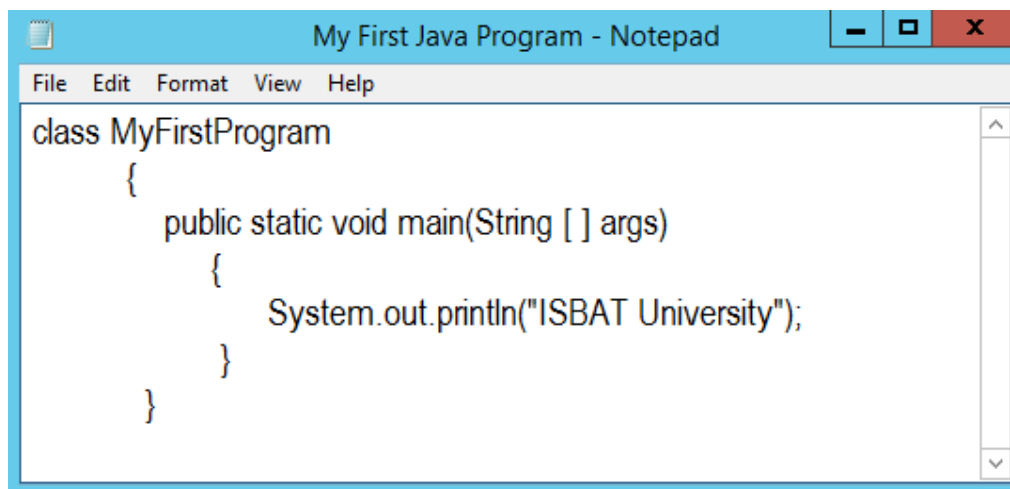


WHAT IS JAVA?

Java is an **object-oriented programming language** and a **platform**. It was made by the company named **Sun Microsystems** which was later acquired by **Oracle Corporation**. Today **oracle** manages and releases different versions of **java**. **Java** was made by a team led by **James Gosling** at **Sun Microsystems**. As a **programming language**, we can write programs in **java** using **English** based commands which can be executed on different **types of devices**. Some basic examples of programs are "**Addition of two number**", "**Multiplication of two number**" "**Printing a string**" etc.

WHAT IS PROGRAM? A program is a group of instructions written with the help of a programming language syntax. It performs a **specific task** when executed on any device E.g., "adding two number", "multiplication of two number" etc.

WHAT IS SYNTAX IN JAVA? Like grammar in English, **Java** also has a set of defined rules that must be followed while writing the programs. These rules are known as **Syntax** in java. A programmer should know these syntaxes before writing. The image below shows a very basic java program written inside a file.



```
class MyFirstProgram
{
    public static void main(String [ ] args)
    {
        System.out.println("ISBAT University");
    }
}
```

In above image, the program has **English based words** also known as **commands**). These words have special meaning in **java** programming language. It shows some of the devices where java programs are executed to perform specific tasks.

IS JAVA A HIGH-LEVEL LANGUAGE? Yes, java is a **high-level language**. You can ably write computer instructions using **English based commands** instead of writing instructions in **low level language** (**machine code** or **assembly language**). At the time of **execution**, these commands or **high-level instructions** are translated into **low level instructions** that computer can understand and execute.

JAVA PLATFORM: Java is also a **Platform**. When you run a **java program** or **application**, java creates a **runtime environment** where your java programs or application runs. Generally operating systems like **Microsoft Windows**, **Linux**, **Solaris**, **Mac OS** etc. are known as **platforms**. These platforms give you an environment where you can run different types of **softwares** or **applications**, similarly java creates an environment **at runtime** where you only can run **java programs** or **applications**. Java

platform is a **software-only** platform that runs on top of a **hardware-based platforms (OS)** and they, Java platforms, have a **Java Virtual Machine (JVM)** and **application programming interface (API)**.

HOW DOES A COMPUTER UNDERSTAND A JAVA PROGRAM?

Computer doesn't understand java program directly as it's written in English based commands. Computer understands only **low-level instructions** (**machine code** or **assembly language**). Java program is converted into low level instructions using java software (**JDK** and **JRE**) which is then executed by a computer.



TYPES OF JAVA PLATFORMS

Java platforms are basically java softwares that is used in development and execution of java programs or applications. You can download and use these softwares. There are four types of java platforms:

1. **Java Platform, Standard Edition (Java SE).**
2. **Java Platform, Enterprise Edition (Java EE)**
3. **Java Platform, Micro Edition (Java ME).**
4. **JavaFX**

WHY DOES JAVA PROVIDES DIFFERENT TYPES OF PLATFORMS?

As a **developer** or **enterprise**, you may want to develop **different categories of applications** like **desktop application**, **web application**, **gaming applications** etc. To do so java provides different types of platforms.

1. Java SE: This is the most **basic** or **core** platform. It defines basics of java language like **data types**, **syntaxes**, **classes**, **objects**, **interfaces** etc. It also includes some **high-level classes** that are used for security, networking, database access etc. All **stand-alone** or **desktop applications** are developed using this platform. **AWT** and **Swing** used for developing desktop applications are also part of a **Java SE**. You can also consider **Java SE's API** as Java programming language. Apart from core **API's**, it also consists of a **java virtual machine (JVM)**, **development tools**, and other **class libraries** which are commonly used in Java applications.

2. Java EE: platform is built on top of **Java SE platform**. It also entails of an **API** and runtime environment that is used for developing and running **large-scale**, **multi-tiered**, **reliable**, **scalable** and **secure network applications**. All the java web and enterprise applications are developed using this platform. Technologies like servlet, jsp, struts, JPA, EJB etc are part of this platform.

3. Java ME: This platform also consists of an **API** and a small virtual machine for running Java programs and applications on **small devices like mobile phones**. This **API** is a subset of the Java SE API, along with some additional **class libraries** that is useful for small device application development. All java mobile applications and games are developed using this platform.

4. JavaFX: has a **set of graphics** and **media packages** to design **rich client applications** that can run consistently on different platforms. The basic intent was to replace **Swing** because it was bit complex. **JavaFX** offers a much simpler way to create desktop applications as well as rich internet applications because it provides the facility to develop user interfaces as well. JavaFX Scene Builder provides drag and drop UI component feature to develop applications.

USAGE OF JAVA LANGUAGE

With time the use of java has increased significantly which makes it one of the most ideal languages for programmers and enterprises. According to **Oracle**, more than **3 billion** devices run java. Some of the

devices that uses java include; **ATMs, Smartcards, Routers, Switches, Smart Meters, Access Control System, Medical Devices** etc. There are several applications and softwares that use java in one way or the other. Some of the uses of java are:

- Android operating system is developed using java. Most games and softwares running on android are developed using java technologies.
- Popular desktop applications like acrobat reader, antiviruses softwares, media players are developed using java language.
- Popular web applications like Linkedin.com, twitter.com are also using java in one way or other.
- Due to enhanced security of java, most of the banking applications are developed using java.
- Big Data technologies like Hadoop, HBase, ElasticSearch etc are also using java.
- Programmer's text editor like eclipse, Netbeans, JEdit etc are also developed using java.

HISTORY OF THE JAVA LANGUAGE

Java was made by a company called **Sun Microsystems**. **James Gosling** is known as the maker of **java** language. In **1991**, a team named **Green team**, led by **James Gosling** started working on java language project. Their intention was to develop the language that can be used for digital devices like **cell phones, televisions, set-top boxes** etc. **Sun Microsystems** released the first version as Java 1.0 in **1995**, and its main focus had shifted to use on the Internet. Firstly, the name of language was **Oak**, a tree name which was outside of **Gosling office**. Later it was renamed as **Green** and was finally renamed as **Java**. Gosling designed the syntaxes of Java similar to C/C++, so that programmers could find it easy to use as these languages were already being used by most of the programmers at that time. On **Nov 13, 2006**, **Sun Microsystems** released most of its **Java virtual machine (JVM)** as free and open-source software under the terms of a GNU **General Public License (GPL)**. On May 8, 2007, Sun made all its JVM's core code available under free software/open-source distribution terms.

WHAT IS OPEN-SOURCE SOFTWARE?

Open-source software is a software whose source code is also released along with software so that you can read or change the source code as per your needs and can re-distribute the software as well. You can download and use it without any cost. **Java** is not completely open-source software. In **2009-10 Oracle** acquired **Sun Microsystems**, since then all major versions and updates are handled by Oracle. With time Java has evolved as a very successful language to be used on and off the Internet. Today many of the applications and devices which we use on daily basis are using java.

IS JAVA SE 11 FREE OR PAID?

If you want to use **java 11** for commercial purpose then yes, you will have to pay and buy license from oracle. But for development and testing purposes, it's free, you can download and use it without any cost. The table below shows the list of other java versions and their release dates. The latest stable release of java is **Java SE 16** as of now (**March 2021**).

JAVA VERSION	RELEASE DATE
Java SE 12	March 19, 2019
Java SE 13	September 17, 2019

Java SE 14	March 17, 2020
Java SE 15	September 15, 2020
Java SE 16	March 16, 2021

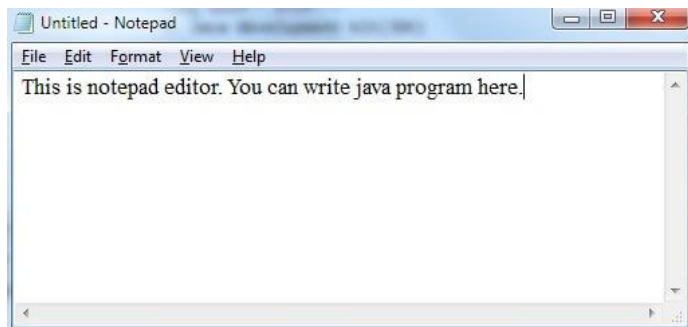
HARDWARE AND SOFTWARE REQUIREMENTS FOR JAVA

It's easy to start programming in java, you just need to have the following hardware and software needs in order to create and run java program in your system. Hardware requirements for java programming. The hardware requirement actually depends on the version of java you want to use. Java 16 version requires following hardware configuration:

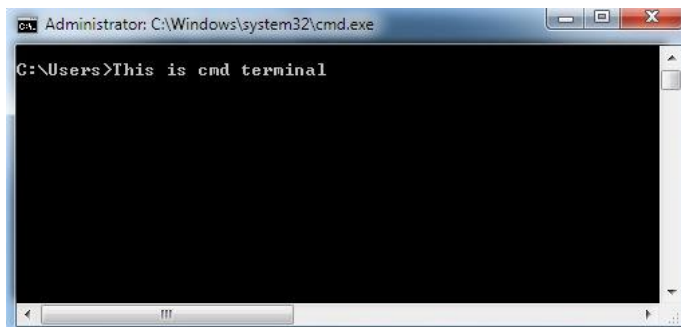
- **Processor:** 1.8 GHz or faster processor.
- **RAM:** 2 GB of RAM; 4 GB of RAM recommended (2.5 GB minimum if use on a virtual machine)
- **Disk Space:** up to 130 GB of available space, depending on features installed; typical installations require 20-50 GB of free space.

