# Introduction:

## Software:

A software is an automated solution for real-world problems or real-world needs. It can also be called as set of programs that provide instructions to any programmable hardware to automate the work/functionalities.

## Programs:

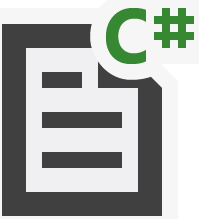
Programs are set of instructions or code that is developed for a platform.

## Language:

It is method that is used to communicate. Communication is always a two-way process (send and receive).

## Programming Language:

Language which is used to develop a software is called as programming language.



## Platform:

Fig: Programs in different languages.

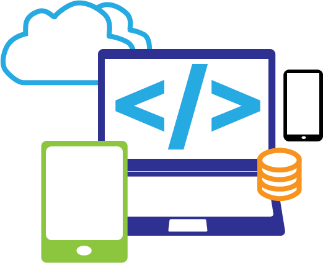
Platform is a combination of hardware and software environment (operating system + Processing Unit) where the program runs. These days, in the current generation, every electronic device is equipped with a platform. Similarly, Java is a software platform (software-only platform).

Fig: Different platforms

**Example**: **Hardware and Operating Systems:** Windows + Intel – Wintel – Platform AMD + Linux – Platform

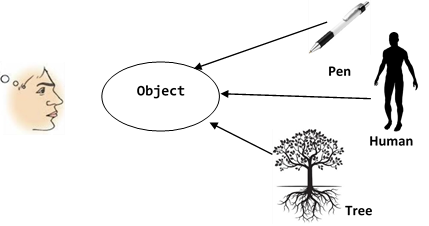
IBM PC + Mac – Platform

**Car Infotainment:** Android Auto Apple CarPlay

**Software only platform:** JVM – Java Virtual Machine Mobile platform; refrigerator platform (smart fridge).

## Object Orientation:

Pursuing / seeing everything in the form of object is called as object orientation.



**Fig: Object – Orientation**

*Java* is a high-level object-oriented programming language. As java is an object-oriented language, we create replica of object in the real world which has to be programmed. In other words, software objects (objects created programmatically) is often used to model real world objects.

# The evolution of Java:

There is a very interesting history behind Java. A team of engineers from Sun Microsystems wanted to develop a programming language for electronic devices. The team was named Green Team, led by James Gosling. As the programming language was the product of the Green Team, they initially called it Green Talk but later changed the name to Oak, which is a symbol of Strength or Robustness. But since the name Oak was already a registered trademark of Oak Technologies, it can’t be named as Oak. Then they had brain storming session and came up with many names like DNA, Silk, Ruby, WRL, and Java. Finally named the product as Java which symbolizes the coffee from the island “Java” where the coffee was grown.

When the Java was built according to the initial idea of 'language for electronic devices' it was considered to be designed for interactive television. But it was found to be very advanced for digital cable industry at that time. Java was evolved as a very powerful and robust programming language. James Gosling designed Java with a C/C++ style syntax that system and application programmers would find familiar.

Java has several versions. The current version release is Java SE-11.

The first version of Java was released in January1996 (JDK 1.0—JDK stands for Java Development Kit)

The other versions and their release dates are as follows:

* JDK Alpha and Beta (1995)
* JDK 1.0 (23rd Jan 1996)
* JDK 1.1 (19th Feb 1997)
* J2SE 1.2 (8th Dec 1998)
* J2SE 1.3 (8th May 2000)
* J2SE 1.4 (6th Feb 2002)
* J2SE 5.0 (30th Sep 2004)
* Java SE 6 (11th Dec 2006)
* Java SE 7 (28th July 2011)
* Java SE 8 (18th March 2014)
* Java SE 9 (21st Sep 2017)
* Java SE 10 (20th March 2018)
* Java SE 11 (25th Sep 2018)

Java is a high-level object-oriented programming language; high level, in the context of languages, means human understandable. Since java’s syntax resembles the words and symbols that are commonly used by humans, it is called as high-level programming language. As everything is pursued as object in java, java is high-level object-oriented programming language.

## Java Features:

Simple:

Java is simple to learn and develop. The syntax is based on C++. Java doesn’t include any complicated and rarely used features like operator overloading which was used in native language (C). There is no need to explicitly remove unreferenced objects as there is an automatic garbage collection.

Object Oriented:

Everything in java is considered as an object. It covers the basics of Object-Oriented Programming System.

Secured:

Java has Byte code verifier to check the code fragments for illegal code that violates access rights to objects. Java Security Manager that determines what resources a class can access.

###### Platform Independent:

Java code can be run on any platform. Java compiler converts the source code into byte code and byte code is not platform specific code. Java Virtual Machine that is installed in various platforms is responsible to convert the byte code into machine instructions.

Portable:

Java byte code can be carried in a pen drive or any portable devices and can be executed in any platform with the help of JVM.

Robust:

Java has strong memory management. There is an automatic garbage collector to manage disregarded objects.

Java’s exception handling and type check mechanism provides strong application support.

Performance:

Java is faster than other traditional interpreted programming languages because java byte code is “close” to

native code.

Distributed:

Java facilitates users to create distributed applications. Distributed applications are created using Remote Method Invocation (RMI) and Enterprise Java Beans (EJB).

Interpreted:

Java byte code is translated on the fly to native machine instructions.

Multi-Threaded:

We can write java programs to deal with many tasks at once by defining multiple threads. As threads shares

common area, it doesn’t occupy much memory.

Dynamic:

Classes are loaded dynamically by class loader. Java also supports functions from native languages like C & C++. Java supports dynamic compilation.

Architecture – Neutral:

There are no implementation dependent features. That is the feature are not architecture specific. For example, size of primitive types is fixed. Primitive integer size is always 4 bytes in java irrespective of the processor type, wherein in C, the size of integer changes based on the processor. It is 2 bytes in 32-bit processor and 4 bytes in 64-bit processor.

## Class

* It is a logical entity and doesn’t have any physical existence.
* It can also be defined as blue print using which objects can be created and hence without the class, objects cannot be created.
* It is also called as master copy as more than one objects can be created out of the same class.
* For the object that was created out of the class to have the states and behaviours, they are to be declared in the class to use them when an object is created.
* A class is also called as non-primitive data type.

In Java, class can have Blocks (Static/Instance), States (Data members), Constructors, Behaviours (Methods), Nested classes (Class inside another class). These are all called members of the class.

How to create a class?

**Syntax:**

**class <Class – name>**

**{**

**}**

|  |
| --- |
| Class Name |
| + State 1: Type  + State 2: Type  + State 3: Type  + State 4: Type |
| + Method 1 (arg list): return  + Method 2 (arg list): return  + Method 3 (arg list): return  + Method 4 (arg list): return |

**Fig: UML Notation of the class**

## Java identifiers

To speak any language, there are certain rules. Similarly, to write a code in java, we also follow certain rules. We need to first identify how to call or address or refer or identify the components of java. Only then we can use those components to create a code that can be used to solve real world problems.

All the components of Java, such as classes, functions, variables, and so on must have a name, which we call the Java identifier.

There are four important points about Java identifiers that you need to keep in mind. Following are the identifier rules.

* Identifier must not contain white spaces.
* All identifiers should begin with a letter (A to Z or a to z), dollar character ($), or an underscore (\_).
* Identifiers cannot be Java keywords or reserved words (refer to the next image).
* Identifiers should not begin with numbers but can have numbers.

Identifier Rules

Here are a few examples of Java identifiers:

**Legal identifiers:**

Salary \_salary $salary salary\_month salary$Java Main

**Following are illegal, because it starts with number**

123Test 1\_2Test 123 1$Develop

**Following are illegal, because it starts with other special characters**

\*salary #Salary @Box %Marks

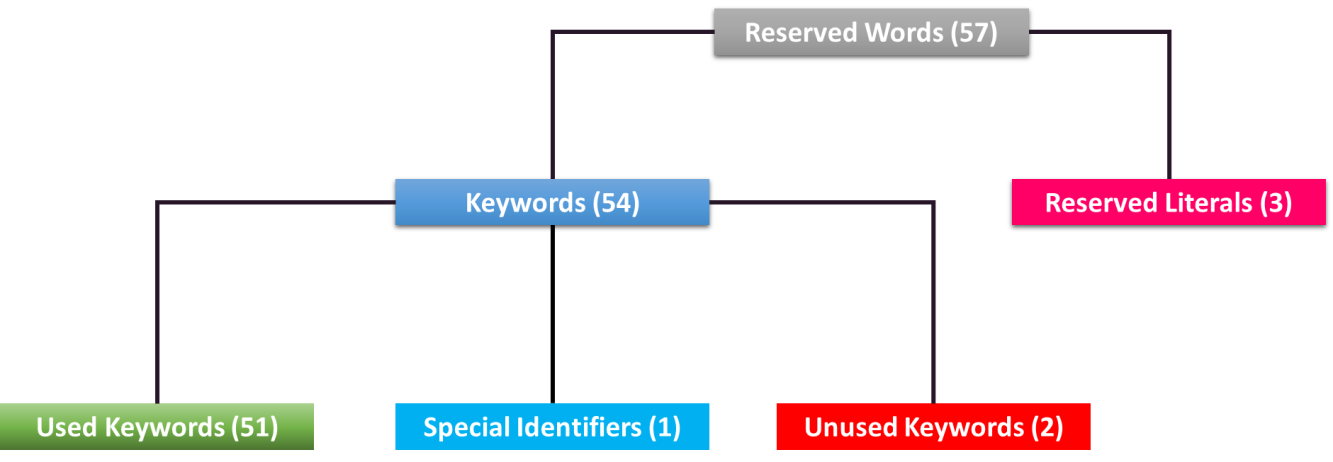
**Following are illegal, because it contains white space**

West Indies White Box Software Testing Java Dev

## Keywords:

In Java, many predefined words are already present (inbuilt) and they have some predefined meaning, which we call as Java keyword of reserved word.

The following are the Java keywords. These keywords must not be used as Java identifiers:



***Used Keywords (51)***

|  |  |
| --- | --- |
| *Keywords* | *Purpose* |
| ***abstract*** | Specifies that a class or method will be implemented later, in a subclass. |
| ***assert*** | Assert describes a predicate (a true–false statement) placed in a Java program to indicate that the developer thinks that the predicate is always true at that  place. If an assertion evaluates to false at run-time, an assertion failure results, which typically causes execution to abort. |
| ***boolean*** | A data type that can hold True and False values only. |
| ***Break*** | A control statement for breaking out of loops |
| ***Byte*** | A data type that can hold 8-bit data values |
| ***Case*** | Used in switch statements to mark blocks of text |
| ***Catch*** | Catches exceptions generated by try statements |
| ***Char*** | A data type that can hold unsigned 16-bit Unicode characters |

|  |  |
| --- | --- |
| *Keywords* | *Purpose* |
| ***class*** | Declares a new class |
| ***continue*** | Sends control back outside a loop |
| ***default*** | Specifies the default block of code in a switch statement |
| ***do*** | Starts a do-while loop |
| ***double*** | A data type that can hold 64-bit floating-point numbers |
| ***else*** | Indicates alternative branches in an if statement |
| ***enum*** | A Java keyword used to declare an enumerated type. Enumerations extend the base class. |
| ***exports*** | Used in modular java to export a package with a module. This keyword is only available in Java 9 and later. |
| ***extends*** | Indicates that a class is derived from another class or interface |
| ***final*** | Indicates that a variable holds a constant value or that a method will not be overridden |
| ***finally*** | Indicates a block of code in a try-catch structure that will always be executed |
| ***float*** | A data type that holds a 32-bit floating-point number |
| ***for*** | Used to start a for loop |
| ***if*** | Tests a true/false expression and branches accordingly |
| ***implements*** | Specifies that a class implements an interface |
| ***import*** | References other classes |
| ***instanceof*** | Indicates whether an object is an instance of a specific class or implements an interface |
| ***int*** | A data type that can hold a 32-bit signed integer |
| ***interface*** | Declares an interface |
| ***long*** | A data type that holds a 64-bit integer |
| ***module*** | The module keyword is used to declare a module inside of a Java application. This keyword is only available in Java 9 and later. |
| ***native*** | Specifies that a method is implemented with native (platform-specific) code |
| ***new*** | Creates new objects |
| ***package*** | Declares a Java package |
| ***private*** | An access specifier indicating that a method or variable may be accessed only in  the class it’s declared in |
| ***protected*** | An access specifier indicating that a method or variable may only be accessed in  the class it’s declared in (or a subclass of the class it’s declared in or other classes  in the same package) |
| ***public*** | An access specifier used for classes, interfaces, methods, and variables indicating that an item is accessible throughout the application (or where the  class that defines it is accessible) |
| ***requires*** | Used to specify the required libraries inside of a module. This keyword is only available in Java 9 and later. |
| ***return*** | Sends control and possibly a return value back from a called method |
| ***short*** | A data type that can hold a 16-bit integer |
| ***static*** | Indicates that a variable or method is a class method (rather than being limited to one particular object) |
| ***strictfp*** | A Java keyword used to restrict the precision and rounding of floating-point calculations to ensure portability |
| ***super*** | Refers to a class’s base class (used in a method or class constructor) |

|  |  |
| --- | --- |
| *Keywords* | *Purpose* |
| ***switch*** | A statement that executes code based on a test value |
| ***synchronized*** | Specifies critical sections or methods in multithreaded code |
| ***this*** | Refers to the current invoking object in a method or constructor |
| ***throw*** | Creates an exception |
| ***throws*** | Indicates what exceptions may be thrown by a method |
| ***transient*** | Specifies that a variable is not part of an object’s persistent state |
| ***try*** | Starts a block of code that will be tested for exceptions |
| ***void*** | Specifies that a method does not have a return value |
| ***volatile*** | Indicates that a variable may change asynchronously |
| ***while*** | Starts a while loop |

