.println("a && b = " + (a && b));

    System.out.println("a || b = " + (a || b));

    System.out.println("!(a && b) = " + !(a && b));
  }
}
```

This will produce the following result:

```
a && b = false
a || b = true

!(a && b) = true
```

5.5 The Assignment Operators

Following are the assignment operators supported by Java language:

Sr. No.	Operator and Description
1	**=** Simple assignment operator. Assigns values from right side operands to left side operand. **Example:** C = A + B will assign value of A + B into C
2	**+=** **Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.** **Example: C += A is equivalent to C = C + A**
3	**-=** **Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.** **Example:C -= A is equivalent to C = C – A**

4	***=** **Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.** **Example: C *= A is equivalent to C = C * A**
5	**/=** **Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.** **Example: C /= A is equivalent to C = C / A**
6	**%=** **Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.** **Example: C %= A is equivalent to C = C % A**
7	**<<=** **Left shift AND assignment operator.** **Example: C <<= 2 is same as C = C << 2**
8	**>>=** **Right shift AND assignment operator** **Example: C >>= 2 is same as C = C >> 2**
9	**&=** **Bitwise AND assignment operator.** **Example: C &= 2 is same as C = C & 2**
10	**^=** **bitwise exclusive OR and assignment operator.** **Example: C ^= 2 is same as C = C ^ 2**
11	**|=** **bitwise inclusive OR and assignment operator.** **Example: C |= 2 is same as C = C | 2**

Example

The following program is a simple example that demonstrates the assignment operators. Copy and paste the following Java program in Test.java file. Compile and run this program:

```
public class Test {
  public static void main(String args[]) { int a = 10;
    int b = 20; int c = 0;
    c = a + b;
    System.out.println("c = a + b = " + c );

    c += a ;
    System.out.println("c += a = " + c );

    c -= a ;
    System.out.println("c -= a = " + c );

    c *= a ;
    System.out.println("c *= a = " + c );
    a = 10;
    c = 15;
    c /= a ;
    System.out.println("c /= a = " + c );

    a = 10;
    c = 15;
    c %= a ;
    System.out.println("c %= a = " + c );

    c <<= 2 ;
    System.out.println("c <<= 2 = " + c );

    c >>= 2 ;
    System.out.println("c >>= 2 = " + c );

    c >>= 2 ;
    System.out.println("c >>= a = " + c );

    c &= a ;
    System.out.println("c &= 2 = " + c );

    c ^= a ;
    System.out.println("c ^= a = " + c );

    c |= a ;
    System.out.println("c |= a = " + c );
  }
}
```

This will produce the following result: a=10 b=20

```
c = a + b = 30
c += a  = 40
c -= a = 30
c *= a = 300
c /= a = 1
c %= a  = 5
c <<= 2 = 20
c >>= 2 = 5
c >>= 2 = 1
c &= a  = 0
c ^= a   = 10
c |= a   = 10
```

5.6 Miscellaneous Operators

There are few other operators supported by Java Language.

Conditional Operator (? :)

Conditional operator is also known as the **ternary operator**. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide; which value should be assigned to the variable. The operator is written as:

```
variable x = (expression) ? value if true : value if false
```

Following is an example:

```
public class Test {
  public static void main(String args[]){ int a, b;
    a = 10;
    b = (a == 1) ? 20: 30;
    System.out.println( "Value of b is : " +  b );

    b = (a == 10) ? 20: 30;
    System.out.println( "Value of b is : " + b );
  }
}
```

This will produce the following result:

```
Value of b is : 30
Value of b is : 20
```

instanceof Operator

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as:

```
( Object reference variable ) instanceof  (class/interface type)
```

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example:

```
public class Test {

  public static void main(String args[]) {
    String name = "James";
    // following will return true since name is type of String
    boolean result = name instanceof String;
    System.out.println(result);
  }
}
```

This will produce the following result:

```
true
```

This operator will still return true, if the object being compared is the assignment compatible with the type on the right. Following is one more example:

```
class Vehicle {
}

public class Car extends Vehicle {
  public static void main(String args[]) {
    Vehicle a = new Car();
    boolean result = a instanceof Car;
    System.out.println(result);
  }
}
```

This will produce the following result:

```
true
```

5.7 Precedence of Java Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example, x = 7 + 3 * 2; here x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

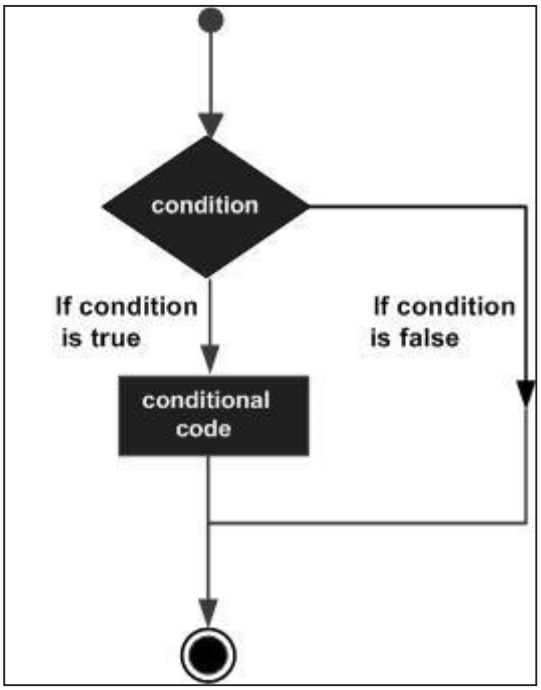
Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left toright
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left

6 JAVA – DECISION MAKING

Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:



Java programming language provides following types of decision making statements. Click the following links to check their detail.

Statement	Description
if statement	An **if statement** consists of a Boolean expression followed by one or more statements.
if...else statement	An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.
nested if statements	You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

switch statement	A **switch** statement allows a variable to be tested for equality against a list of values.

6.1 IF STATEMENT IN JAVA

An **if** statement consists of a Boolean expression followed by one or more statements.

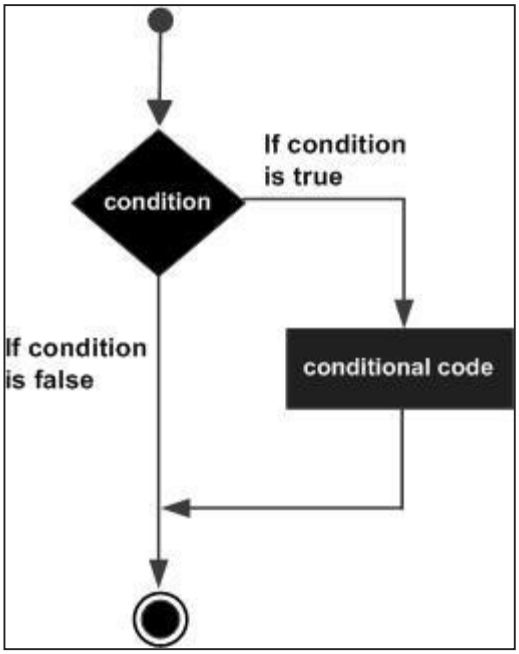
Syntax

Following is the syntax of an if statement:

```
if(Boolean_expression)
{
   //Statements will execute if the Boolean expression is true
}
```

If the Boolean expression evaluates to true, then the block of code inside the if statement will be executed. If not, the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Flow Diagram



Example

```
public class Test {
  public static void main(String args[]) {
    int x = 10;
    if (x < 20) {
      System.out.print("This is if statement");
    }
  }
}
```

This will produce the following result:

```
This is if statement.
```

6.2 IF-ELSE STATEMENT IN JAVA

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

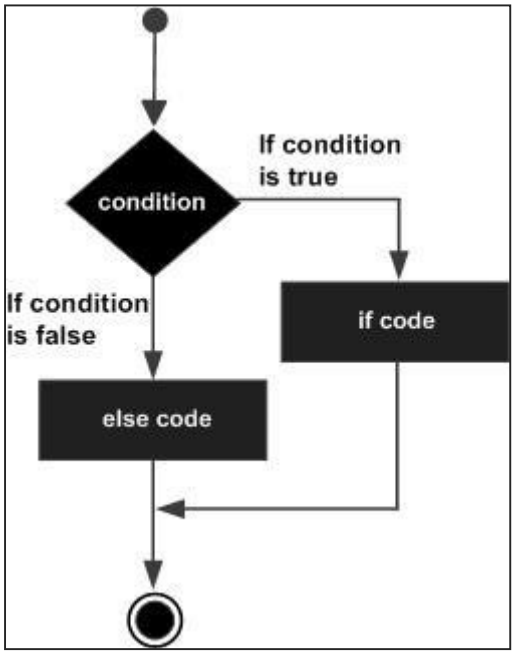
Syntax

Following is the syntax of an if...else statement:

```
if(Boolean_expression){
   //Executes when the Boolean expression is true
}else{
   //Executes when the Boolean expression is false
}
```

If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

Flow Diagram



Example

```
public class Test {
  public static void main(String args[]) {
    int x = 30;
    if (x < 20) {
      System.out.print("This is if statement");
    } else {
      System.out.print("This is else statement");
    }
  }
}
```

This will produce the following result:

```
This is else statement
```

6.3 THE IF...ELSE IF...ELSE STATEMENT

An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single if...else if statement.

When using if, else if, else statements there are a few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax

Following is the syntax of an if...else statement:

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2){
   //Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3){
   //Executes when the Boolean expression 3 is true
}else {
   //Executes when the none of the above condition is true.
}
```

Example

```
public class Test {
  public static void main(String args[]) {
    int x = 30;
    if (x == 10) {
      System.out.print("Value of X is 10");
    } else if (x == 20) {
      System.out.print("Value of X is 20");
    } else if (x == 30) {
      System.out.print("Value of X is 30");
    } else {
      System.out.print("This is else statement");
    }
  }
}
```

This will produce the following result:

```
Value of X is 30
```

6.4 NESTED IF STATEMENT IN JAVA

It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

Syntax

The syntax for a nested if...else is as follows:

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true
   if(Boolean_expression 2){
      //Executes when the Boolean expression 2 is true
   }
}
```

You can nest **else if...else** in the similar way as we have nested *if* statement.

Example

```
public class Test {
  public static void main(String args[]) {
    int x = 30;
    int y = 10;

    if (x == 30) {
      if (y == 10) {
        System.out.print("X = 30 and Y = 10");
      }
    }
  }
}
```

This will produce the following result:

```
X = 30 and Y = 10
```

6.5 SWITCH STATEMENT IN JAVA

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

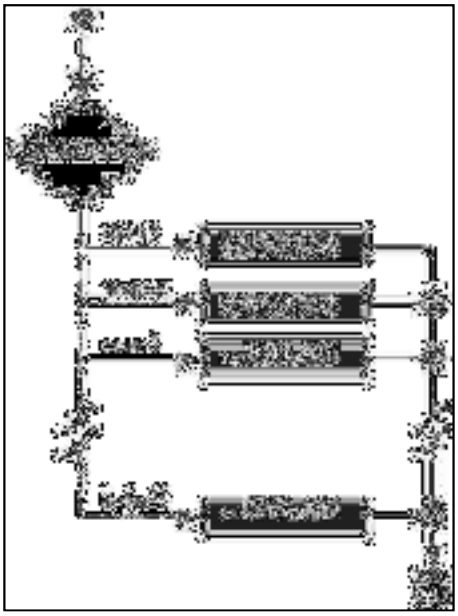
The syntax of enhanced for loop is:

```
switch(expression){
    case value :
       //Statements
       break; //optional
    case value :
       //Statements
       break; //optional
    //You can have any number of case statements.
    default : //Optional
       //Statements
}
```

The following rules apply to a **switch** statement:

- The variable used in a switch statement can only be integers, convertable integers (byte, short, char), strings and enums.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Flow Diagram



Example

```
public class Test {
  public static void main(String args[]) {
    //char grade = args[0].charAt(0);
     char grade = 'C';

    switch (grade) {
      case 'A':
        System.out.println("Excellent!");
        break;
      case 'B':
      case 'C':
        System.out.println("Well done");
        break;
      case 'D':
        System.out.println("You passed");
      case 'F':
        System.out.println("Better try again");
        break;
      default:
        System.out.println("Invalid grade");
    }
    System.out.println("Your grade is " + grade);
  }
}
```

Compile and run the above program using various command line arguments. This will produce the following result:

```
$ java
Test Well
done

Your grade is a C
```

6.6 THE ? : OPERATOR:

We have covered **conditional operator ? :** in the previous chapter which can be used to replace **if...else** statements. It has the following general form:

```
Exp1 ? Exp2 : Exp3;
```

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon. To determine the value of the whole expression, initially exp1 is evaluated.

- If the value of exp1 is true, then the value of Exp2 will be the value of the whole expression.
- If the value of exp1 is false, then Exp3 is evaluated and its value becomes the value of the entire expression.