SOFTWARE REQUIREMENTS FOR JAVA PROGRAMMING

1. An editor like notepad, **notepad++**, **TextPad** etc. for windows operating system and Vim, Notepadqq, Geany etc for linux based **operating system** to write the java program. The **notepad editor** comes by default with windows operating system. The programmer can choose the editor of their choice.



An Editor



A Terminal

2. Any **terminal** like **cmd** shell etc. to compile and run java program. By default, CMD is available in **windows OS** while shell is available in Linux based OS. Programmer can use *PowerShell* (if available) and in windows to compile and run java programs. **WHAT IS TERMINAL?** A terminal is a program or software which offers a **user interface** where you can execute commands for e.g. **CMD** in **Window OS** and Secure Shell Client in Linux based OS are terminals.

3. **Java development kit (JDK)** is a software that is used to develop and execute of java programs or applications. If **JDK** is not already installed in your system then download and install it (JDK as per your OS) first. **Which JDK version should i use?** It's completely your choice. Just to add a point, from **JDK 11** and onward java is licensed, you can use it free only for development and testing purposes not for commercial purposes. If you are starting to learn java, **JDK 8** is still a good choice. That's all you need to enter in the world of java. By default, **Notepad** and **CMD** is available in windows operating system. You just have to download and install JDK if not already installed.

HOW TO INSTALL JDK IN WINDOWS

The first step is to decide which version of **jdk** you want to download. Once you have decided the version, just search on internet like "**jdk 8 download**" or "**jdk 11 download**" or "**jdk 16 download**". After that open and follow the oracle download link to download the JDK. Just to note, you will have to create an oracle account first to download it from oracle site. You can also refer to the Oracle JDK link to download the JDK. If available, you can download it from another site as well. The image link below downloads JDK 8 for windows 64-bit OS. Once you downloaded it, probably you have got an exe file. Now just double click on this file and then follow the instructions given in new pop-up window to install JDK in your system.



Java Platform(JDK) 8

NOTE: Make sure you download the right JDK for your operating system. If you are using windows OS, download a JDK for windows, similarly for other OS, download respective JDK. Also download the 32-bit JDK if your OS is 32 bits else download 64-bit JDK if your OS is 64-bit operating system. Latest downloads of JDK's includes 32- and 64-bit's packages in same exe or zip file.

PATH AND CLASSPATH IN JAVA

Once you have installed java, you may need to set a **Path** variable. A **Path** is an environment variable in an operating system. This **Path** variable stores a list of directories separated by semicolons (;) as value. These directories points to locations where executable files are available. Once a command is executed in a terminal e.g. **cmd**, the **OS** looks for the corresponding executable in these directories. In java, executable files are found in the **bin** e.g. **C:\Program Files\Java\jdk1.8.0_102\bin** folder of the java installation. Folder has executables; **javac.exe**, **java.exe**, **javap.exe**, **javadoc.exe** etc. These executables are used for different types of tasks in java e.g. **javac.exe** is used for compilation of java programs, **java.exe** used for execution of java programs. If **Path** variable is not set, you will have to go to the directory with the executables to execute a command in terminal. Once **Path** variable is set, java commands or JDK tools can be executed from any directory in a terminal. Let's see how to set **Path** variable in different operating system.

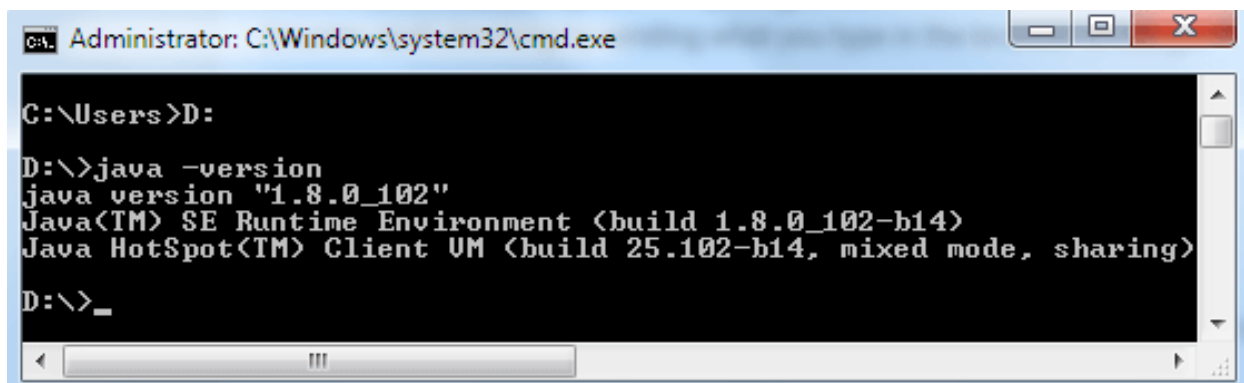
HOW TO VERIFY JAVA VERSION AND JAVA INSTALLATION

Once you have installed java in your system, check whether it has been installed properly or not. It covers both the cases, first if **Path** variable for java is not set and second if **Path** variable for java is set after installation. 1. If a **Path** variable is not set, to check if java is installed properly or not, open a command terminal like **cmd** etc. and change or move to the java installation directory the **bin** folder, now execute the command **java -version**. It will display the installed java version and other relevant data about java. Java compiler version can also be checked by executing **javac -version** command.

A screenshot of a Windows command prompt window titled "Administrator: C:\Windows\system32\cmd.exe". The prompt shows the user navigating to the Java bin directory:
C:\Users>cd C:\Program Files\Java\jdk1.8.0_102\bin
Then the user runs the command `java -version`, which outputs:
java version "1.8.0_102"
Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
Java HotSpot(TM) Client VM (build 25.102-b14, mixed mode, sharing)
The prompt returns to C:\Program Files\Java\jdk1.8.0_102\bin>

The image above shows the steps for checking a java installation in Windows operating system using **cmd** terminal. Similarly, for Linux based operating system, execute the same command in any linux based terminal to check java installation.

2. If **Path** is set, there is no need to move to java installation directory, just execute the command **java -version** in any directory. It will display the installed java version and other relevant information about java. For compiler version execute **javac -version** command.



```
Administrator: C:\Windows\system32\cmd.exe

C:\Users>D:

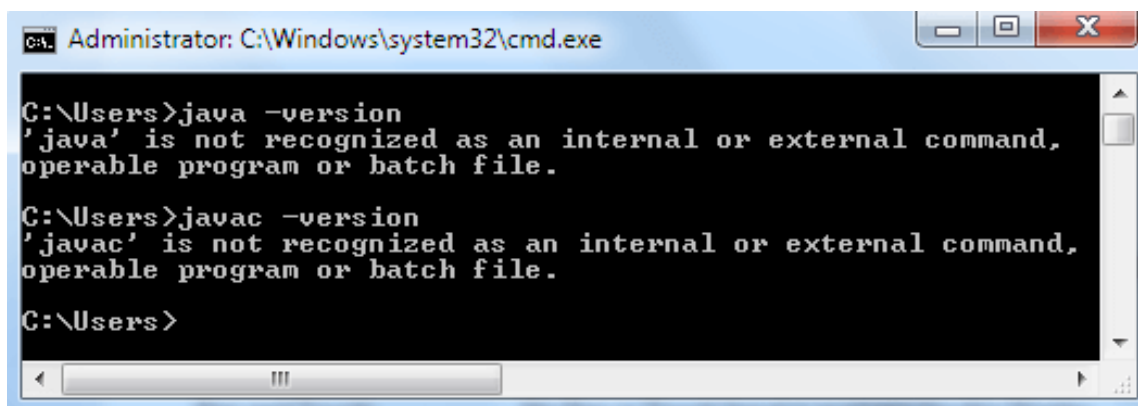
D:\>java -version
java version "1.8.0_102"
Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
Java HotSpot(TM) Client VM (build 25.102-b14, mixed mode, sharing)

D:\>_
```

If any of the above methods returns the version correctly, which means java is installed properly. Now you can start your **first program** in java.

HOW DO I KNOW IF JAVA IS INSTALLED ON WINDOWS 10?

The steps are same irrespective of which version of windows you are using. Just follow above steps to check if java is installed on windows 10 or not. If java is not installed, it will display an error message in Windows like *"java is not recognized as internal or external command, operable program or batch file"* after executing the command.



```
Administrator: C:\Windows\system32\cmd.exe

C:\Users>java -version
'java' is not recognized as an internal or external command,
operable program or batch file.

C:\Users>javac -version
'javac' is not recognized as an internal or external command,
operable program or batch file.

C:\Users>
```

HOW DO I KNOW IF I HAVE A JAVA JDK INSTALLED?

Just execute **java -version** and **javac -version** command as given above inside a terminal to see if java **jdk** is installed or not. If these commands return version of java and java compiler, it means java **jdk** is installed on your system. **NOTE:** You must be in the **bin folder** of the **JDK** to check the java installation if **Path** variable is not set. As a good practice you should always set your path variable first before you start writing programs in **java**. You can execute **java -help** command on terminal to get different usage of java command. Once you set the path of java, you need to restart the **cmd** (if already open) to get the correct result of **java -version** or **javac -version** command.

WHAT IS A SYNTAX IN THE JAVA LANGUAGE

It is a serious requirement for a programmer to first know the **syntax** of writing a program in a specific programming language. Sometimes it gets confusing for **beginners** to know what exactly a **syntax** is. Every programming language has a set of rules for writing programs in that programming language. The rules are called **syntax**. Java also has a set of rules for defining different types of **members**(**class**, **variable**, **methods** etc) in a program, we call these rules as **syntax** in java. A java programmer should know these syntaxes before writing program in java. Our real-life languages also have defined sets of rules in order to write sentences in them e.g. **English** language has also defined a set of rules, which we call as **Grammar**. You should know the grammar first in order to write sentences in english, the same applies with programming languages as well. In java, a program commonly contains **a class**. A **class** has **variables**, **methods**, **constructors** etc in it. A program may also have **looping statements**, **arrays**, **objects**, **if else statements** etc. Each of these members has some syntax which must be followed while writing them inside a program. Let's see the basic syntax of some members one by one.