***Special Identifier:***

|  |  |
| --- | --- |
| *Keywords* | *Purpose* |
| ***var*** | A special identifier that cannot be used as a type name (since Java 10). |

***Unused:***

|  |  |
| --- | --- |
| *Keywords* | *Purpose* |
| ***const*** | Although reserved as a keyword in Java, const is not used and has no function. |
| ***goto*** | Although reserved as a keyword in Java, goto is not used and has no function. |

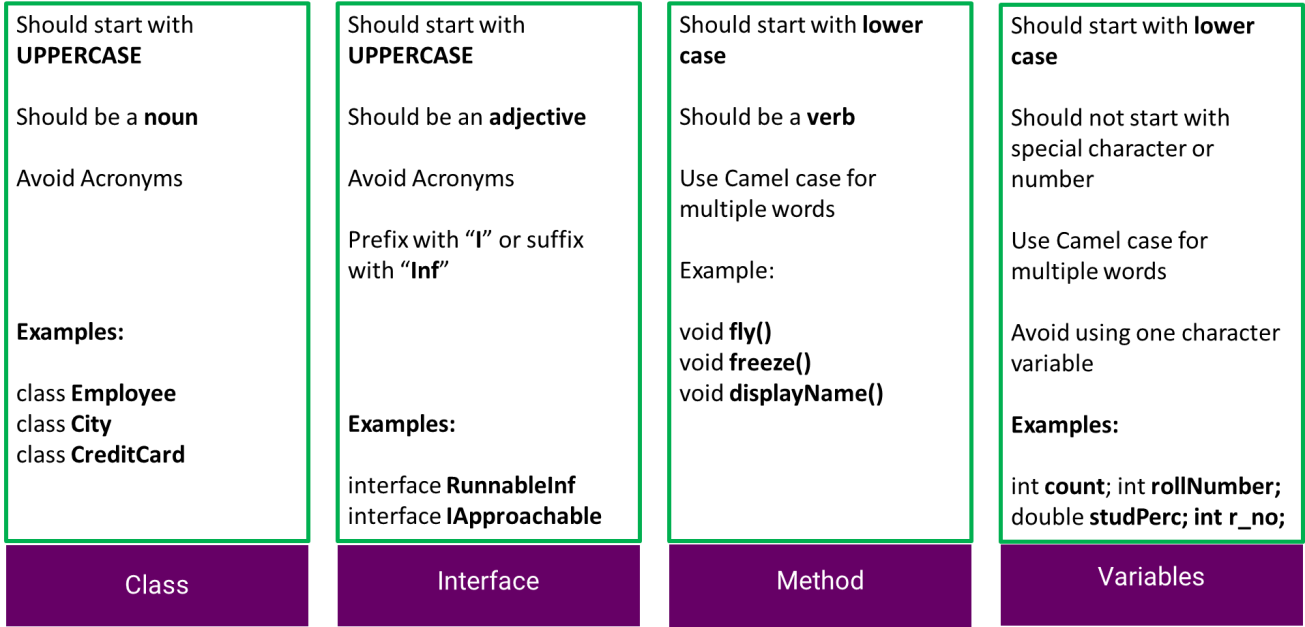
***Reserved literal values:***

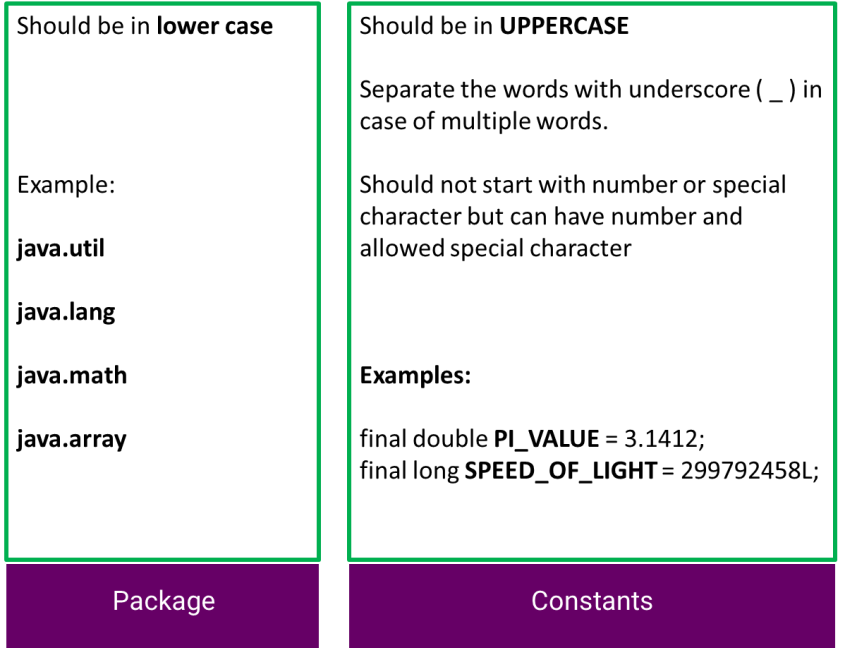
|  |  |
| --- | --- |
| *Keywords* | *Purpose* |
| ***true*** | A boolean literal value. |
| ***null*** | A reference literal value. |
| ***false*** | A boolean literal value. |

All the keywords are in lowercase in java.

**Java is case sensitive**

Naming conventions:



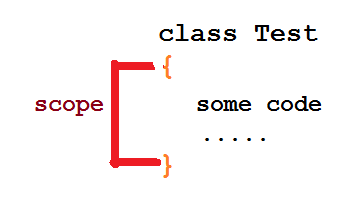


###### Fig: Naming conventions in Java

It is important to remember, when we write a program in Java, class name and the filename, while saving the file, should be the same, and the filename should have the .java extension.

## Java compilation

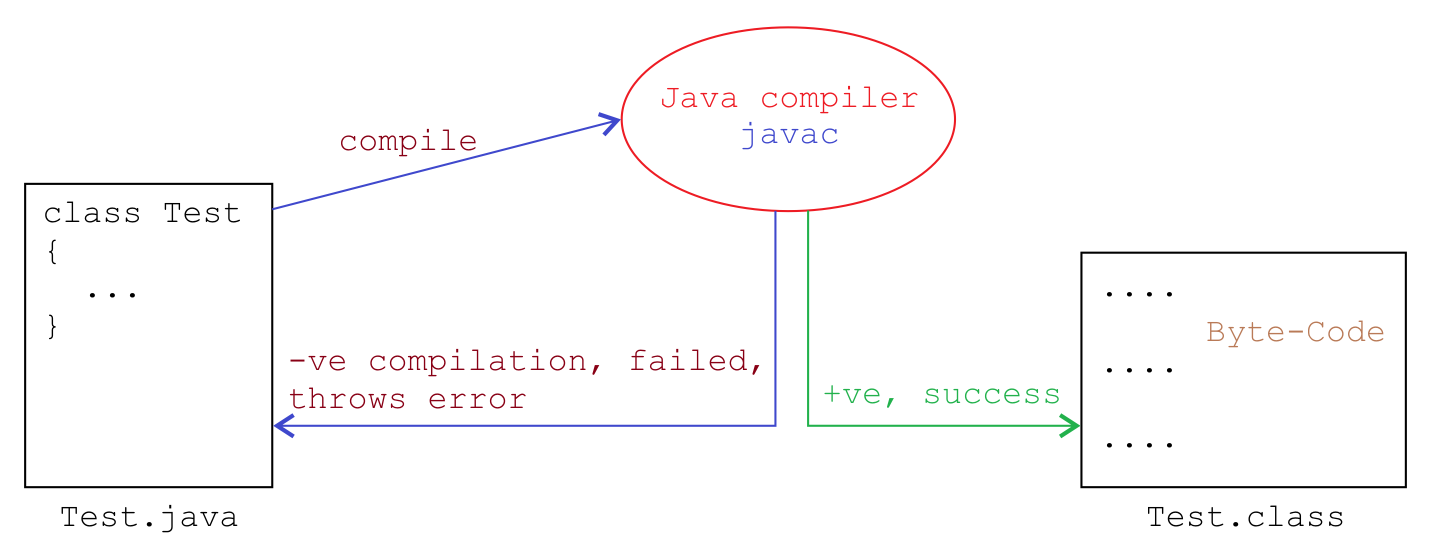
Now, examine the following image:



###### Example: Test.Java

The Javac is the Java compiler, which does two things:

* + It checks the syntax (grammar) of the Java program, and if the program is not syntactically correct, then it throws a compilation error. We say that the compilation has failed.
  + If the compilation is successful, then the compiler generates the bytecode (.class program) Observe the following diagram:



###### Fig: Execution of Java Code

When our source Java code (Test.Java) is compiled using Javac and no syntax errors are found, then Javac produces an executable file (Test.class), which is the bytecode. The extension of bytecode is .class. Bytecode is platform-independent and only JVM can understand the bytecodes.

## Byte Code – What it is?

Let’s see what byte-code means; it is intermediate code produced by Javac (the Java compiler) upon successful compilation. Bytecode is an intermediate code or an intermediate set of instructions, which is neither a high- level language nor a low-level language (that is, machine level language, consisting of only 0sand 1s).

## How did Java become platform-independent?

The feature of platform independence of Java is considered as Write Once and Run Anywhere (WORA). To understand the concept of WORA, let’s consider an example. Suppose a novel in Punjabi has to be published in different European languages. In such a case, we’ll first have to translate the novel from the Punjabi language to one common language (say English). We can then translate it to a different language, since directly translating Punjabi to any European language is quite challenging.

Similarly, in Java, when we write the program initially, it would be in high level language (Java). It has to be then converted into a common language for all the systems to understand (byte codes). Then these byte codes can be converted into the respective machine level language (operating system or platform)

The process of converting Java language into machine level language is explained as follows:

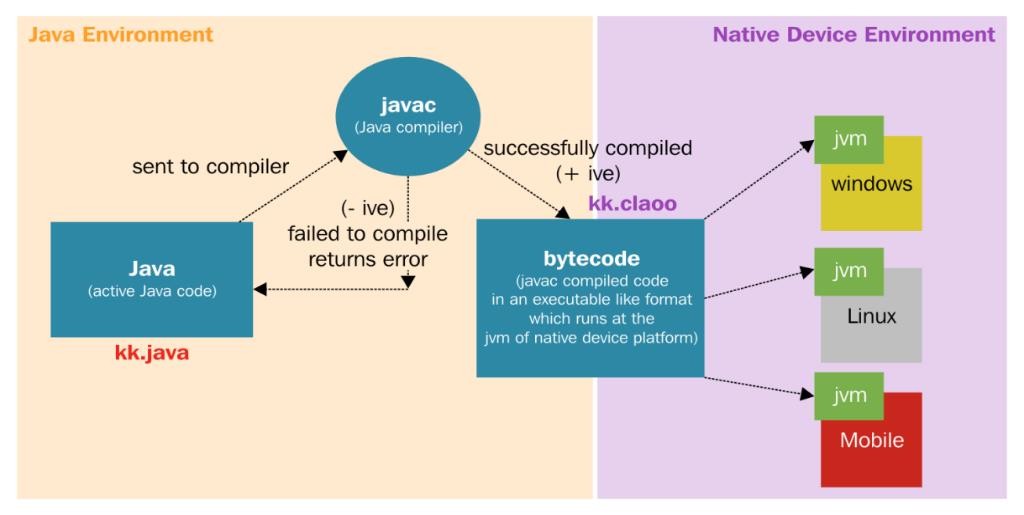
When we write any Java program, let’s say Test.Java (high level language), it is converted to Test.class (byte codes), which can be executed on any platform. There are two steps involved while converting the high-level language into machine level language:

1. Compilation
2. Execution

## Java execution

After the successful compilation (when the .class file is produced) JVM converts the bytecodes into the machine-level language—it interprets the bytecode so that the Java program can be executed to perform the associated tasks on a native machine (Windows machine, in this example).

It is because of the bytecode and JVM that Java is platform-independent!



###### Fig: Execution of Java Code

## Java Development Kit:

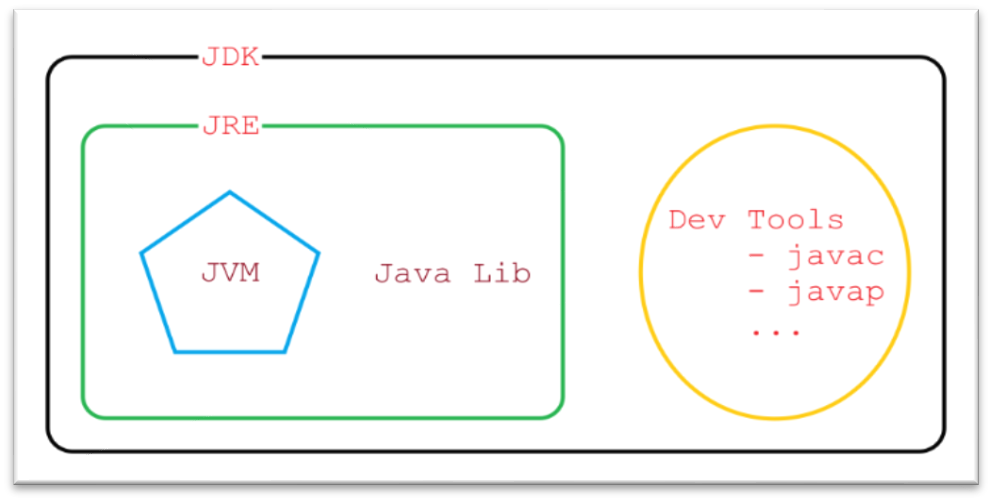
* Java Development Kit (JDK) is a software kit, containing all the necessary resources required to develop and execute Java programs. JDK internally has a Java Runtime Environment (JRE)and other tools such as Javac (Java compiler).
* It is a software development environment which is used to develop java applications.
* It contains a private Java Virtual Machine and few other resources such as interpreter/loader, compiler, documentation generator which are used to develop a java application

## Java Runtime Environment:

* The Java Runtime Environment (JRE) is a set of software tools which are used for executing java applications.
* It is used to provide runtime environment. It is the implementation of Java Virtual Machine.
* It contains set of libraries and other files that Java Virtual Machine uses at runtime.
* It can be downloaded separately.

## Java Virtual Machine:

* The Java Virtual Machine (JVM) is a virtual machine or platform (software only) that sits on top of an actual platform (operating system and processor).
* It is an engine or environment for running or executing .class files (bytecode).
* As the name indicates, JVM is a virtual machine (but not a physical component) which is a set of instructions written in C program.
* Every operating system has its own JVM and JVM is platform dependent, that is, the JVM of Linux is different from the JVM of mac OS, which is different from the JVM of Windows.
* It is a specification that provides a runtime environment in which java byte code can be executed.
* Converts java byte code into machine language or machine instructions.
* Responsible for allocating memory space.
* Can also run those programs that are written in other languages and are compiled to java byte code.



###### Fig: Java Development Kit

JDK can be downloaded from the following website; it’s free and open source:

<https://www.oracle.com/technetwork/java/javase/downloads/index.html>

###### JDK installation

The installation of JDK is very simple. After the installation, we need to set the path. The steps to set the path for Windows 10 and Windows 8 are as follows:

1. In Search, search for environment and then select Edit the system environment variables
2. Click the Environment Variables… button.
3. In the System variables section, find the PATH environment variable and select it. Click Edit. If the PATH environment variable does not exist, click New.
4. In the Edit environment variable (or New System Variable) window, specify the value of the PATH environment variable. Click OK. Close all the remaining windows by clicking OK.
5. Open the command prompt window and run your Java code.

###### Setting the path for mac OS:

To run a different version of Java, either specify the full path or use the Java\_home tool:

% /usr/libexec/Java\_home -v 1.8.0\_73 --exec Javac -version

###### Variants

JDK has different versions and the latest version is 11.0 or jdk 11.0. Also, Java has different editions, such as:

**Standard Edition - Java SE:** This edition provides the core functionalities for developing a desktop application, such as networking, security, database access, Graphical User Interface (GUI) development, and so on

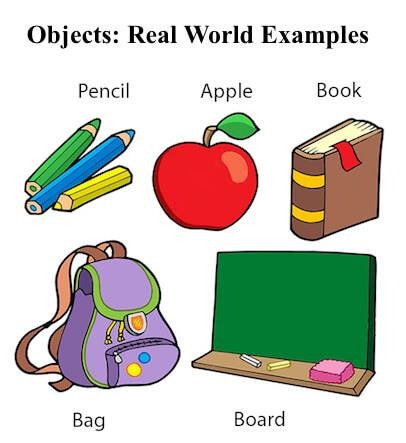
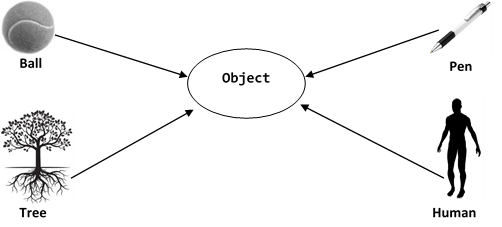
**Enterprise Edition - Java EE:** This edition is built on top of Java SE and is used for large-scale, multi-tiered, and secured network enterprise application

**Micro Edition - Java ME:** This edition is used for creating applications that run on embedded and mobile devices

Note that the Java is platform-independent. We can run it on different operating systems and different hardware, such as Windows, mac OS, Linux, mobile, ticket machines, ATMs, microwave, cars, and so on. So, Java is not just OS-independent but, in fact, platform-independent (that is, it can run on different OS that run on different machines).

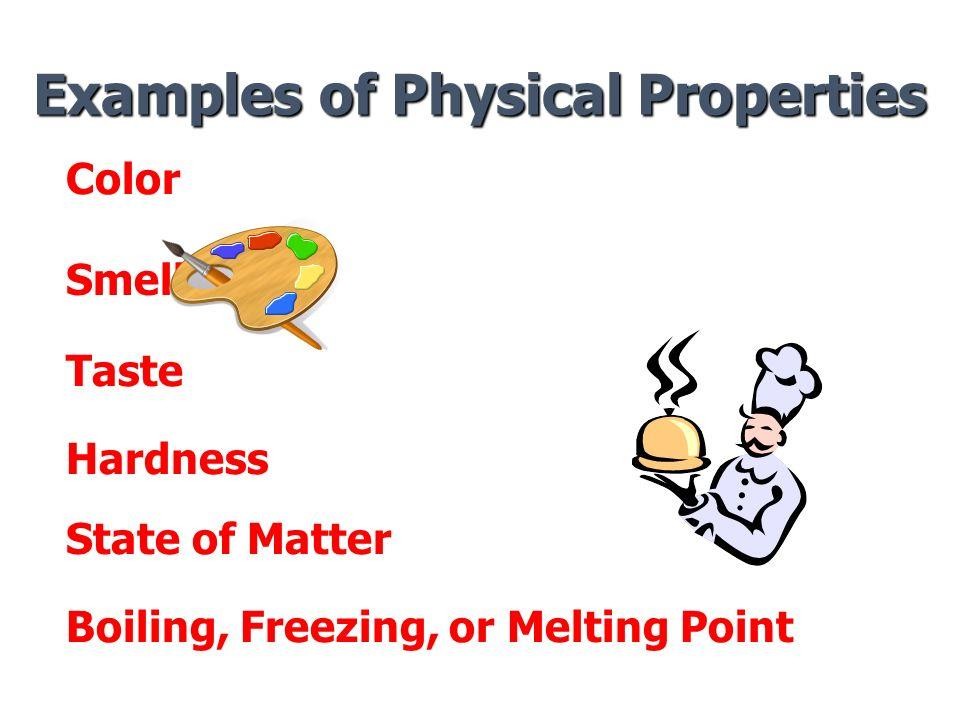
Object

Any real-world entity that is physically present or which can be experienced or touched or seen or felt and that which has defined states and behaviours and has mass and occupies space is called as an object.



###### Fig: Real World Objects

* ***States*** are the one that describes about the properties of the object
* ***Behaviors*** are the one that describes the functionality of the object, in other words the actions that are performed by the object.



**Fig: Properties of an object**

Let’s take a dog object. The actions that can be performed by the dog are as follows.



**Fig: Behaviors of the dog object**

* Object is also called as instance or specimen of a class.
* Objects cannot be created without the class.
* There can be more than one reference to an object.
* Multiple objects that are created from same class are called as “Similar Objects”.
* Changes in one object neither affects other objects nor the class.
* Software objects are often used to model real-world objects.

## States and behaviours in Java:

States are the one that describes about the properties of the object. For example, few properties of pen are colour of the pen, type of the pen, price of the pen. These states are referred to as data members in java.

Behaviours are the one that describes the functionality of the object, in other words the actions that are performed by the object. These behaviours are referred to as methods in java.

When we say data members, we have to define what type of data it stores. For example, colour of the pen and price of the pen cannot be of same type. Data that is stored as states have different types which is called as data types.

## In IT, I (information) matters a lot

If we observe, most of the applications deal directly or indirectly with information or data. For example, the Google search engine, job portals, matrimony websites, e-commerce websites, and so on, all deal with data. Hence, any programming language must support data storage and data usage while programming.

In the real world, we see data in the following categories/types:

Numeric data:

* Decimal data Example formats are:

height=6.2 weight = 32.5 temperature=-14.6

* Non-decimal data Example formats are:

age=23 salary = 85000 price=50

Non-numeric data or text data

Example: name = “Narendra Modi” place = “Bangalore” initial=‘A’

In Java or in any other programming language, data must be mentioned with its type, that is, what type of data/information it is. Now let’s see what data types are. Data types define or specify the type and size of the data.

In Java, we have two different types of data:

1. Primitive data type
2. Non-primitive, reference, or custom data type

## Primitive data type

Primitive data types are the basic in-built data types in Java. Data type is used for the classification of the type of data which is stored in the variables mentioned.

Primitive data type is used to declare raw data which holds the numeric values and is directly understood by the system.