SYNTAX OF CLASS IN JAVA

In order to create a class in java, we should follow below syntax :

```
class ClassName {  
    // Define class members like variables, method, constructors etc.  
}  
Example:  
class MyFirstJavaProgram {  
    // Define members of class.  
}  
  
// The class declaration below is wrong because the keyword,  
// must be in small letter.  
Class Test {  
    // Define members of class.  
}
```

Here **class** is a keyword, used to define a class in java. **ClassName** is the name of class, given by the programmer. After the **class name**, it's the class body given inside **{ }**. You can define **variables**, **methods**, **constructors** etc of this class inside the **{ }**. So in order to create a class in java you have to follow the above syntax.

IS CLASS KEYWORD CASE-SENSITIVE ?

All **keywords** in java are **case-sensitive** and they must be in **small letter**. So, you can't use the **class** keyword as **Class**, **CLASS**, **clAss** etc, it must be in small letters.

SYNTAX OF VARIABLE IN JAVA

In java if you have to create a variable, you have to follow below syntax:

```
DataType variable_name = value;
```

Example:

```
int age = 20;  
// Below variable declaration are incorrect  
// variable name should not start with digits.  
String 12message = "ISBAT University";  
// Variable name should come after data type  
year int = 2021;
```

Every variable in java must have a **data type**. The data type of a variable tells what type of data that variable can store e.g. **int** type of variable stores only **integer type value**, similarly **float** type of variable stores **float type of values only**. A **name of variable** is given by the programmer which must be a **valid identifier**. There are java **rules and conventions defined** by for naming variables.

SYNTAX OF A METHOD IN JAVA

In java if you want to create a **method**, you have to follow below syntax:

```
Return_Type methodName(DataType param1, DataType param2 ...) {  
    // method logic or code.  
    return some_value;  
}
```

Example:

```
int add(int num1, int num2) {  
    int sum = num1 + num2;  
    return sum;  
}
```

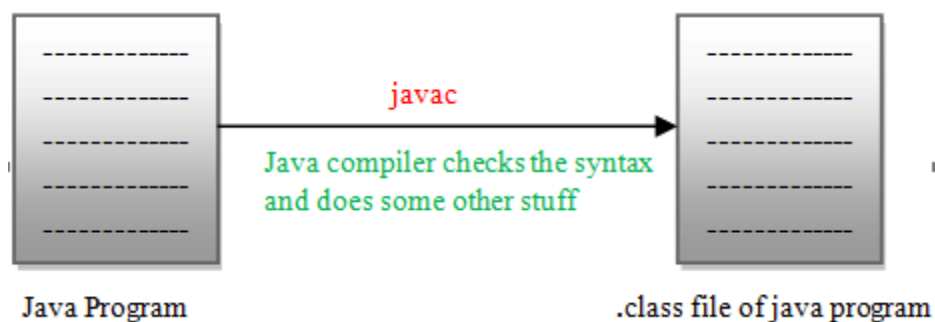
// The method declaration below is incorrect, the () must follow the method name.

```
void test {  
    // method logic or code.  
}
```

A **method** must have a **return type** which tells what type of value it returns. The **Return_Type** of a method can be **primitive** or **non-primitive data type**. If a method doesn't return a value, its return type must be **void**. After the return type, is a **method name** given by the programmer. After the **method name**, are the **parameters** given inside () which are mostly **variables**. **Parameters** are optional which means a method may or may not have parameters. You can access these parameters within the **method body only**, not **outside** the method body. **Data type** of parameters can also be **primitive** or **non-primitive**. After parameters, it's the method body given inside { }. Everything given inside { } after method name are the part of that method.

WHAT IS SYNTAX VALIDATION IN JAVA

Syntax validation is a process where a java compiler (**javac**) validates whether the members of a program are defined as per the syntax or not. If it's not as per the syntax, the java compiler will show a syntax error for that member while compiling a program. In java, once a program is written, its compiled first using java compiler before it can be executed.



If there is an **error in compilation**, a java programmer must fix that error first and then recompile the program. Only after successful compilation, will the compiler create a **.class** file of that program which is then used for execution of that program. **What is syntax error in Java?** When a member of a program is not defined **as per the syntax** of that member, the java compiler generates an error while compiling that program. We call such an **error**, a **syntax error**. **How do you find syntax errors?** By compiling the program using java compiler we get the syntax errors of a program. **How do you fix a syntax error?** Just follow the correct syntax of defining that member which is giving the syntax error. Now let us write our first program in java which will help us to understand how to write program in java.

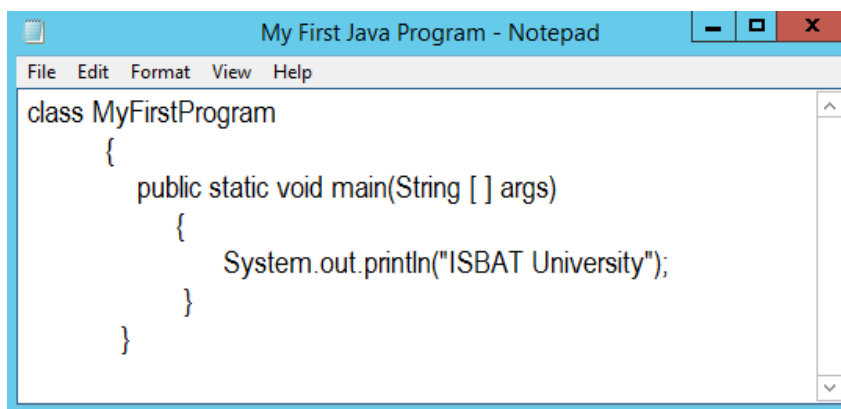
MY FIRST JAVA PROGRAM - HOW TO COMPILE AND RUN JAVA PROGRAM

It's easy to start writing program in java. To write your first java program, open an editor like notepad, notepad++ or any other editor and write bellow line of code.

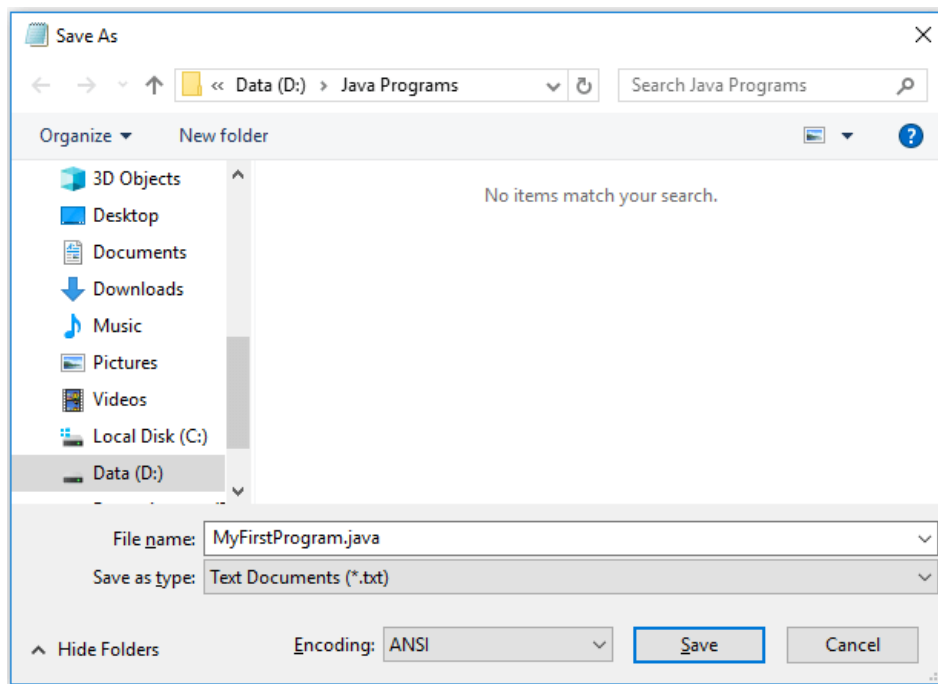
```

class MyFirstProgram
{
    public static void main(String [] args)
    {
        System.out.println("ISBAT University");
    }
}
  
```

The program given above is a basic java program which will just print a **string message** in console after execution. The image given below shows the above program written in Notepad editor.



After writing a **JAVA code** in **an editor**, save the file as per the convention given by java which says **the file name should be the same as the class name** with **.java** extension. So, let's save the above program file as **MyFirstProgram.java**. Above, the program class name is **MyFirstProgram**.

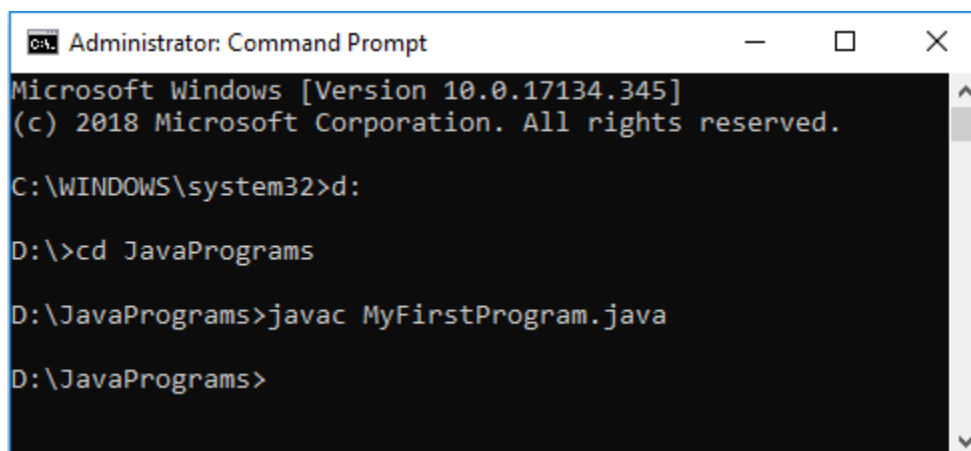


Is it a must to save a **file name** as a **class name**? **No**, if it doesn't have a **class** declared along with **public** keyword. If it has a public class then it must be saved with **public** class name. It's good to follow the convention given by java. The keyword **public** is an **access modifier**. What does a **.java file** contain? A **.java file** contains the **source code** written in **java of a program**.