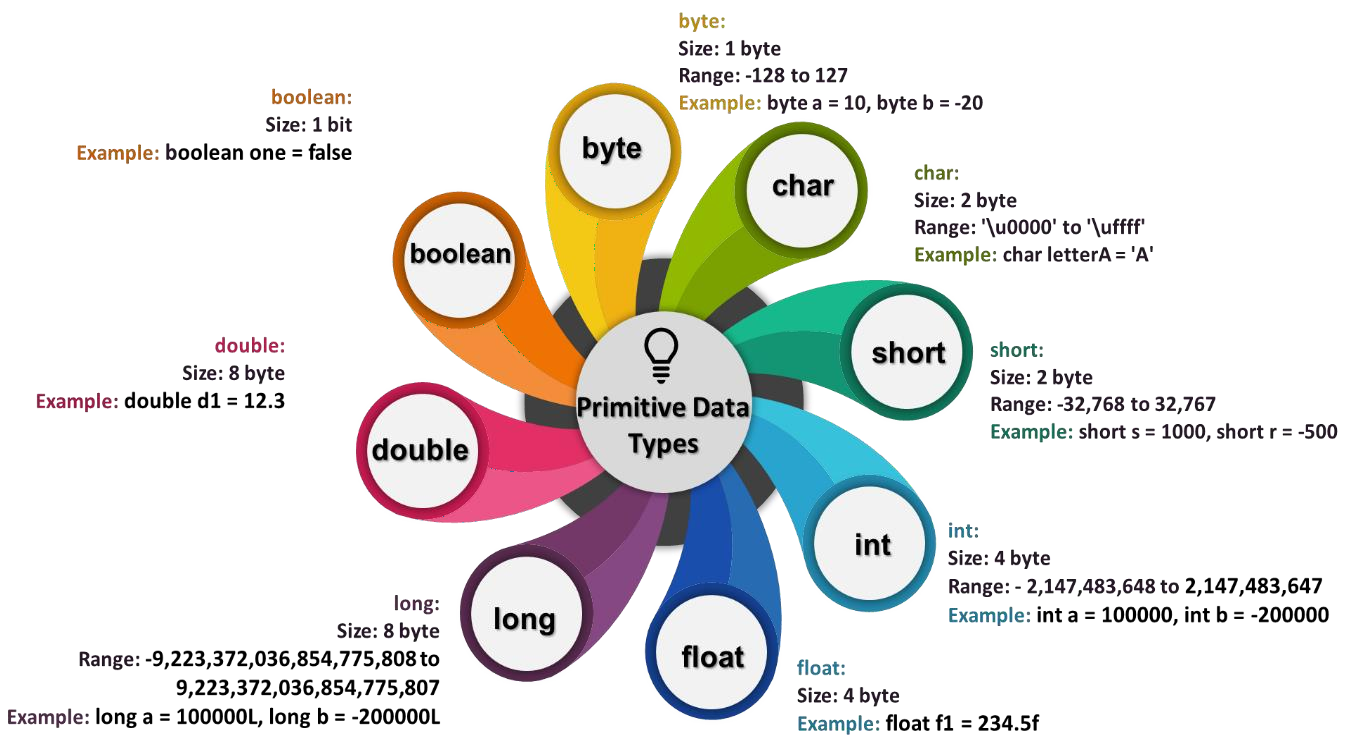
Range:

Each data type supports or accommodates a range of values which has minimum and maximum limit

Default values:

Data types in Java are initialized to default values when the variables are declared as class members or instance variables. The table, lists primitive data types available in Java along with the range of values it supports and the default value of each data type:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Numeric | Range | Size/Capacity | Default Value | Example |
| **Byte** | -128 to 127 | 8 bits / 1 byte | 0 | byte a = 10, byte b = -20 |
| **Short** | -32768 to 32767 | 16 bits / 2 bytes | 0 | short s = 1000, short r = -500 |
| **Int** | -2,147,483,648 (-2 ^15) to  2,147,483,647 (2^15-1) | 32 bits / 4 bytes | 0 | int a = 100000, int b = -200000 |
| **Long** | -9,223,372,036,854,775,808 to (-2^63)  9,223,372,036,854,775,807 to (2^63-1) | 64 bits / 8 bytes | 0 | long a = 100000L, long b  = -200000L |
| **Float** | ± 3.40282347E+38F | 32 bits / 4 bytes | 0.0f | float f1 = 234.5f, float f2 = 12.34f |
| **Double** | ± 1.79769313486231570E+308 | 64 bits / 8 bytes | 0.0d | double d1 = 12.3, double d2  = 345.23 |
|  |  |  |  |  |
| **Other** |  |  |  |  |
| **Boolean** | true or false | 1 bit | false | boolean b = false, boolean c  = true |
| **Char** | '\u0000' to '\uffff' | 16 bits / 2 bytes | \u0000 | char ch = 'A',  char c = ‘a’ |



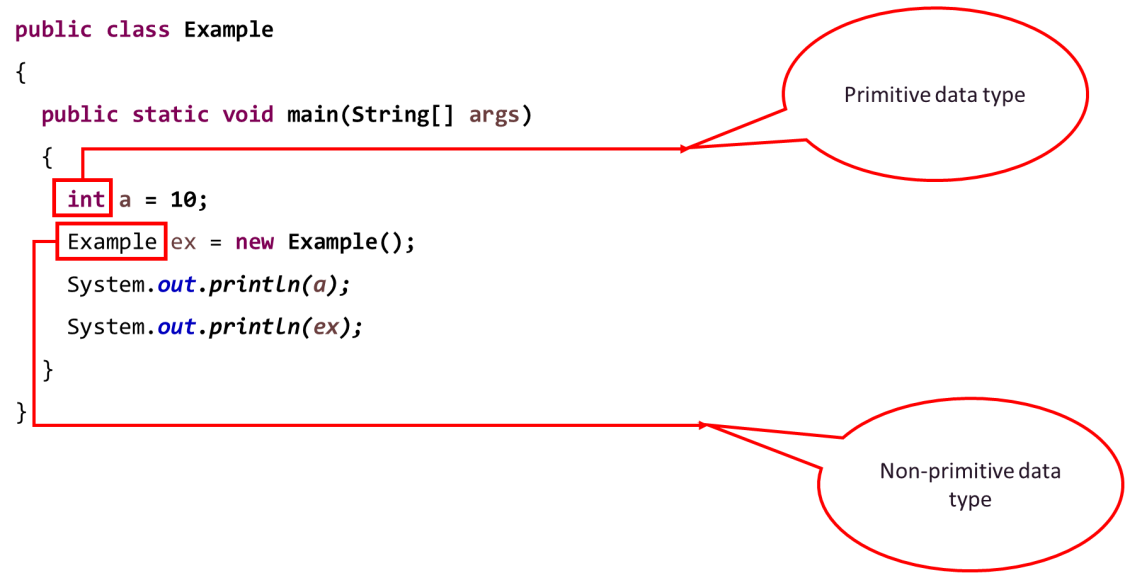
**Fig: Primitive Data Types**

## Non-primitive / Reference / Object type

The meaning of non-primitive data type is, the data type which is not created or in-built in the programming language but created by the programmer. These variables are often called as reference variables. These reference variables refer to the objects (location of the object in the memory)

String is an example (S upper case).

In further chapters String and Objects are discussed in detail along with reference variables.



## Data representation / data members:

After learning data and datatypes, let’s see how to represent the data.

There are two types of data:

1. Variable
2. Constant

## Variable

As the name indicates, a variable is data that may change often. For example, the human weight changes/varies. Here are a few examples of Java variables:

float weight = 62.7f; double height = 6.2; int age = 22;

The preceding variables are called as instance variables.

The following variable is considered as a class variable, which is explained in detail in the further sections:

static String address = “Bangalore, India”;

Weight, height, and age are all changeable entities; hence, they are called variables.

It is a good practice to follow the naming conventions while declaring the variables and the naming conventions are as follows:

* The variable name must start with lowercase letters
* There should be no space between words
* The second word must start with uppercase letters Check out the following example:

String firstName = “Shishira”;

int monthlySalary = 85000;

## Constant

The data/information that does not change is called constant. Here, are a few examples of constants:

* In mathematics, we use pie, which is 3.142
* In the real-world example, data of birth is a constant for a person
* Similarly, model number is a constant for a vehicle etc.

In Java, constants are declared using an access modifier or keyword, called final.

final double pie = 3.142; final String model = “Honda City”;

Note that it’s a good practice (not a rule) to declare the final data member in uppercase, as follows:

final double PIE = 3.142; final String MODEL = “Honda City“;

## Functions or Methods

In the real world, we perform many tasks, which is nothing but functionality. Programmatically, a task can be represented by something called *function/method.*

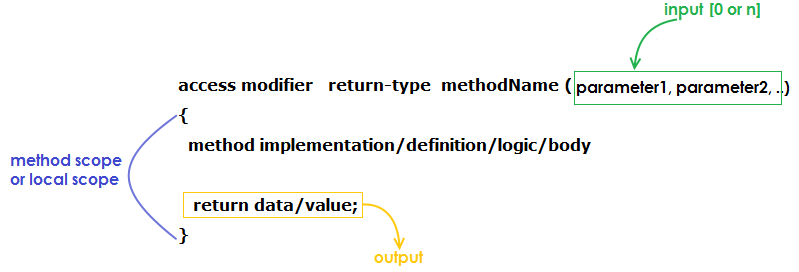
***Toast- So, method means work, task, action or some job.***

For example, sendMessage, sleep, switchOn, drive, run, eat, and so on. Oh! I forgot; drinkBeer is also a method.

Visualize. Sometimes, when we work, we may need an input or we may not. Sometimes, after some work, we may give some output or we may not.

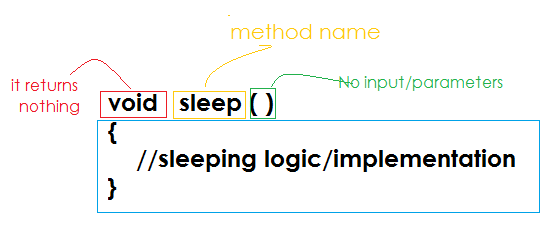
So, even in programming, when we write a method, we may give some input to the method or the method may sometimes give some output.

Here is the structure of a method:



Let’s understand the concept with some examples.

Imagine that sleep is a functionality that neither requires any input nor gives any output. So, it can be represented as follows:



This means the following:

* The void keyword means that the method does not return any data or output.
* This method does not take any parameters or input.
* This method does not give any output, which means there is no return statement.

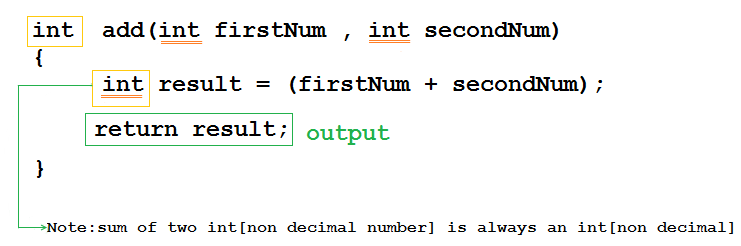
Methods always represent some action, and hence, the name of the method has to be a verb (in English, a verb represents action). This is one of the coding standards to remember while creating a method.

Method **PARAMETERS** means **INPUT**

Method **RETURN** statement means **OUTPUT**

The naming convention or the coding standard for a method must be lowerCaseToUpperCase: switchOn() openDoor() addNumber() displayInfo()

Here’s an example of a method that adds two numbers and returns the sum as a result:

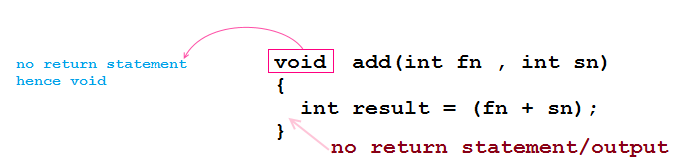


The preceding method’s name is add, which takes two inputs, that is, parameters, firstNum and secondNum. The type of both the inputs is int, that is, a non-decimal numeric.

The method’s logic is that it adds two numbers and the result is obviously int. It returns the result of the

addition of two numbers as the output.

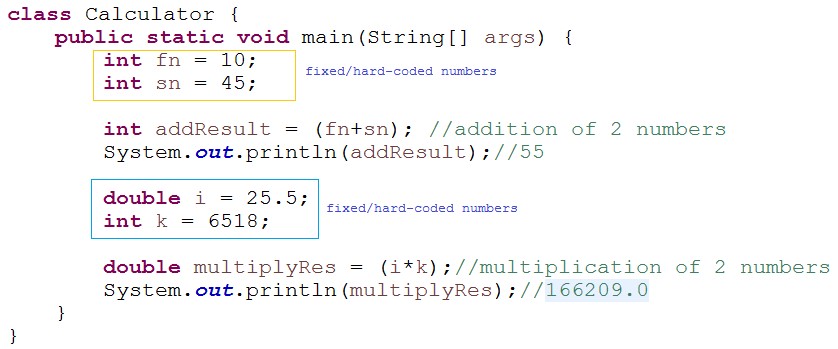
###### Any variable or parameter within the method scope is called local variable.

Here is an example of the same method without the output or return statement:

###### Mind sharpener

So far you have learned how to declare a method. However, what is the use of a method? Why do we require a method while programming?

Let’s understand the answers to the same with a small case study, which includes a small Java program for

adding and multiplying two numbers:

Let’s understand the preceding program in detail.

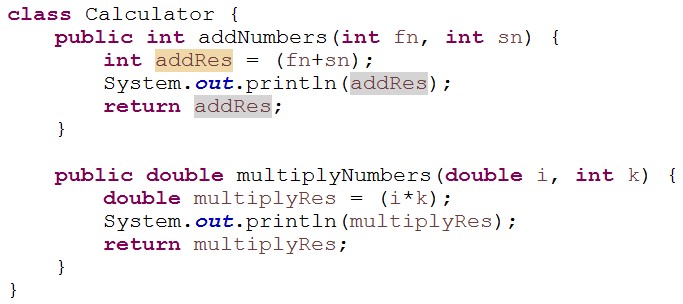
In the preceding program, we performed the addition of two numbers fn and sn and stored the result in addResult variable whose data type is int. Because both fn and sn are integers and addition of two integers is an integer. The result of the addition is printed as 55.

Similarly, we multiplied two numbers, i and k, and the result of multiplication gets stored in a variable multiplyRes whose data type is double and then prints the result. Note that the result is printed as 166209.0, but not 166209 because the datatype of the multiplyRes variable is double, which is a decimal value.

Now, let’s look at some disadvantages or problems of the program. The data is hardcoded or fixed. When we execute the program multiple times, it does the same thing—adds and multiplies. Imagine that you purchased an expensive calculator that always performs only addition and multiplication with two fixed numbers!! Would you appreciate that? Certainly, no. It is the same for our program.

What could we do to resolve this? We will need dynamic values (choice of values), wherein we must be able to perform addition, multiplication, or both, any number of times. Hence, the solution is to use methods.

The following is the solution program with two methods:



The preceding code has two methods—one adds two numbers and the other multiplies two numbers. Advantages:

* An individual method can be executed
* While executing the method, dynamic values can be passed
* Methods can be executed any number of times with different inputs each time
* Methods can be reused in the entire project, thereby increasing code reusability But how do we execute the methods? How do we pass values for input parameters?

Relax. We can execute any method only by invoking it or calling it. That is, a method can’t be executed on its

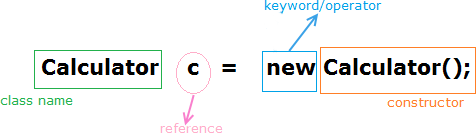
own! Rather, it must be explicitly called. For calling the method, we need an object.

## Object Creation

Objects are created and managed by JVM. Objects are created in heap memory. There are 4 different ways to create an object

* Using new keyword.
* Using Class.forName()
* Using clone ()
* Using object deserialization

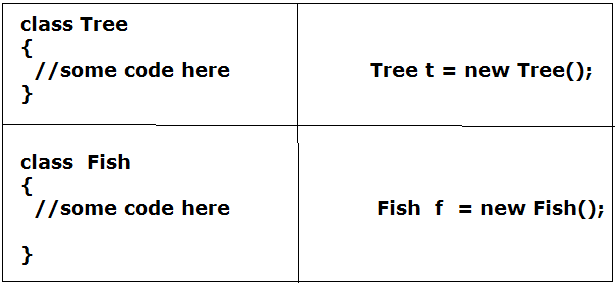
The addNumbers and multiplyNumbers methods belong to the Calculator class; hence, we need to create an object of the Calculator class. Check out the following code:



***The class name and the constructor name must be the same.***

The new operator instructs that a new object or a new memory location for the object must be created. The variable c is the reference (or reference variable). It can be any name (but not a keyword).

Here are a few more examples:

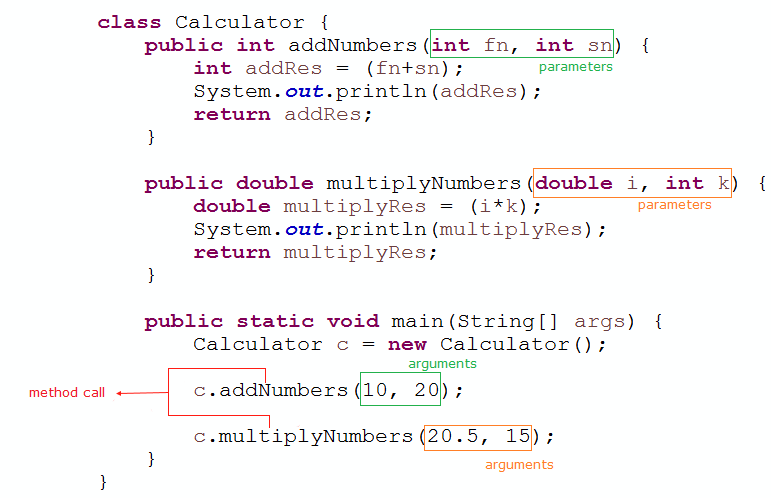


## Calling the method

The method of a class can be called within the main method and outside the class by using the dot (.) operator and an object reference.

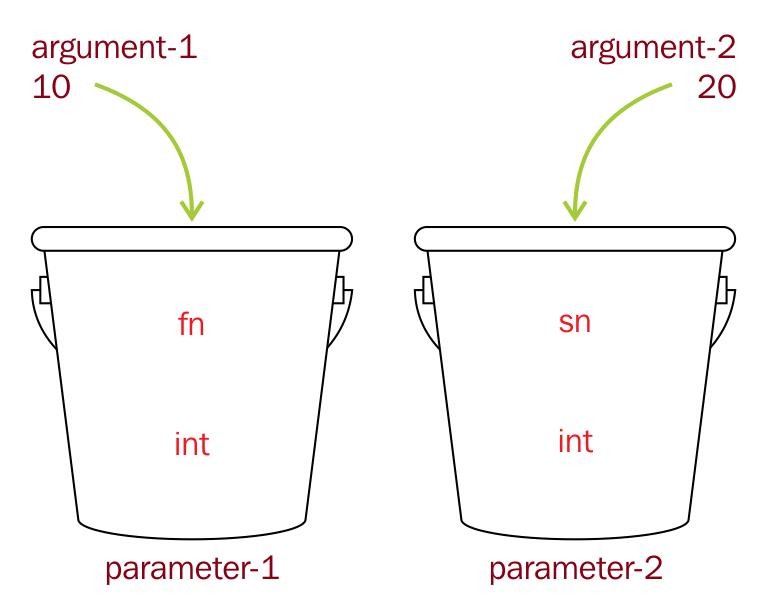
Note that static methods can be called directly with the class name, which is explained in the static sections in further chapters.

Examine the following code:



The main method calls the addNumbers and multiplyNumbers methods by passing some inputs/arguments. A parameter is like a variable that can store or hold a value, and an argument is the actual value or data.

Examine the following image:



***The number of parameters and the number of arguments must be the same. The data types of the parameter and the argument must be the same.***

## Main method

A main is also a method which is needed for executing the program in Java. Note that main method is a static method (static will be explained later) and if we want to call a method or variable from main method we should always create object first and call the members using the object reference, even if the main method is present in the same class.

## Anatomy of main method

Main method contains different keywords such as the following:

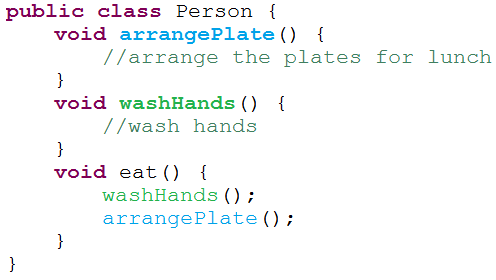
* public - which means it can be accessible everywhere, without any restriction
* static - indicates that the method belongs to the class where it is present
* void - main method simply executes the program but does not return anything
* main - name of the method

Main method has a parameter - String[] args: which means that the method accepts series of data (or array), [] represents array and args is the reference variable

Each of these components are explained in detail in the further chapters

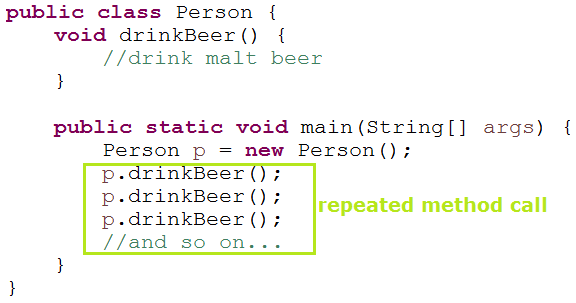
One more thing: a class’s method can call another method of the same class directly by its

name, without the dot (.) operator(main method is an exception for this as we have seen)

For example, look at the following image:

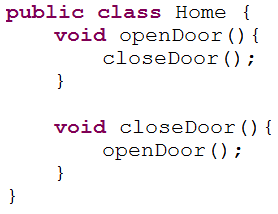
The arrangePlage, washHands, and eat methods belong to the same class. Hence, the eat method can call/invoke the washHands and arrangePlate methods directly. This can also be called method chaining.

A method can be called multiple times, repeatedly. Observe the following image:



Now, let’s see the tricky scenario of recursive methods—a method calls another method and that method calls the first method back, which is a never-ending story.

See the following image:



When we run the preceding program, we’ll get the following error:

Java.lang.StackOverflowError

Finally, that’s the end of methods.

Let us now see how we can use Java to model the real-world objects. But before we get into that we need to understand few more things about java. Java is called as platform independent programming language. That is, we can develop the software using java and it can be run on anything and everything. Curious to know how? Let’s explore it how java is so powerful that it provides us with such a breath-taking feature.

A simple Java Program

In Java programming, everything starts with a class.

Here’s an example of valid Java code, showing the class structure:

public class ClassName { variables

constructors methods

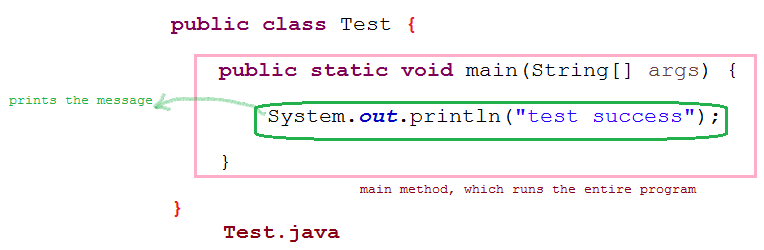
}

Java is a case-sensitive language; that is, class is not the same as Class.

A class starts with the keyword, class, and the name of the class. Inside the body of the class, the first section is for variables. Constructors and methods are the other prominent members of a class. Each of these will be explained in detail as we proceed with further concepts.

## Compiling a program

Let’s practically create, compile, and run a Java program. See the following image:



Study the following code explanation:

* The class name is Test, which is saved as Test.Java.
* The main method is like an engine. In Java, the program execution starts with the main method, without which the program does not run.
* The test success message is printed using System.out.println.
* JVM calls the main method.
* System.out.println is used to print some content.
* System represents a standard system resource. It’s an inbuilt class.
* The out represents the output (details will be discussed later).
* The println method is used to print the content.

Note that In a Java program, single or multiple lines of code can be commented on. The commented code is not considered by the compiler:

//single line comment

/\*

Multi line comment

\*/

## Literals in Java

Literals are the constant values that are assigned to a variable. In java, literals are of 5 types namely,

* 1. Integer Literals
  2. Floating Point Literals
  3. Character Literals
  4. String Literals
  5. Boolean Literals

## Integer Literals

We can specify the integer literals in 4 different ways.

1. Binary
2. Octal
3. Decimal
4. Hexadecimal

## Binary Integer Literals

Digits 0 & 1 only are allowed. It should always have a prefix 0b.