HOW TO COMPILE JAVA PROGRAM

After saving a **java** file, **compile** the **java program** using the **javac** command. Before compiling, set the **Path variable** for **java** on your PC. To compile the program, open a **terminal** (e.g. **cmd** in windows) and move to the directory where above program is saved. Here our program is saved on **D:\JavaPrograms directory**. Now run this command:

javac MyFirstProgram.java



Once the programmer has executed above command, java compiler will check for any **syntax error** in the program, if it **does not** find any **error**, a **.class** file as **MyFirstProgram.class** will be generated at the same location where program is saved. In successful compilation, **compiler** does not display any message on command prompt instead a **.class** file is made. Recall you need to write the program file

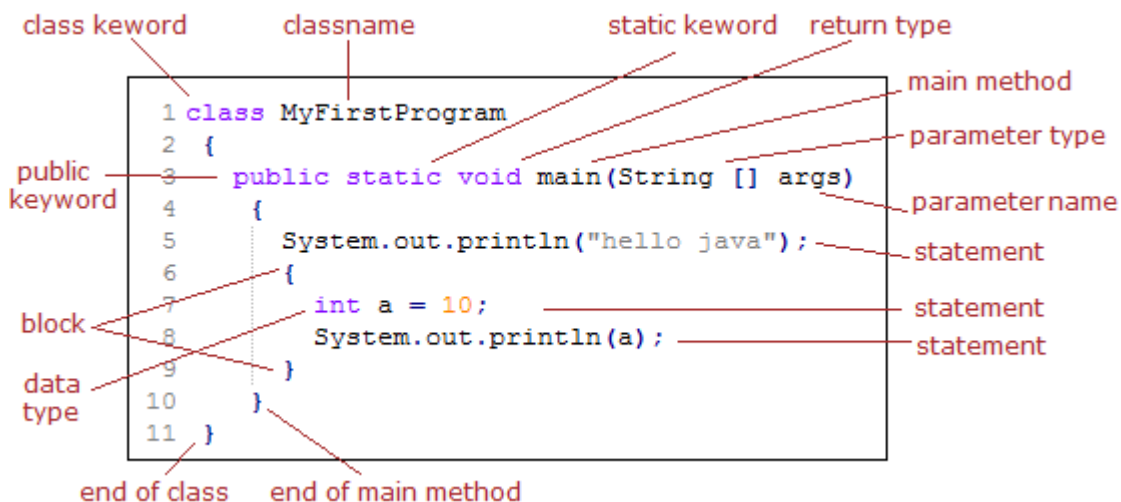
name with a **.java extension** after the **javac command** while compiling a program. **What is .class file?** A **.class** file is a compiled code of a java program into **Bytecode**. It's a middle representation of your java program. **Where does a Java program begin its execution?** From **main method**, it's the starting point of a program execution. Does **java** use the **.java** file while running a program? **No**, it only uses the **.class** file while running a program. If delete a **.java** file after its made **.class** file, can I run my program? **Yes**, you can, because the **.java** file is not needed at runtime. Only **.class** file is needed while running the program.

NOTE:

- In java, every program must have **at least one class**, but it can have more than one class.
- There should be only **one main method** having argument type as **String []** in a class.
- If there are **multiple classes** in a program, it must be saved by **class name having public access modifier if any**.
- Every starting **{** must have a balanced **}** for it.

THE BASIC OR COMMON TERMS USED IN A JAVA PROGRAMS.

A description of each term is given to beginners because it's necessary to understand these terms in order to start writing program in java.



● **Keywords** - All programming language defines a set of words which have a predefined meaning in them. These words are called **keywords**. You cannot use these words for **variable names**, **method names**, **class names** or other **identifier names** in your program, since these are **reserved words**. **Java** also has reserved words. E.g. You cannot use the **int** keyword as a **variable**, **method** or **class names**. Similarly, the words **class**, **public**, **static**, **void** are java keywords in above program.

// Following are incorrect declaration since final and switch are keyword in java.

```
int final = 20; // final cannot be used as variable name
```

```
switch { } // switch cannot be used as class name
```

Are java keywords case sensitive? **Yes**, keywords in java are **case sensitive**. All letters of keyword must be small. **What if I use keywords as my variable, class or method name?** Your program won't compile, java compiler will throw compilation error.

● **class** - The **class** is a keyword in java which is used to define a class. In java, every program must have a class. A class holds set of methods and variables. After a **class** keyword, programmers need to write the **name** of the class which is used to refer that class **within** or **outside** the class. In above program **MyFirstProgram** is the name of a class, everything that is inside the **{ }** after the class name are the part of class.

● **Statement** - A statement is like to a sentence in english language. As sentences makes a complete idea. A java statement makes a complete unit of execution. In above program line 5,7,8 are statements.

● **Method** - A method is set of statements that performs specific task or call other methods. A method has a **name** and **return type**. The name of the method is used to refer that method **within** or **outside** a class. A class can have **multiple methods**. In above program **main** is the method name. Everything that comes between the **{ }** after method name are part of the method. Every java program must have a **main** method if it needs to be run independently. The **main** method is the starting point of execution of a program in java.

● **block** - A block is a group of zero or more statements. It starts with **curly braces { }** and ends with balanced **}**. All statements inside **balanced { }** are part of a block. A **block** is used to group several statements as a single unit. A **block** can have another block inside it. Blocks do not have a name, they are just logical grouping of statements inside **{ }**. Refer **static and instance initializer blocks** in java to get more detail about blocks in java.

● **public** - The keyword **public** is an access modifier that decides the **visibility** or **accessibility** of a member. **Variables** or **methods** declared with a **public** keyword can be accessed outside the class. Since **main method** is called by the **JVM** at the time of program execution that is why it must be declared as **public**, otherwise **JVM** won't be able to find the **main method** in your program and your program will not execute. Refer **access modifiers** in java to get more data.

● **static** - The **static** is a keyword in java. A **method** or **variable** declared with **static** keyword can be called without creating an object of that class. Since a **JVM** calls the main method without creating the object of the class, that is why it must be declared as **static**, otherwise **JVM** won't be able to call the **main** method.

● **return** - The **return** is also a keyword in java. It is used to return value from the method to the caller of the method. Every **method** must have a **return type**, if it's not returning any value then the return type of that method must be **void**. Since **main** method doesn't return any value, that's why it's return type is **void**.

● **Variable & Data type** – A **Variable** in java is used to store a value. A **Data type** of a variable defines what type of data that variable can store. In above program **a** is a **variable** and **int** is its **data type**, which means **a** can only hold an integer type value.

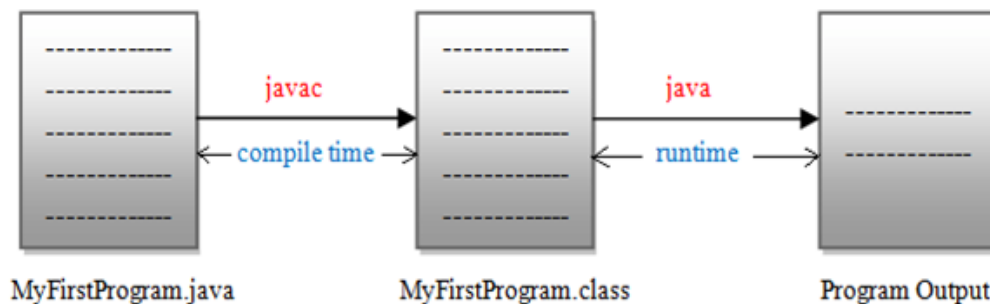
● **Parameter** - A **parameter** is special kind of **variable** which receives a **value** from the caller of a method. A **parameter** can be used in a **method** where it's declared. In the above example, **args** is

a parameter of type **String array**. Any arguments passed to a program while running the program is stored in the **args** parameter.

● **System.out.println()** – It's used to print the output **string** or **variable** of a program on **console**. Anything passed to the **println method** will be printed on the **console**. A **console** is a window or terminal where you can pass input to a program or print a programs output. **Command prompt (cmd)** in windows is an example of console.

COMPILE TIME AND RUNTIME IN JAVA

Runtime and **compile time**, these are two **programming terms** that are more often used in java programming language. Programmers especially beginners find it so difficult to know what exactly they are. In java, running a program occurs in two steps, **compilation** and **then execution**. The image below shows where does compile time and runtime takes place in execution of a program.



WHAT IS COMPILE TIME IN JAVA

After writing the program, programmer needs to **compile their programs**. As soon as the programmer starts compiling a program using **javac** compiler, the **compile time** gets started, and it ends when either a **.class** file is generated after successful compilation or an error is thrown in compilation. In other words, the process of compiling a program is referred as **compile time**. For an example if you wrote a program and saved it as **MyFirstProgram.java**, if you compile it using a **javac** command as **javac MyFirstProgram.java**, the **compile time** gets started and it ends when a **.class** file as **MyFirstProgram.class** is made or any error is thrown in compilation.

WHAT HAPPENS AT COMPILE TIME IN JAVA?

At **compile time**, a java **compiler(javac)** takes the **source code(.java file)** and checks if there is any **syntax**, **type-checking** or any **semantic errors** inside the program. If there is no **error**, the compiler generates a **.class(bytecode)** file for that **.java file**. If there is any **compilation error**, java compiler displays that **error** in command window (eg cmd).

WHAT IS COMPILE TIME ERROR IN JAVA?

If a program **element(class, method, variable, statements** etc) is not written as per it's syntax in java, the compiler **throws an error** for that element while compiling the program. We call these errors **compile time errors** as these errors are detected at compile time by the java compiler. **Does java compiler generate a .class file even if it throws compilation error? No**, it will not generate .class file, it

will only display the compilation error in console(eg. cmd) window. Let us see compile time error by an example. The most common mistake that beginners do is, they forget to add semicolon(;) at the end of a statement which results as a compilation error while compiling the program.

```
class MyFirstProgram {
    public static void main(String [] args) {
        System.out.println("first statement") // missing semicolon(;)
        System.out.println("second statement");
    }
}
```

Once you compile above program as **javac MyFirstProgram.java**, it will display a compilation error in console window like below :

Compile time Error:

```
MyFirstProgram.java:3: error: ';' expected
    System.out.println("first statement") // missing semicolon(;)
                                ^
1 error
```

WHAT IS RUNTIME IN JAVA?