**Example:** int x = 0b1010; int y = 0B0101;

### Example Program:

##### public class Example {

##### public static void main(String[] args) { int a = 0b1010;

##### int b = 0B0101; System.out.println(a); System.out.println(b);

##### }

##### }

### Output:

##### 10

##### 5

## Octal Integer Literals

Digits from 0 – 7 are allowed. It should always have a prefix 0.

**Example:** int x = 0146; int y = 010;

### Example Program:

##### public class Example

##### {

##### public static void main(String[] args)

##### {

##### int a = 010; System.out.println(a);

##### }

##### }

### Output:

##### 8

## Decimal Integer Literals

Digits from 0-9 are allowed in this form.

**Example:** int x = 146; int y = 10;

### Example Program:

##### public class Example

##### {

##### public static void main(String[] args)

##### {

##### int a = 10; System.out.println(a);

##### }

##### }

### Output:

##### 10

## Hexadecimal Integer Literals

Digits from 0 – 9 and A – F are allowed. It should always have a prefix 0x.

**Example:** int x = 0x2FACE; int y = 0xfeed;

### Example Program:

##### public class Example

##### {

##### public static void main(String[] args)

##### {

##### int a = 0x2FACE; int b = 0xfeed;

##### System.out.println(a); System.out.println(b);

##### }

##### }

### Output:

##### 195278

##### 65261

## Floating Point Literals

Accepts decimal, Octal, binary and hexadecimal values without decimal point and in this case, the value will be stored with extra .0 to indicate floating point

In case of decimal point, the suffix f is mandatory as any value without the suffix f is considered as double.

In case of character, assigned to floating point, the ascii equivalent of the character with .0 will be stored as floating-point value.

**Example:** float flo = 3453.34f; float dflo = 34521;

### Example Program:

##### public class Example

##### {

##### public static void main(String[] args)

##### {

##### float dec = 78934, oct = 010, bin = 0b0101, hex = 0xface, cha = 'a’, flo = 345623.345f; System.out.println(dec); System.out.println(oct); System.out.println(bin); System.out.println(hex); System.out.println(cha); System.out.println(flo);

##### }

##### }

### Output:

##### 78934.0

##### 8.0

##### 5.0

##### 64206.0

##### 97.0

##### 345623.34

## Character Literals

We can specify the character literals in 4 different ways

1. Single Quote
2. Integer, Oct, Hex, Bin
3. Escape Sequence
4. Unicode Representation

## Single Quote Character Literals

Java Literal can be specified to char data type as a single character within a single quote.

**Example:** char ch = ‘A’; char c = ‘a’; char cha = ‘0’

### Example Program:

##### public class Example

##### {

##### public static void main(String[] args)

##### {

##### char ch ='a’; char c ='A’; char cha = '0’;

##### System.out.println(ch); System.out.println(c); System.out.println(cha);

##### }

##### }

### Output:

##### a A 0

## Integer, Octal, Hexadecimal and Decimal Character Literals

Any integer value, binary, octal and hexadecimal value can be assigned to a character. It basically converts the literal into corresponding ascii value and display the character equivalent of it.

**Example:** char decmax =65535; char decmin = 0; char oct = 0153;

### Example Program:

##### public class Example {

##### public static void main(String[] args) {

##### char decmax =65535, char decmin = 0, char oct = 0153, bin = 0b1110101, hex = 0xe9; System.out.println(decmax); System.out.println(decmin);

##### System.out.println(oct); System.out.println(bin); System.out.println(hex); }

##### }

### Output:

##### ?

##### k u é

## Escape Sequence Character Literals:

Char literals in Java can be specified using escape sequence. A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler.

**Example**: char ch6 = '\‘’; char ch7 = '\“’; char ch8 = '\\’;

|  |  |
| --- | --- |
| **Escape Sequence** | **Description** |
| \t | Insert a tab in the text at this point. |
| \b | Insert a backspace in the text at this point. |
| \n | Insert a newline in the text at this point. |
| \r | Insert a carriage return in the text at this point. |
| \f | Insert a form feed in the text at this point. |
| \' | Insert a single quote character in the text at this point. |
| \" | Insert a double quote character in the text at this point. |
| \\ | Insert a backslash character in the text at this point. |

### Example Program:

##### public class Example

##### {

##### public static void main(String[] args)

##### {

##### char ch6 = '\‘’;

##### char ch7 = '\“’; char ch8 = '\\’;

##### System.out.println('\t'+ "Welcome"); System.out.println(ch6); System.out.println(ch7); System.out.println(ch8);

##### }

##### }

### Output:

##### Welcome

##### '

##### "

##### \

## Unicode Representation:

Char literals in Java can be specified in Unicode representation ‘\uxxxx’. Here XXXX represents 4 hexadecimal

numbers.

**Example:** char ch = '\u00ad’; char chad = '\u001a';

### Example Program:

##### public class Example {

##### public static void main(String[] args)

##### {

##### char chad = '\u0070'; char ch = '\u00ad'; char cha = '\u0061';

##### System.out.println(chad); System.out.println(ch); System.out.println(cha);

##### }

##### }

### Output:

##### P

##### -

##### a

## String Literals:

String literals in Java are any sequence of characters with a double quote. In case of escape sequences, they have to be specified in the expected format.

**Example**: String s = “Hi”; String str = “Welcome”

### Example Program:

##### public class Example

##### {

##### public static void main(String[] args)

##### {

##### String s = "Welcome"; String name = "Welcome"

##### + " to"

##### + " Bangalore";

##### System.out.println("\' Example of char inside string"); System.out.println("\\");

##### System.out.println(s); System.out.println(name);

##### }

##### }

### Output:

##### ' Example of char inside string

##### \ Welcome

##### Welcome to Bangalore

## Boolean Literals:

Java Boolean literals allow only two values i.e. true and false.

**Example**: boolean bo = true; boolean bool = false;

### Example Program:

##### public class Example

##### {

##### public static void main(String[] args)

##### {

##### boolean b = true; boolean c = false;

##### //boolean d = 0; //Type mismatch: cannot convert from int to boolean

##### //boolean b = 1; //Type mismatch: cannot convert from int to boolean System.out.println(b);

##### System.out.println(c);

##### //System.out.println(d);

##### //System.out.println(e);

##### }

*}****Output:*** *true false*

## Operators in java

Operator in java is a symbol that is used to perform operations. For example: +, -, \*, / etc. There are many types of operators in java which are given below:

* Arithmetic Operator
* Unary Operator
* Assignment Operator
* Relational Operator
* Logical Operator
* Ternary Operator
* Bitwise Operator
* Shift Operator
* instanceof Operator
* Operator precedence and Associativity

## Arithmetic Operator:

* Arithmetic operators require two operands to work.
* It is a binary operator.
* Arithmetic operator is used to perform simple arithmetic operations on primitive data types like:
  + \*: Multiplication
  + /: Division
  + %: Modulo

 +: Addition

* + –: Subtraction

### Example Program:

##### public class ArithmeticOperators {

##### public static void main(String[] args)

##### {

##### int a = 20, b = 10;

##### String x = "Hello", y = "World";

##### // + and - operator System.out.println("a + b = " + (a + b)); System.out.println("a - b = " + (a - b));

##### // + operator if used with strings

##### // concatenates the given strings. System.out.println("x + y = " + x + y);

##### // \* and / operator System.out.println("a \* b = " + (a \* b)); System.out.println("a / b = " + (a / b));

##### // modulo operator gives remainder

##### // on dividing first operand with second System.out.println("a % b = " + (a % b));

##### }

##### }

### Output:

##### a + b = 30 a - b = 10

##### x + y = HelloWord a \* b = 200

##### a / b = 2 a % b = 0

###### Note:

In case of whole numbers, if the numerator is smaller than the denominator, then in case of / operator, the result will always be zero. In case of % operator, the value will numerator itself.

**= 0**



**NUMERATOR / DENOMINATOR**

**<**

###### NUMERATOR % DENOMINATOR = NUMERATOR

**<**

## Unary Operator:

* Unary operators need only one operand. They are used to increment, decrement or negate a value.

###### Unary minus [-]

 used for negating the values.

###### Unary plus [+]

 used for giving positive values. Only used when deliberately converting a negative value to positive.

###### Increment operator [++]

 used for incrementing the value by 1. There are two varieties of increment operator. **Post-Increment:** Value is first used for computing the result and then incremented. **Pre-Increment:** Value is incremented first and then result is computed.

###### Decrement operator [--]

 used for decrementing the value by 1. There are two varieties of decrement operator. **Post-Decrement:** Value is first used for computing the result and then decremented. **Pre-Decrement:** Value is decremented first and then result is computed.

###### Logical not operator [!]

 used for inverting a boolean value.

### Example Program:

##### public class UnaryOperators {

##### public static void main(String[] args)

##### {

##### int a = 20, b = 10, c = 0, d = 20, e = 40;

##### boolean condition = true;

##### // pre-increment operator

##### // a = a+1 and then c = a; c = ++a;

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

##### // post increment operator

##### // c=b then b=b+1 c = b++;

##### System.out.println("Value of c (b++) = " + c);

##### // pre-decrement operator

##### // d=d-1 then c=d c = --d;

##### System.out.println("Value of c (--d) = " + c);

##### // post-decrement operator

##### // c=e then e=e-1 c = e--;

##### System.out.println("Value of c (e--) = " + c);

##### // Logical not operator

##### System.out.println("Value of !condition =" + !condition);

##### }

##### }

### Output:

##### Value of c (++a) = 21 Value of c (b++) = 10 Value of c (--d) = 19 Value of c (e--) = 40 Value of !condition =false

## Assignment Operator:

Assignment operator [=] is used to assign a value to any variable. It has a right to left associativity, i.e. value given on right hand side of operator is assigned to the variable on the left and therefore right-hand side value must be declared before using it or should be a constant.

###### General format:

###### variable = value;

Assignment operator can be combined with other operators to build a shorter version of statement called Compound Statement. For example, instead of a = a+5, we can write a += 5.

+=, for adding left operand with right operand and then assigning it to variable on the left.

-=, for subtracting left operand with right operand and then assigning it to variable on the left.

\*=, for multiplying left operand with right operand and then assigning it to variable on the left.

/=, for dividing left operand with right operand and then assigning it to variable on the left.

%=, for assigning modulo of left operand with right operand and then assigning it to variable on the left.

### Example Program:

##### public class AssignmentOperators { public static void main(String[] args)

##### {

##### int a = 20, b = 10, c, d, e = 10, f = 4, g = 9;

##### // simple assignment operator c = b;

##### System.out.println("Value of c = " + c);

##### // This following statement would throw an exception

##### // as value of right operand must be initialised

##### // before assignment, and the program would not

##### // compile.

##### // c = d;

##### // instead of below statements, shorthand

##### // assignment operators can be used to

##### // provide same functionality.

##### a = a + 1; b = b - 1; e = e \* 2; f = f / 2;

##### System.out.println("a, b, e, f = " + a + ", " + b + ", " + e + ", " + f); a = a - 1;

##### b = b + 1; e = e / 2; f = f \* 2;

##### // shorthand assignment operator a += 1;

##### b -= 1;

##### e \*= 2;

##### f /= 2;

##### System.out.println("a, b, e, f (" + "using shorthand operators) = " + a + ", " + b + ", "

##### + e + ", " + f);

##### }

##### }

### Output:

##### Value of c = 10

##### a, b, e, f = 21, 9, 20, 2

##### a, b, e, f (using shorthand operators) = 21, 9, 20, 2

## Relational Operators:

These operators are used to check for relations like equality, greater than, less than. They return boolean result after the comparison and are extensively used in looping statements as well as conditional if else statements.

###### General format:

variable **relation\_operator** value

Some of the relational operators are-

 ==, Equal to: returns true of left-hand side is equal to right hand side.

* !=, Not Equal to : returns true of left hand side is not equal to right hand side.
* <, less than: returns true of left-hand side is less than right hand side.
* <=, less than or equal to: returns true of left-hand side is less than or equal to right hand side.
* >, Greater than: returns true of left-hand side is greater than right hand side.
* >=, Greater than or equal to: returns true of left-hand side is greater than or equal to right hand side.