As soon as the programmer **starts executing** the program using **java** command, runtime gets started and it ends when execution of program ended either successfully or unsuccessfully. In other way the process of running a program is known as runtime. For an example if you wrote a program and saved it as MyFirstProgram.java. **After compilation** when you execute the command **java MyFirstProgram** for running the program, runtime gets started and it ends when either the output of program is generated or any runtime error is thrown.

WHAT IS RUNTIME ERROR IN JAVA?

Errors which come during the **execution(runtime)** of a program are known as **runtime errors**. If a program contains **a runtime error**, it won't run successfully, rather that **runtime error** will be shown in command window(eg. cmd) at the time of execution. Imagine you wrote a program MyFirstProgram.java like below, After compilation when you run the below program using **java MyFirstProgram** command, it will throw a runtime error.

```
class MyFirstProgram {
    public static void main(String [] args) {
        int num1 = 10;
        int num2 = 0;
        System.out.println(num1/num2); // Runtime error: Divide by zero exception
    }
}
```

Runtinme Error:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at MyFirstProgram.main(MyFirstProgram.java:5)
```

DIFFERENCE BETWEEN RUNTIME AND COMPILE TIME

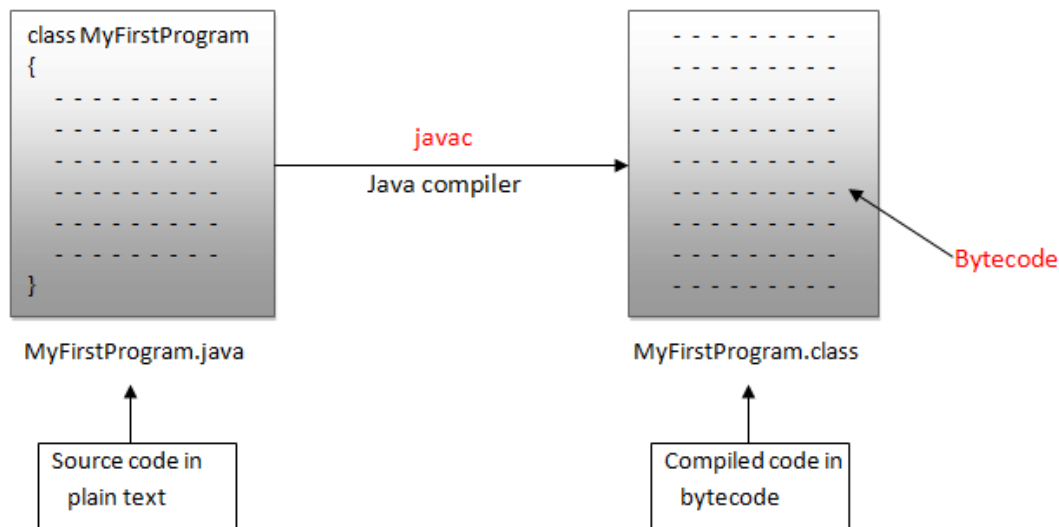
Compile time is a process where a java compiler compiles a java program and makes a **.class file**. In other way, in **compile time java source code(.java file)** is converted in to **.class** file using java compiler. While in **runtime**, the **java virtual machine JDK** loads the **.class** file in memory and executes that class to generate the output of program.

DIFFERENCE BETWEEN COMPILE TIME ERROR AND RUNTIME ERROR

Compile time errors are errors that arise when compiling a program while **runtime errors** are errors that comes at the time of execution(run-time) of a program. An example of a **compile time error** is “not adding a semicolon (;) at the end of a statement” in your java program while “dividing a number by zero” is an example of **runtime error**.

BYTECODE IN JAVA

Bytecode: In java when you compile a program, the java **compiler(javac)** converts or rewrites your program in machine language form called **bytecode**. A **.class** file that is generated after compilation are the bytecode instructions of a program. **Bytecode** and **.class** are used interchangeably, so if someone says bytecode, it simply means the **.class** file of program.



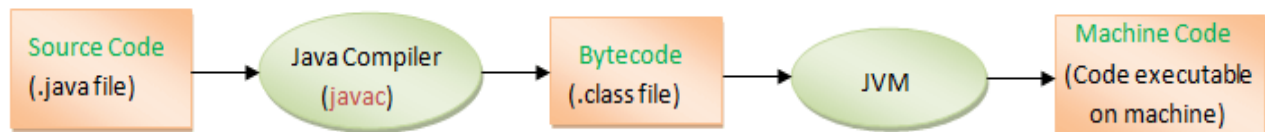
At runtime **Java virtual machine** takes the **bytecode(.class)** as an input and convert this into **machine(windows, Linux, MacOS etc)** specific code for further execution. So, **bytecode** in java is just an **intermediate representation** or **code** of a java program in form of a **.class** file. We call it intermediate representation/code because it lies between source and machine code.

Source code → Bytecode → Machine code

Bytecode is also one of java's **magic** since this along with **java virtual machine (JVM)** makes java platform **independent** and a **secure language**. It becomes platform independent because you can run the bytecode made on one PC on any other PC without the need of the original **.java** file, all you need is a **JVM** or **JRE** installed on a computer e.g. **.class** file generated on windows machine can be run on Linux machine. **Bytecode** also makes java secure because it can be run by java virtual

machine only. Also, any explicit edit in **.class** file breaks down the class meaning you will not be able to run your program through the edited **.class** file.

Does a PC understand bytecode? No, Computers don't understand bytecode directly. At runtime, the bytecode is converted into machine specific code by a java interpreter. This **machine code** is then executed by the **PC CPU** to make the output of a program. **Is bytecode human readable?** No, bytecode is not human readable. **What makes bytecode in java?** It's a java compiler(**javac**). **Is bytecode machine dependent or independent code?** Bytecode is machine independent code, it can be run on any machine, you just need the **JRE/JVM** of corresponding machine. How does java bytecode work? The image below shows the execution flow of a program in java.



In **java** language, a **source program(.java file)** is first compiled using a **java compiler(javac)**. After successful compilation, the **bytecode(.class file)** of that program is generated. When you run the program using the **java** command, the **java virtual machine(JVM)** loads the **.class** file inside memory and then gets converted(by a **java interpreter**) the instructions of **.class** file into machine executable code and finally executes that machine code in order to generate the output of program.

WHAT IS WRITE ONCE RUN ANYWHERE (WORA) IN JAVA

Write Once Run Anywhere (WORA) is a feature in java which says write your program once and then execute it anywhere (OS). In java it is attained with the help of the **bytecode**. Programmer needs to write the program once and can execute the **.class** file generated from it anywhere and any number of times. **What is the use of bytecode in Java?** The main reason to use bytecode in java is to make **java platform independent and secure**. The bytecode of a program can be run on any machine which is why java is called **platform independent**. Also, the use of **bytecode** makes java a **secure language**, any external edit in bytecode of a program fails its execution. It's the bytecode of a program which is used for running a java program. At runtime java virtual machine loads the bytecode of a program inside the memory and converts into machine code and then executes that machine code.

ADVANTAGES OF BYTECODE

- Bytecode helps java to be a **platform independence** language. This is one of the main advantages of bytecode in java.
- It helps java to be a **secure programming language**, as it can be run by java virtual machine only. Also, at runtime bytecode is validated by bytecode verifier, any explicit edit in bytecode fails its execution.
- It makes java **portable** which helps to achieve the "**Write once, run anywhere**" feature. It simply means you need to write the program once, and execute it anywhere (OS), all you need to have is the **.class** of that program generated on any machine.
- Generally, the **size** of bytecode is less than the **source code**, thus it is easy and fast to transport them over the network/internet.

- Another advantage of bytecode is, one can exchange it to others without disclosing the logic implemented in the program.

BYTECODE VS MACHINE CODE

Bytecode is an intermediate code, it can't be run directly on any **machine** (computer, laptop etc). It must first be converted into machine code before it can be executed. In java, a java **interpreter**(part of **JVM**) converts the bytecode into machine code. **Machine code** is a native or **low-level code** that machine can understand and execute. They are generally binary or hexadecimal instructions which can be executed directly by computers CPU.

DIFFERENCE BETWEEN SOURCE CODE AND BYTECODE

Source codes are code written inside a program by following the syntax of a programming language. In java, code written inside a **.java** file are known as **source codes**. While **bytecode** is the code which is generated after compiling the **java program**. In java, **.class** files are referred as **bytecodes**.

WHAT IS A BYTECODE VERIFIER IN JAVA?

Bytecode verifier is a special java runtime module or program which verifies the **bytecode(.class)** before its run. **Bytecode verifier** traverses the bytecodes and checks if the bytecode loaded are valid and it does not breach java's security restrictions. Think of a **bytecode verifier as a gatekeeper who ensures that the code passed to the java interpreter is in a correct state to be executed and can run without fear of breaking the Java interpreter**. As soon as you run a program, the **classloader(part of JVM)** loads the **.class** file in memory. After that the bytecode verifier verifies if the bytecode is valid or not. If it's valid then only it is passed to java interpreter for further execution.

WHAT IS A JDK JRE AND JVM?

The terms **JDK JRE** and **JVM** are used very often in java programming. Some programmers find it a little difficult to understand what exactly they are, the differences in them and which one is needed for their needs. **JDK JRE** and **JVM** are **java software packages** or **part of the packages** which have **tools** and **libraries** needed to compile and execute **java programs**. Once you install java, these tools and libraries get installed in your PC which allows you to **compile** and run java programs.

WHAT ARE TOOLS IN JAVA?

A **tool** is a program which helps programmer to accomplish a task for example **javac, java, javap** etc are tools. **javac is used to compile a program, java is used to run a program**. If you install java, you can see the **tools(available as .exe files in windows)** inside the **bin** folder of java installation.

SAMPLE JAVA CODE TO BE RUN IN A NOTEPAD

```
public class HelloWorld {  
    public static void main(String[] args) {  
        // Creates a reader instance which takes  
        // input from standard input - keyboard  
        Scanner reader = new Scanner(System.in);  
        System.out.print("Enter a number: ");
```

```
// nextInt() reads the next integer from the keyboard
int number = reader.nextInt();

// println() prints the following line to the output screen
System.out.println("You entered: " + number);
}
}
```

A Java Code to print an Integer entered by a user

```
public class SwapNumbers {
    public static void main(String[] args) {
        float first = 1.20f, second = 2.45f;
        System.out.println("--Before swap--");
        System.out.println("First number = " + first);
        System.out.println("Second number = " + second);

        // Value of first is assigned to temporary
        float temporary = first;

        // Value of second is assigned to first
        first = second;

        // Value of temporary (which has the initial value of first) is assigned to second
        second = temporary;

        System.out.println("--After swap--");
        System.out.println("First number = " + first);
        System.out.println("Second number = " + second);
    }
}
```

A Java code to Swap two numbers using temporary

```
public class Main {
    public static void main(String[] args) {
        // year to be checked
        int year = 1900;
        boolean leap = false;

        // if the year is divided by 4
        if (year % 4 == 0) {
            // if the year is century
            if (year % 100 == 0) {
                // if year is divided by 400
                // then it is a leap year
                if (year % 400 == 0)
                    leap = true;
                else
```



```

        leap = false;
    }
    // if the year is not century
    else
        leap = true;
    }
    else
        leap = false;
    if (leap)
        System.out.println(year + " is a leap year.");
    else
        System.out.println(year + " is not a leap year.");
    }
}

```

A Java Code to Check if its a Leap Year

```

public class StringEg
{
    public static void main(String[ ] args)
    {
        String s1 = "Computer Science";
        int x = 307;
        String s2 = s1 + " " + x;
        String s3 = s2.substring(10,17);
        String s4 = "is fun";
        String s5 = s2 + s4;

        System.out.println("s1: " + s1);
        System.out.println("s2: " + s2);
        System.out.println("s3: " + s3);
        System.out.println("s4: " + s4);
        System.out.println("s5: " + s5);

        //showing effect of precedence
        x = 3;
        int y = 5;
        String s6 = x + y + "total";
        String s7 = "total " + x + y;
        String s8 = " " + x + y + "total";
        System.out.println("s6: " + s6);
        System.out.println("s7: " + s7);
        System.out.println("s8: " + s8);
    }
}

```

A Java Code to show how a string works

```

public class EnhancedFor
{

```

```

public static void main(String[] args)
{
    int[] list = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int sum = sumListEnhanced(list);
    System.out.println("Sum of elements in list: " + sum);

    System.out.println("Original List");
    printList(list);
    System.out.println("Calling addOne");
    addOne(list);
    System.out.println("List after call to addOne");
    printList(list);
    System.out.println("Calling addOneError");
    addOneError(list);
    System.out.println("List after call to addOneError. Note elements of list did not change.");
    printList(list);
}
// pre: list != null
// post: return sum of elements
// uses enhanced for loop
public static int sumListEnhanced(int[] list)
{
    int total = 0;
    for(int val : list)
    {
        total += val;
    }
    return total;
}
// pre: list != null
// post: return sum of elements
// use traditional for loop
public static int sumListOld(int[] list)
{
    int total = 0;
    for(int i = 0; i < list.length; i++)
    {
        total += list[i];
        System.out.println( list[i] );
    }
    return total;
}

// pre: list != null
// post: none.
// The code appears to add one to every element in the list, but does not
public static void addOneError(int[] list)
{
    for(int val : list)
    {
        val = val + 1;
    }
}

```

```

    }

    // pre: list != null
    // post: adds one to every element of list
    public static void addOne(int[] list)
    {
        for(int i = 0; i < list.length; i++)
        {
            list[i]++;
        }
    }

    public static void printList(int[] list)
    {
        System.out.println("index, value");
        for(int i = 0; i < list.length; i++)
        {
            System.out.println(i + " , " + list[i]);
        }
    }
}

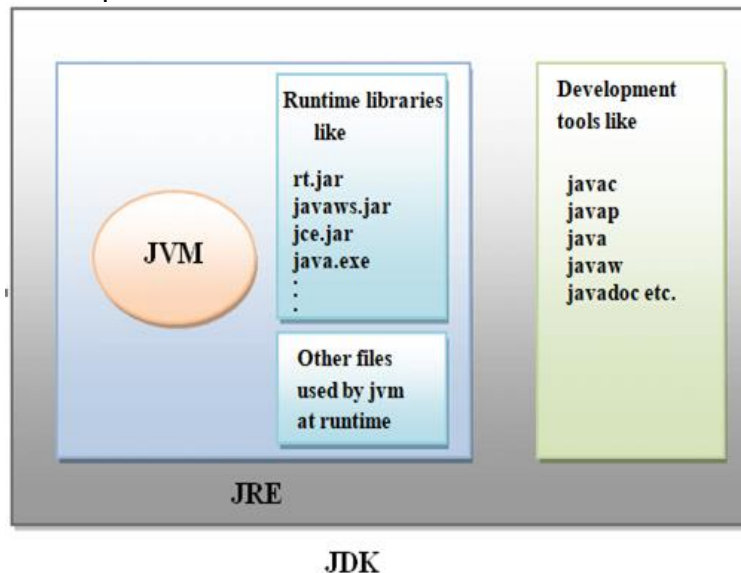
```

WHAT ARE LIBRARIES IN JAVA?

Libraries are a group of **related classes** and **interfaces** contained inside packages. In java, libraries come as **.jar** files. If you extract a **jar** file, you see all the **.class(compiled classes and interfaces)** files included in that jar.

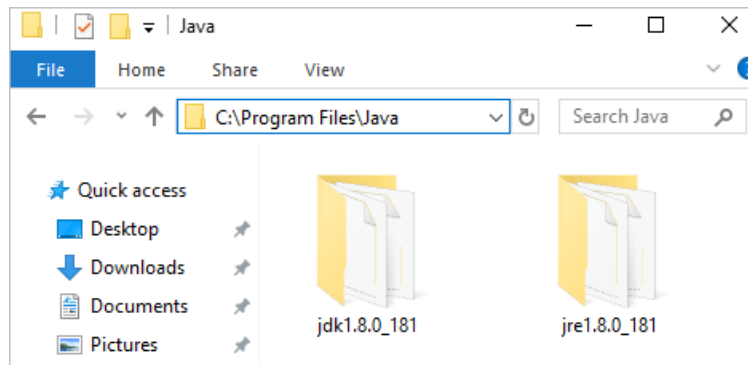
WHAT IS A JDK IN JAVA?

JDK stands for **java development kit**. **JDK** helps programmers in **development** (compilation and **execution**) of java programs. **JDK** a software package that contains tools and libraries needed for compilation and execution of java programs. If you are a programmer then you must need to install **JDK** first in your **PC** in order to start programming in java. For windows OS, this software generally comes in the form of .exe or .zip file.



A **JDK** is a superset of **JRE**, which means it holds the **JRE** in itself. Apart from the **JRE**, it has tools for **developing, debugging, and monitoring java applications**. Some of the significant tools are **java**(Java application launcher), **javac**(java compiler), **javap**(java disassembler), **javadoc**(java documentation tool) etc. **JDK = JRE + Development tools(eg. javac, javap, java, javadoc etc)**

Does JDK exist physically in a PC? Yes, JDK exists physically in computer. If java is installed in your PC, you can see the **jdk** in java installation directory. In windows OS, by default it gets installed in C: directory as given in below image.



Can we run a java program without a JDK? Yes, we can run a program without **JDK** if we have already compiled the program. If it's not compiled, then we can't run it without a **JDK** since our **java compiler(javac)** is in a **JDK**, not in a **JRE**. A **compiled program(.class file)** can be run with **JRE** only.

WHAT IS JRE IN JAVA?

JRE stands for **java runtime environment**. It provides a runtime environment to your program for its execution. As soon as you start running your program using the **java** command, the job of a **JRE** **starts and it ends when either the program ran successfully or a error is thrown**. **JRE** is also a software package which has **JVM, runtime libraries** and other supporting files needed to run a program. **JRE** does not have tools like **javac, javap** etc but it has tools like **java, javaw** etc since they are needed to run java programs. For windows **OS**, a software comes in the form of .exe or .tar file. **JRE** is a superset of **JVM**, as it contains **JVM** in itself. At runtime, **JVM** loads the required runtime libraries and files of **JRE** in memory in order to execute the program and generate the output.

JRE = JVM + runtime libraries(contains packages like util, math, lang etc.)+ runtime files

Does JRE physically exist? Yes, **JRE** also exists physically in a PC. If java is installed in a PC, you can see the **JRE** in the java installation directory. It exists inside the **JDK** folder.

WHAT IS JVM IN JAVA

JVM stands for **java virtual machine**, a virtual machine which loads and executes java programs in memory. When you run a program by using **java** command, a **java virtual machine** is created which has some default size of memory available for loading and executing **classes, objects, variables** etc of a program. The **JVM** loads your **class(program)** and other **required classes and libraries** as well which are needed to execute the program. It's a **virtual machine** because you cannot see this in your computer. Also, it can execute only java programs and applications. **JVM** is not a separate software package, instead it comes as part of **JRE** package. If you have installed **JDK** or **JRE** then you don't have to do anything else for **JVM**. It will be there. Let's see some of the tasks performed by **JVM**:

- Loads code - Loads .class files in memory.
- Verifies code - Verifies the bytecode.
- Interprets code - Converts bytecode into machine specific code.
- Provides runtime environment - Makes availability of dependent classes and libraries.
- Executes code - Executes the code and generate the output.
- Memory management - Manages memory for classes, objects etc. by using garbage collection.

IS JVM PLATFORM INDEPENDENT?

No, **JVM** is platform dependent. There are different **JVM** implementations for different OSs(windows, linux, MacOS etc) from different vendors. JVM along with **bytecode** makes java as WORA(write once, run anywhere) language.

DIFFERENCE BETWEEN JDK JRE AND JVM

JDK	JRE	JVM
JDK stands for java development kit.	JRE stands for java runtime environment.	JVM stand for java virtual machine.
JDK is a software package.	JRE is also a software package.	JVM is not a separate software package, it's part of JRE.
Using JDK you can do both, compilation and execution of java programs.	Using JRE you can do only execution of java programs.	JVM is part of JRE, responsible for execution of your java programs.
JDK exists physically in your computer.	JRE also exists physically in your computer.	JVM does not exist physically in your computer.
You can download and install JDK in your computer.	JRE can also be downloaded and installed in your computer.	JVM comes as part of JRE.
JDK is a superset of JRE.	JRE is a subset of JDK but superset of JVM.	JVM is a subset of JRE.
JDK contains tools like javac , javap , java , javadoc etc.	JRE contains runtime tools like java , javaw etc.	JVM doesn't contains any such tools.

BASIC EXAMPLE PROGRAMS IN JAVA

Here are some basic examples of java programs, especially for beginners to get some more idea about execution flow in java programs. The programs given below demonstrates the execution flow of java programs.