### Example Program:

##### public class RelationalOperators {

##### public static void main(String[] args)

##### {

##### int a = 20, b = 10; boolean condition = true;

##### // various conditional operators 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("a >= b :" + (a >= b)); System.out.println("a != b :" + (a != b));

##### System.out.println("condition==true :"+ (condition == true));

##### }

##### }

### Output:

##### a == b :false a < b :false a <= b :false a > b :true

##### a >= b :true a != b :true

##### condition==true :true

## Logical Operators

These operators are used to perform “logical AND” and “logical OR” operation, i.e. the function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e. it has a short-circuiting effect. Used extensively to test for several conditions for deciding.

Conditional operators:

* &&, Logical AND : returns true when both conditions are true.
* ||, Logical OR : returns true if at least one condition is true.

### Example Program:

##### public class LogicalOperators {

##### public static void main(String[] args)

##### {

##### String x = "Admin"; String y = "password";

##### // Check if user-name and password match or not. if ((x.equals("Admin") && y.equals("password"))

##### || (y.equals("password") && x.equals("Admin"))) { System.out.println("Welcome user.");

##### }

##### else {

##### }

##### }

##### }

##### System.out.println("Wrong uid or password");

### Output:

##### Welcome user.

## Ternary Operator:

Ternary operator is a shorthand version of if-else statement. It has three operands and hence the name ternary. It is also called as conditional operator. It is actually the if condition that we use in decision making, but using conditional operator, we turn the if condition statement into a short and simple operator.

###### General format:

###### condition ? expression 1: expression 2

The question mark "?" in the syntax represents the if part.

The first expression (condition) generally returns either true or false, based on which it is decided whether (expression 1) will be executed or (expression 2)

If (condition) returns true then the expression on the left side of " : " i.e (expression 1) is executed. If (condition) returns false then the expression on the right side of " : " i.e (expression 2) is executed.

### Example Program:

##### public class TernaryOperators {

##### public static void main(String[] args)

##### {

##### int a = 20, b = 10, c = 30, result;

##### // result holds max of three numbers

##### result = ((a > b) ? (a > c) ? a : c : (b > c) ? b : c); System.out.println("Max of three numbers = " + result);

##### }

##### }

### Output:

##### Max of three numbers = 30

## Bitwise Operator

These operators are used to perform manipulation of individual bits of a number. They can be used with any of the integer types. Generally, they are used when performing update and query operations of Binary indexed tree.

* **&, Bitwise AND operator:** returns bit by bit AND of input values.
* **|, Bitwise OR operator:** returns bit by bit OR of input values.
* **^, Bitwise XOR operator:** returns bit by bit XOR of input values.
* **~, Bitwise Complement Operator (Tilde Operator):** This is a unary operator which returns the one’s

compliment representation of the input value, i.e. with all bits inversed.

### Example Program:

##### public class BitwiseOperators {

##### public static void main(String[] args) { int a = 10; int b = 7;

##### 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 = " + ~a);

##### a &= b;

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

##### }

##### }

### Output:

##### a&b = 2

##### a|b = 15

##### a^b = 13

##### ~a = -11

##### a = 2

## Shift Operator

These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively. They can be used when we have to multiply or divide a number by two.

###### General format:

Number/integer variable **Shift Operator** number\_of\_places\_to\_shift

**<<, Left shift operator:** shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two.

**>>, Signed Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit depends on the sign of initial number. Similar effect as of dividing the number with some power of two.

### Example Program:

##### public class ShiftOperators

##### {

##### public static void main(String[] args) { int a = 10;

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

##### }

##### }

### Output:

##### a<<2 = 40

##### a>>2 = 2

## instanceof Operator

Instance of operator is used for type checking. It can be used to test if an object is an instance of a class, a subclass or an interface.

###### General format:

object **instanceof** class/subclass/interface

### Example Program:

##### class InstanceOfOperators {

##### public static void main(String[] args)

##### {

##### Person obj1 = new Person(); Person obj2 = new Boy();

##### // As obj is of type person, it is not an instance of Boy or interface

##### System.out.println("obj1 instanceof Person: "+ (obj1 instanceof Person)); System.out.println("obj1 instanceof Boy: "+ (obj1 instanceof Boy)); System.out.println("obj1 instanceof MyInterface: "+ (obj1 instanceof MyInterface));

##### // Since obj2 is of type boy, whose parent class is person

##### // and it implements the interface Myinterface it is instance of all of these classes

##### System.out.println("obj2 instanceof Person: "+ (obj2 instanceof Person)); System.out.println("obj2 instanceof Boy: "+ (obj2 instanceof Boy)); System.out.println("obj2 instanceof MyInterface: "+ (obj2 instanceof MyInterface));

##### }

##### }

##### class Person { }

##### class Boy extends Person implements MyInterface { } interface MyInterface { }

### Output:

##### obj1 instanceof Person: true obj1 instanceof Boy: false

##### obj1 instanceof MyInterface: false obj2 instanceof Person: true

##### obj2 instanceof Boy: true

##### obj2 instanceof MyInterface: true

Operator Precedence & Associativity

In an expression that contains multiple operators, Java uses a number of rules to decide the order in which the operators are evaluated. The first and most important rule is called **operator precedence**. Operators in an expression that have higher precedence are executed before operators with lower precedence.

If consecutive operators in an expression have the same precedence, a rule called **associativity** is used to decide the order in which those operators are evaluated. An operator can be left-associative, right-associative, or non-associative.

**Left-associative** operators of the same precedence are evaluated in order from left to right.

**Right-associative** operators of the same precedence are evaluated in order from right to left.

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

|  |  |  |
| --- | --- | --- |
| **Precedence** | **Operator** | **Associativity** |
| **1** | (), [] | non-associative |
| **2** | new | non-associative |
| **3** | . | left-associative |
| **4** | ++, - - | non-associative |
| **5** | - (unary), + (unary), !, ~, ++, - -, (*type*) | right-associative |
| **6** | \*, /, % | left-associative |
| **7** | +, - | left-associative |
| **8** | <<, >>, >>> | left-associative |
| **9** | <, >, <=, >=, instanceof | non-associative |
| **10** | ==, != | left-associative |
| **11** | & | left-associative |
| **12** | ^ | left-associative |
| **13** | | | left-associative |
| **14** | && | left-associative |
| **15** | || | left-associative |
| **16** | ?: | right-associative |
| **17** | =, \*=, /=, %=, -=, <<=, >>=, >>>=, &=, ^=, |= | right-associative |

Control Statements:

A control statement works as a determiner for deciding the next task of the other statements whether to execute or not. An ‘If’ statement decides whether to execute a statement or which statement has to execute first between the two. In Java, the control statements are divided into three categories which are Decision Making Statements, Looping Statements, and Branching statements. A program can execute from top to bottom but if we use a control statement, we can set order for executing a program based on values and logic.

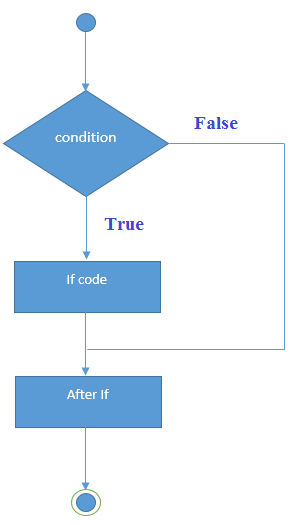
Decision Making in Java

Decision making statements are statements which decides what to execute and when. They are similar to decision making in real time. Control flow statements control the flow of a program’s execution. Here flow of execution will be based on state of a program.

#### Simple if statement

Simple if statement is the basic of decision-making statements in Java. It decides if certain amount of code should be executed based on the condition. It executes the if block if condition is true.

###### Syntax:



if (condition) {

Statemen 1; //if condition becomes true then this will be executed

}

Statement 2; //this will be executed irrespective of condition becomes true or false

### Example Program:

##### public class IfExample

##### {

##### public static void main(String[] args)

##### {

##### int age=20; if(age>18)

##### {

##### System.out.println("Age is greater than 18");

##### }

##### System.out.println(“Now you have learnt simple if”);

##### }

##### }

### Output:

##### Age is greater than 18

##### Now you have learnt simple if

#### if-else statement

In if…else statement, if condition is true then statements in if block will be executed but if it comes out as false then else block will be executed.

###### Syntax:

if (condition) {

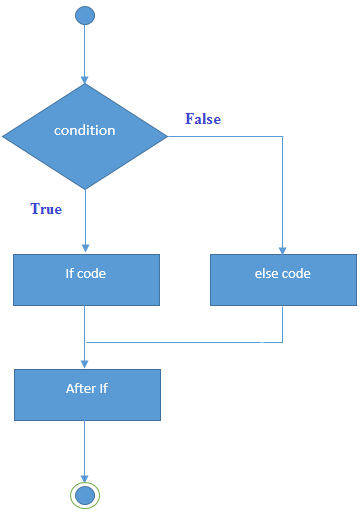
Statement 1; //if condition becomes true then this will be executed

}

else {

Statement 2;// if condition becomes false then this will be executed

}



### Example Program:

##### public class IfExample

##### {

##### public static void main(String[] args)

##### {

##### int age=10; if(age>18) {

##### System.out.println("Age is greater than 18");

##### }

##### else {

##### }

##### }

##### System.out.println(“Age is lesser than 18”);

### Output:

##### Age is lesser than 18

#### if-else-if ladder

if…else if statements will be used when we need to compare the value with more than 2 conditions. They are executed from top to bottom approach. As soon as the code finds the matching condition, that block will be executed. But if no condition is matching then the last else statement will be executed.

###### Syntax:

if (condition2) {

Statemen 1; //if condition1 becomes true then this will be executed

}

else if (condition2) {

Statement 2; // if condition2 becomes true then this will be executed

}

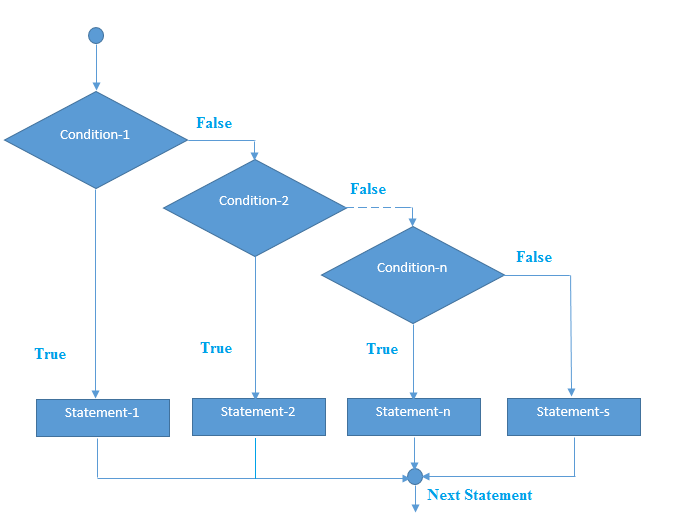
.

.

else {

Statement 3; //executed when no matching condition found

}



### Example Program:

##### public class IfElseIf{

##### public static void main(String[] args) { int marks=65;

##### if(marks<50){

##### System.out.println("fail");

##### } else if(marks>=50 && marks<60){ System.out.println("D grade");

##### } else if(marks>=60 && marks<70){ System.out.println("C grade");

##### } else if(marks>=70 && marks<80){ System.out.println("B grade");

##### } else if(marks>=80 && marks<90){ System.out.println("A grade");

##### } else if(marks>=90 && marks<100){ System.out.println("A+ grade");

##### } else {

##### System.out.println("Invalid!");

##### }

##### }

##### }

### Output:

##### C grade

#### nested if statement

Nested if statement is if inside an if block. It is same as normal if…else statement but they are written inside another if…else statement. Here, the inner if block condition executes only when outer if block condition is true.