```
class FirstProgramFlow {
    public static void main(String [] args) {
        System.out.println("first ");
        System.out.print("second ");
    }
}
```



```

        System.out.println("third ");
    }
}

```

As you can see, the program executes from top to bottom starting from **main method**. It executes first line then moves to second line, executes second line then move to third line and so on.

WHAT IS THE DIFFERENCE BETWEEN PRINT AND PRINTLN IN JAVA?

println("...") method prints a string in it and moves the **execution pointer** to the new line, while **print("...")** method prints the string inside it and keeps the pointer in same line. This program will give you more idea about execution flow in java.

```

class SecondProgramFlow {
    public static void main(String [] args) {
        // calling firstMethod
        firstMethod();
        System.out.println("main method");
    }
    public static void firstMethod() {
        System.out.println("first method");
        // calling secondMethod and assigning return value in i
        int i = secondMethod();
        System.out.println("After second method call, i = "+i);
    }
    public static int secondMethod() {
        System.out.println("second method");
        return 1;
    }
}

```

Once a **method** is called, it finishes all its statement unless a **return** or method call statement is encountered. If a **return** statement is encountered, the execution moves back to the line from where this method was called and if a method call is encountered, the execution moves into the new method.

WHAT IS COMMAND LINE ARGUMENTS IN JAVA

A **command line argument** is an argument passed onto a program while running that program using **java** command. The argument gets stored in the **args** (the name can be different) parameter of the main method. You can access your argument using **args** parameter inside main method. E.g. if you want to pass a command line argument *"firstArgument"* to a program *MyFirstProgram.java* given below, you need to run the program using **java** command like below:

```
java MyFirstProgram firstArgument
```

Here **firstArgument** is the argument passed to **MyFirstProgram** and it's stored in the **args** variable of main method. Anything that is passed after the class name in **java** command acts as command line argument and can be accessed using **args** variable.

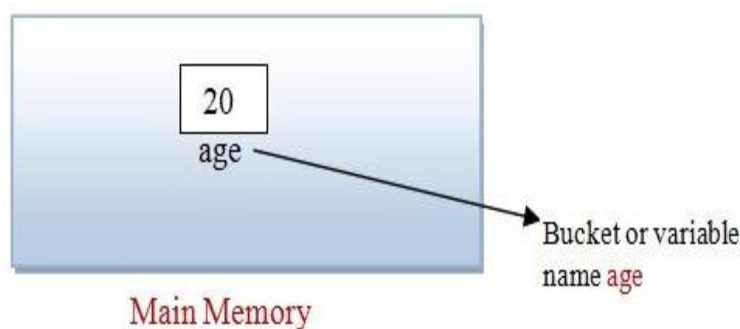
WHAT IS A VARIABLE?

Variables in computer are memory locations used to store data of a particular data type. Think of variable as a “**Bucket**” or “**Envelope**” where you can store data. This bucket has a name called as **variable name**. The information stored in a bucket can be accessed later in program by it's name. They are called variables because the data stored in it can be changed. Basic **syntax** of declaring a variable in java is:

Data_Type variable_name = value;

In place of the **Data_Type** you specify the **data type** of the variable, after the name of the variable and then the variable value at right side of the = **operator** with a **semicolon(;) at the end of the declaration**. For example:

int age = 20; // variable name age with data type as int and value as 20



Now let's understand the different components of a variable declaration:

- Data_Type of variable - **Mandatory**
- Name of variable - **Mandatory**
- Value of variable - **Optional**

● **Each variable** must have a **Data Type**, it can be a **primitive** or a **non-primitive data type**. The data type of a variable represents what type of data that variable can store. E.g. **int** data type can store integer type values like 10, 20, 1000 etc while **float** data type can store float type values like 1.2, 10.5, 0.25 etc.

● **Every variable** must also have a **Name**. The name of a variable is used to refer that **variable value** inside the program. The name of a variable must be **unique**. There are some rules and convention for writing variable names, you will get the detail about this in naming convention for identifiers tutorial.

● **Every variable** will also have a **value**, either it is given by programmer or a default value is assigned by java as per it's data type. So, if you declare a variable like below java will assign a default value.

int num1; Here we have not assigned any value to variable num1, java assigns a default value which is 0 for **int** data type.

VARIABLE DECLARATION AND INITIALIZATION IN JAVA

Declaration and **initialization** of variable are terms used often with variables. **Declaration** means declaring or creating a variable while **Initialization** means assigning some value to that variable.

int speed; // Declaring or creating a variable name speed
 speed = 100; // Initializing variable speed with value as 100

```

int num1, num2; // Declaring variable num1 and num2
num1 = 10; num2 = 20 // Initializing variable num1 and num2
int count = 10; // Declaration and initialization together of variable count
double a= 10.5, b = 20.25; // Declaration and initialization together of variable a and b

```

WHAT IS DIFFERENCE BETWEEN DECLARATION AND INITIALIZATION?

In declaration we only tell the **name** and **data type** of the variable that we are going to create, we don't specify any value to that variable. While in initialization we provide some value to the declared variable. For example, `int speed;` is a declaration while `speed = 100` is initialization.

VARIABLE PROGRAM IN JAVA

```

class VariableProgram {
    public static void main(String [] args) {
        // declaring single variable
        int a = 20;
        // declaring multiple variable
        int b = 30, c = a, d;
        d = a + b;
        System.out.println("d = " +d);
        double pi = 3.14159;
        System.out.println("value of pi = " +pi);
        boolean e = true, f = a>c;
        System.out.println("e = "+e+ "\nf = "+f);
    }
}

```

The following are some of the common mistakes in declaring variables in java:

```

int a = 20 // missing semicolon(;) at the end. Correct is -> int a = 20;
int b c; // two variables must be separated by comma(,) if declared in same line. Correct is -> int b, c;
int d = 20.5; // Can't assign float value in int data type
int p, int q; // Correct is -> int p; int q; or int p, q;
int p = int q; // Right side of = operator must be an expression or value, not variable declaration

```

Variables are declared in diverse program places e.g., **class, methods, constructors, blocks** etc. As per the declaration places, variables are divided in different types. We can use **access modifiers** with **variable declarations** which decides the accessibility of variables in java.

VARIABLE TYPES IN JAVA WITH EXAMPLE

These notes explain different types of variables in java, differences between them and their allocations in memory. They help beginners to use correct **variable type** for their need in programs. In java there are three categories of variables:

1.Local variable **2.Instance variable** **3.Static or Class variable**

JAVA LOCAL VARIABLE

Variables defined in **methods, constructors** or **blocks** are called **local variables**. They are created in **memory(stack memory)** when execution of that method, constructor or block starts

in which they are declared. These variables are accessible only within the method, constructor or block in which they are created. **Local variables** must be **initialized** with some value before their use otherwise compiler throws an error.

HOW TO DECLARE AND ACCESS LOCAL VARIABLE IN JAVA

The declaration of local variable is same as **normal variable**, the only thing is that they are declared in method, constructor or block. In other way if you have declared variables in method, constructor or block they are basically the local variables in java. To access a local variable, you just have to use it's name. The program below shows how to create and access local variables in java.

LOCAL VARIABLE PROGRAM IN JAVA

```
class LocalVariable {
    public LocalVariable() {
        int a = 10; // local variable a, created inside constructor
        System.out.println("a = "+a);
    }
    {
        int b = 30; // local variable b, created inside block
        System.out.println("b = "+b);
    }
    public static void main(String [] args) {
        int c = 20; // local variable c, created inside method
        System.out.println("c = "+c);
        // Constructor and block will be executed by below line
        LocalVariable lv = new LocalVariable();
        int d; // local variable d, created inside method
        // Below code will throw error if uncommented, as d is not initialized.
        // System.out.println("d = "+d);
    }
}
```

When memory is allocated for local variables in Java? Memory for local variables is allocated as soon as the method, constructor or block containing the variable is executed. What is a local variable default value in Java? Local variables don't have any default values in java, they must be initialized before they can be used.

JAVA INSTANCE VARIABLE

Variables defined inside a class but outside any **method, constructor or block** are known as **instance variable**. **Initialization** of **instance variable** is not compulsory, if not initialized, these variables will have default values as per their data type. **Instance variables**, belongs to the **instance** or **object** of a class. **Instance variables** are created in **memory(heap memory)** as soon as an **object** of the class holding the instance variable is created. Every **object** of a class has its own copy of the instance variables, changes made to the variable value of one object doesn't reflect in other object.

HOW TO DECLARE AND ACCESS INSTANCE VARIABLE IN JAVA.

The declaration of an instance variable is same as normal variable, the only difference is that they are declared inside class directly. if you declare variables inside a class directly, they are basically instance variables. **Instance variables** are accessed using object name like **objName.instanceVarName**. You can also access the instance variable directly as well, in this case java internally applies the object name for you. **The good practice is to always use the object name for accessing the instance variable.** The program below shows how to declare and access instance variables in java.

INSTANCE VARIABLE PROGRAM IN JAVA

```
class InstanceVariable {
    int age = 20; // instance variable age, created inside class
    String name = "Rahul"; // instance variable name, created inside class
    public static void main(String [] args) {
        InstanceVariable iv = new InstanceVariable();
        System.out.println("Age = " +iv.age);
        System.out.println("Name = " +iv.name);
    }
}
```

CLASS VARIABLE IN JAVA

Variables defined inside a **class**(not inside a method, constructor or block) using **static** keyword are known as **static** or **class variables**. **Initialization** of a class variable is also not compulsory, if not initialized, these variables will have default values as per their data type. They are known as **class variables** since they belong to a **class**, not **objects**. **Class variables** as the name suggests, belongs to class and is common to all objects of that class. These variables are created in memory as soon as the class containing the variable is loaded inside memory.

HOW TO DECLARE AND ACCESS CLASS VARIABLE IN JAVA

A declaration of a **static** or a **class** variable is similar to an **instance variable**, except that they are declared along with the **static** keyword. **Static variable** can be accessed using the class name as **className.staticVarName** or use any of the object instances like **objName.staticVarName** . The program below shows how to declare and access class variable in java.

CLASS VARIABLE PROGRAM IN JAVA