###### Syntax:

if (condition1) {

if (condition2) {

Statement 1; //executed when condition1 & condition2 are true

}

else {

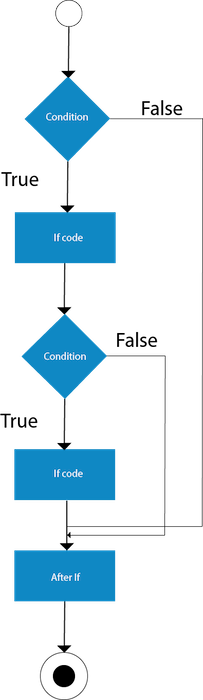
Statement 2; //executed when condition2 is false

}

}

else { Statement 3; //executed when condition1 is false

}



### Example Program:

##### public class NestedIfExample {

##### public static void main(String[] args) { int age=20;

##### int weight=80; if(age>=18){

##### if(weight>50){

##### System.out.println("You are eligible to donate blood");

##### }

##### else {

##### System.out.println(“Not eligible due to underweight”);

##### }

##### }

##### else

##### {

##### }

##### }

##### }

##### System.out.println(“Not eligible due to underage”);

### Output:

##### You are eligible to donate blood

#### Switch Statement

Java switch statement compares the value and executes one of the case blocks based on the condition. It is same as if…else if ladder. In other words, the switch statement tests the equality of a variable against multiple values. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long.

Below are some points to consider while working with switch statements:

* case value must be of the same type as expression used in switch statement.
* Case value must be a constant or literal. It doesn’t allow variables.
* case values should be unique. If it is duplicate, then program will give compile time error.
* There can be one or N number of case values for a switch expression.
* The switch expression must be of byte, short, int, long (with its Wrapper type), enums and String
* Each case statement can have a break statement which is optional.
* When control reaches to the break statement, it jumps the control after the switch expression.
* If a break statement is not found, it executes the next case.
* The case value can have a default label which is optional.

###### Syntax:

switch(expression)

{

case value1:

//code to be executed; break; //optional

case value2:

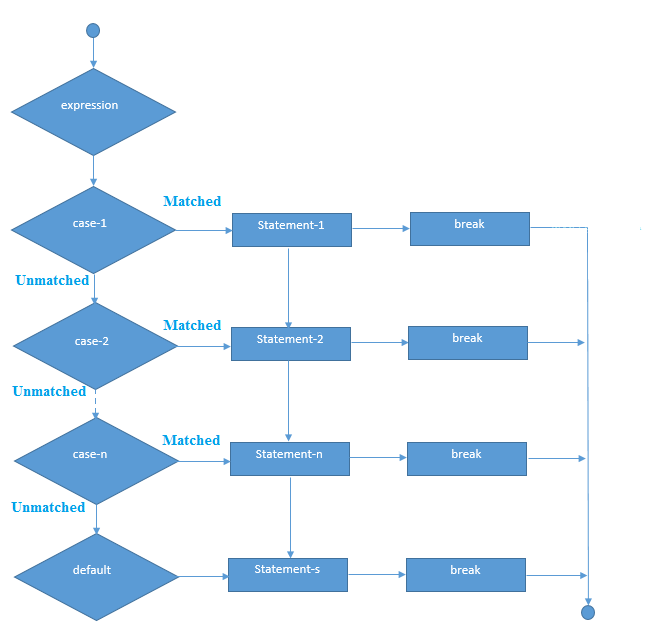
//code to be executed; break; //optional

......

default:

code to be executed if all cases are not matched;

}



### Example Program:

##### public class SwitchExample {

##### public static void main(String[] args)

##### {

##### int number=20; switch(number)

##### {

##### case 10: System.out.println("10"); break;

##### case 20: System.out.println("20"); break;

##### case 30: System.out.println("30"); break;

##### default:

##### System.out.println("Not in 10, 20 or 30");

##### }

##### }

##### }

### Output:

##### 20

Looping Statements

Looping statements are the statements which executes a block of code repeatedly until some condition meet to the criteria. Loops can be considered as repeating if statements. There are 3 types of loops available in Java.

#### for loop

The Java *for loop* is used to iterate a part of the program several times. If the number of iterations is fixed, it is recommended to use for loop. It is the most common and widely used loop in Java. It is the easiest way to construct a loop structure in code as initialization of a variable, a condition and increment/decrement are declared only in a single line of code. It is easy to debug structure in Java.

There are three types of for loops in java.

*Simple for loop*

In a simple for loop we can initialize the variable, check condition and increment/decrement value. It consists of four parts:

**Initialization:** It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.

**Condition:** It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.

**Statement:** The statement of the loop is executed each time until the second condition is false.

**Increment/Decrement:** It increments or decrements the variable value. It is an optional condition.

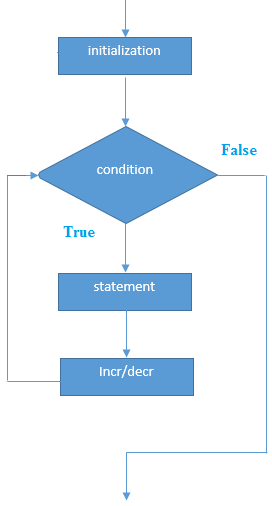
###### Syntax:

for (initialization; condition; increment/decrement)

{

statement;

}



### Example Program:

##### class forLoop

##### {

##### public static void main(String args[])

##### {

##### for (int j = 1; j <= 5; j++) System.out.println(j);

##### }

##### }

### Output:

##### 1

##### 2

##### 3

##### 4

##### 5

*Nested for loop*

If we have a for loop inside another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

### Example Program:

##### public class NestedFor {

##### public static void main(String[] args) { for(int i=1;i<=3;i++){

##### for(int j=1;j<=3;j++){ System.out.println(i+" "+j);

##### }

##### }

##### }

##### }

### Output:

##### 1 1

##### 1 2

##### 1 3

##### 2 1

##### 2 2

##### 2 3

##### 3 1

##### 3 2

##### 3 3

*for each loop*

The for-each loop is used to traverse array or collection in java. It is easier to use because we don’t have to increment the value. It returns the elements from the array or collection one by one. It is easier to use than simple for loop because we don't need to increment value and use subscript notation. Traversing is possible only in forward direction and not in reverse direction.

It works on elements basis not index. It returns element one by one in the defined variable.

###### Syntax:

for(Type var:array/collections){ Statements;

}

### Example Program:

##### public class ForEach {

##### public static void main(String[] args) {

##### int arr[]={12,23,44,56,78};

##### for(int i:arr){ System.out.println(i);

##### }

##### }

##### }

### Output:

##### 12

##### 23

##### 44

##### 56

##### 78

*Labelled for loop*

We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful if we have nested for loop so that we can break/continue specific for loop.

Usually, break and continue keywords breaks/continues the innermost for loop only.

###### Syntax:

labelname: for(initialization;condition;increment/decrement){ Statements;

}

### Example Program:

##### public class LabeledFor{

##### public static void main(String[] args) { aa:

##### for(int i=1;i<=3;i++){ bb:

##### for(int j=1;j<=3;j++){ if(i==2&&j==2){

##### break aa;

##### }

##### System.out.println(i+" "+j);

##### }

##### }

##### }

##### }

### Output:

##### 1 1

##### 1 2

##### 1 3

##### 2 1

*Infinite for loop*

If you use two semicolons ;; in the for loop, it will be infinite for loop. Ctrl + c has to be pressed in order to break the infinite loop.

###### Syntax:

for(;;){ Statements;

}

### Example Program:

##### public class InfiniteFor {

##### public static void main(String[] args) { for(;;){

##### System.out.println("infinite loop");

##### }

##### }

##### }

### Output:

##### Infinite loop will be printed infinite number of times and to exit from this, Ctrl + c has to be pressed in the console / command prompt.

#### while loop

While loops are simplest kind of loop. It checks and evaluates the condition and if it is true then executes the body of loop. This is repeated until the condition becomes false. Condition in while loop must be given as a Boolean expression. If int or string is used instead, compile will give the error. If the number of iterations are not fixed, it is recommended to use while loop.

###### Syntax:

while (condition)

{

statement1;

}

### Example Program:

##### public class WhileExample {

##### public static void main(String[] args) { int i=1;

##### while(i<=7){ System.out.println(i);

##### i++;

##### }

##### }

##### }

### Output:

##### 1

##### 2

##### 3

##### 4

##### 5

##### 6

##### 7

*Infinite while loop*

If you pass true in the while loop, it will be infinitive while loop. Ctrl + c has to be pressed in order to break the infinite loop.

###### Syntax:

while (true)

{

statements;

}

### Example Program:

##### public class InfiniteWhile {

##### public static void main(String[] args) { while(true){

##### System.out.println("infinite while loop");

##### }

##### }

##### }

### Output:

##### Infinite while loop will be printed infinite number of times and to exit from this, Ctrl + c has to be pressed in the console / command prompt.

#### do-while loop

do…while works same as while loop. It has only one difference that is in do…while, condition is checked after the execution of the loop body. That is why this loop is considered as exit control loop. In do…while loop, body of loop will be executed at least once before checking the condition. If the number of iterations is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

###### Syntax:

do{ Statements;

}while(condition);

### Example Program:

##### public class dowhileLoop

##### {

##### public static void main(String args[])

##### {

##### int j = 12; do

##### {

##### System.out.println(j); j = j+1;

##### } while (j <= 10)

##### }

##### }

### Output:

##### 12

*Infinite do while loop*

If you pass true in the while loop, it will be infinitive while loop. Ctrl + c has to be pressed in order to break the infinite loop.

###### Syntax:

do{ Statements;

}

While(true);

### Example Program:

##### public class Infinitedowhile {

##### public static void main(String[] args) { do{

##### System.out.println("infinite do while loop");

##### } while(true);

##### }

##### }

### Output:

##### Infinite do while loop will be printed infinite number of times and to exit from this, Ctrl + c has to be pressed in the console / command prompt.

Branching Statement

Branching statements jump from one statement to another and transfer the execution flow. There are 3 branching statements in Java.

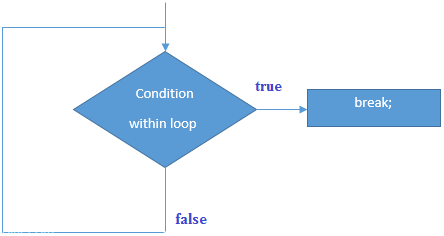
#### break statement

break statement is used to terminate the execution and bypass the remaining code in loop. It is mostly used in loop to stop the execution and comes out of loop. When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop. It is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

We can use Java break statement in all types of loops such as for loop, while loop and do-while loop.

###### Syntax:

jump-statement; break;



### Example Program:

##### public class BreakExample {

##### public static void main(String[] args) {

##### //using for loop for(int i=1;i<=10;i++){

##### if(i==5){

##### //breaking the loop break;

##### }

##### System.out.println(i);

##### }

##### }

##### }

### Output:

##### 1

##### 2

##### 3

##### 4

*break Statement with Inner Loop*

It breaks inner loop only if you use break statement inside the inner loop.

### Example Program:

##### public class BreakInnerLoop {

##### public static void main(String[] args) {

##### //outer loop

##### for(int i=1;i<=3;i++){

##### //inner loop

##### for(int j=1;j<=3;j++){ if(i==2&&j==2){

##### //using break statement inside the inner loop break;

##### }

##### System.out.println(i+" "+j);

##### }

##### }

##### }

##### }

### Output:

##### 1 1

##### 1 2

##### 1 3

##### 2 1

##### 3 1

##### 3 2

##### 3 3

*break Statement with labelled for loop*

We can use break statement with a label. This feature is introduced since JDK 1.5. So, we can break any loop in Java now whether it is outer loop or inner.

### Example Program:

##### public class BreakLabelledFor {

##### public static void main(String[] args) { aa:

##### for(int i=1;i<=3;i++){ bb:

##### for(int j=1;j<=3;j++){ if(i==2&&j==2){

##### //using break statement with label break aa;

##### }

##### System.out.println(i+" "+j);

##### }

##### }

##### }

##### }

### Output:

##### 1 1

##### 1 2

##### 1 3

##### 2 1

*break Statement with while loop*

### Example Program:

##### public class BreakWhile {

##### public static void main(String[] args) {

##### //while loop int i=1; while(i<=10){

##### if(i==5){

##### //using break statement i++;

##### break;//it will break the loop

##### }

##### System.out.println(i); i++;

##### }

##### }

##### }

### Output:

##### 1

##### 2

##### 3

##### 4