```
class ClassVariable {
    static String classVar = "Class variable Test"; // class variable classVar
    public static void main(String [ ] args) {
        System.out.println(ClassVariable.classVar);
    }
}
```


DIFFERENCE BETWEEN **LOCAL**, **INSTANCE** AND **STATIC VARIABLES** IN JAVA

FEATURES	LOCAL VARIABLE	INSTANCE VARIABLE	STATIC VARIABLE
Declaration point	Inside a method, constructor, or block.	Inside the class.	Inside the class using static keyword.
Lifetime	<p>Created when method, constructor or block is entered or executed.</p> <p>Destroyed on exit of method, constructor or block.</p>	<p>Created when instance of class is created with new keyword.</p> <p>Destroyed when object is available for garbage collection.</p>	<p>Created when the program starts.</p> <p>Destroyed when the program execution completed.</p>
Scope/Visibility	Visible only in the method, constructor or block in which they are declared.	Visible to all methods in the class. It could be visible to other classes as well according to its access modifier.	Same as instance variable
Initial value	None. Must be assigned a value before its use.	Default value as per Data Type. 0 for integers, false for booleans, or null for object references etc.	Same as instance variable.
Use of Access Modifier	Access modifier(public , private , protected) can not be used with local variables	Can be used.	Can be used
Access from	Not possible, Local variable can be accessed within the method,	Instance variables can be accessed outside	Same as instance variable. It can be

outside	constructor or block in which they are declared.	depending upon it's access modifier. It can be accessed using object of that class.	accessed using object or class name.
Memory Area	Stack memory.	Heap memory	Heap memory
Use	Used for local computations	Used when value of variable must be referenced by more than one method	Generally used for declaring constants

JAVA PROGRAM OF LOCAL, INSTANCE AND STATIC VARIABLE

```

class Student
{
    static String college = "SUMVS"; // static or class variable college
    int rollNo; // instance variable rollNo
    String name; // instance variable name

    public static void main(String [] args)
    {
        Student s1 = new Student(); // local variable s1
        s1.rollNo = 1;
        s1.name = "Ram";

        Student s2 = new Student(); // local variable s2
        s2.rollNo = 2;
        s2.name = "Shyam";

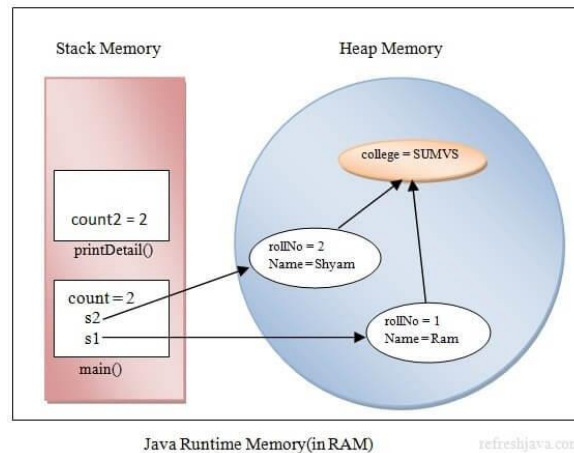
        s1.printDetail();
        s2.printDetail();

        int count = 2; // local variable count
        System.out.println("Total students = "+ count);
    }
    void printDetail()
    {
        int count2 = 2; // local variable count2
        System.out.println(rollNo + ", " + name + ", " + Student.college + ", " + count2);
    }
}

```

MEMORY ALLOCATION FOR LOCAL, INSTANCE AND STATIC VARIABLES IN JAVA

The image below shows where the memory is allocated for the different types of java variables used in above program. The variable *count* and *count2* are local variables, *rollNo* and *Name* are instance variables whereas *college* is a class variable.



CAN A LOCAL VARIABLE HAVE THE SAME NAME AS AN INSTANCE VARIABLE OR STATIC VARIABLE? Yes, you can declare a local variable with same name as an instance or static variable. When accessed within the local block, method or constructor, local variable will take precedence over instance or static variable.

JAVA VARIABLE NAMING RULES AND CONVENTIONS

Identifiers are the names of **variables, methods, classes, packages and interfaces**. The name of **variable identifiers** must be written according to the rules and conventions given by the programming language. **Rules** are mandatory to follow while **conventions** are optional, but it's good to follow them. Like every programming language, java has it's own set of rules and conventions that we should follow while writing the identifiers name.

The rules and conventions for naming your variables in java can be summarized as follows :

- Every **variable name** should start with either **alphabets, underscore (_)** or **dollar (\$)** symbol. It should **not start with number** or any **other special symbols**. But, the convention is to always begin the variable names with a letter, not "\$" or "_". Conventionally the dollar sign should never be used at all. So variable names like `age`, `_age`, `$age` are valid names in java.
- Variable names are **case-sensitive**. Names like `age`, `Age`, `aGe`, `agE` are different variables. These names can be used together in same program, but doing so is not a good practice.
- **Spaces are not allowed in the variable names**. For eg. `int bike speed = 100;` is not a valid declaration. It should be like `int bikespeed = 100;` or `int bike_speed = 100;` etc.
- Other characters apart from first character can be alphabets, numbers, \$, or _ characters. For eg. `user_age`, `a12$_ge`, `_$1age2` etc are valid names of variables.
- When choosing a name for your variables, the convention is to use full words instead of short forms, doing so will make your code easier to read and understand. Names like `age`, `height` and `speed` are easier to understand than using short forms like `a`, `h` and `s`.
- The variable name you choose must not be a keyword(reserved word) of java. Java has defined some keywords like `int`, `double`, `float`, `char`, `if`, `else` etc. These keywords cannot be used as your variable names. Refer java keyword table given below to see the list of all keywords in java.

- Variable name should always exist in the left-hand side of assignment operator. `int height = 5` is correct while `5 = int height` is incorrect.
- One more convention is, if the name you choose consists of only one word, use that word in lower case letters. If it consists of more than one word, use first letter of each subsequent word as capital letter. The variable names like `userName`, `minHeight` and `maxHeight` are good examples of this convention.

LIST OF JAVA KEYWORDS

Table below shows a list of all java keywords that programmers cannot use for naming their variables, methods, classes etc.

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

The keywords `const` and `goto` are reserved, but they are not used. The words `true`, `false`, and `null` might seem like keywords, but they are actually literals, you cannot use them as identifiers in your programs.

What is literal in java programming? Any constant value which can be assigned to the variable or can be used in an expression is called literal/constant. For example 10, 20.5f, 'a', `true`, `"RefreshJava"` etc are literals. **How many keywords are there in Java?** In java there are 50 keywords(reserved words). Refer the table above to see all the keywords in java. **Can keywords be used as variable names in Java?** **No**, you can't use keywords as your variable name in java, doing so will result in compilation error. **Can I use java keywords in upper case as my variable names?** **Yes** you can definitely use, as keywords in java are case-sensitive but that's not a good programming style. If you want you can use them along with other words like `intVal`, `floatVal`, `charValue`, `doubleVar` etc are valid identifier names.

JAVA IDENTIFIER RULES PROGRAM

```
class NamingConvention {
    public static void main(String [] args) {
        byte $num1=20, _num2 = 10, num3$_ = 25;
        System.out.println("$num1 = "+ $num1 +", _num2 = "+ _num2 +", num3$_ = "+ num3$_);
        byte age=20, Age = 30, agE = 40;
        System.out.println("age = "+ age +", Age = "+ Age +", agE = "+ agE);
    }
}
```

```

int For = 15, If = 30, Byte = 50;
System.out.println("For = "+ For +", If = "+ If +", Byte = "+ Byte);
int bike_speed = 50;
System.out.println("bike_speed = "+ bike_speed);
int minSpeed = 10, maxSpeed = 80;
System.out.println("minSpeed = "+ minSpeed +", maxSpeed = "+ maxSpeed);
}
}

```

Following are some incorrect variable naming which will result in compile time error.

`int 2num1 = 40, 12num = 10;` // digits not allowed as first character

`int @num = 40, &num = 10;` // only \$ and _ are allowed as first character

`int bike speed = 30;` // Space cannot be used in variable name

`int byte = 20, float = 30, for = 40;` // keywords cannot be used as variable names

PRIMITIVE AND NON-PRIMITIVE DATA TYPES IN JAVA

Data is simply an instruction, it could be 123, -125, 3.14, "hello" etc. A **data type** is classification of these data. Data can be divided in different data types like integer number, float number, character etc. In java there are basically two types of data types: **Primitive Data Type & Non Primitive Data Type**

PRIMITIVE DATA TYPE

Primitive data type deals on basic data like 123, -125, 3.14 etc. Primitive data types are predefined by the java language itself. Considering size and type of data, java has defined eight types of primitive data type.

Type	Size	Range	Default Value	Example
byte	1 byte	-128 to 127	0	<code>byte b = 123;</code>
short	2 byte	-32768 to 32767	0	<code>short s = 1234;</code>
int	4 byte	-2^{31} to $2^{31}-1$	0	<code>int i = 123456;</code>
long	8 byte	-2^{63} to $2^{63}-1$	0L	<code>long l = 312456L;</code> <code>long ll = 312456L;</code>
float	4 byte	1.4E-45f to 3.4028235E38f	0.0f	<code>float f = 123.45f;</code> <code>float ff = 123.45F;</code>
double	8 byte	4.9E-324 to 1.7976931348623157E308	0.0d	<code>double d = 1234.67d;</code> <code>double dd = 1234.67D;</code>
char	2 byte	'\u0000' (or 0) to '\uffff' (or 65,535 inclusive)	'\u0000'	<code>char c = 'C';</code>
boolean	represents 1 bit of information	Not Applicable	false	<code>boolean b = true;</code>

Data types `byte`, `short`, `int`, `long` is also known as **Integer** data type because they can contain only integer type values while **data types** `float`, `double` is also known as **Floating** point data type because they are used to store float type values.

HOW DO YOU FIND THE RANGE OF INTEGER DATA TYPES IN JAVA?

The range of integer data type values is calculated as $-(2^{n-1})$ to $(2^{n-1})-1$, where n is the number of bits required. For example the **byte** data type requires 1 byte(8 bits). Therefore, the range of values that can be stored in **byte** data type is $-(2^{8-1})$ to $(2^{8-1})-1 = -2^7$ to $(2^7) - 1 = -128$ to 127

Primitive types are also called **value types** in java since variables of primitive data types directly hold the values rather than the reference(address) of that value in memory. That is why if you access a **primitive type** variable, you directly get value rather than the reference of that value in memory.

PRIMITIVE DATA TYPE PROGRAM IN JAVA

```
class PrimitiveDataTypeExample {
    public static void main(String[] args) {
        byte byteVar = 123;
        short shortVar = 1234;
        int intVar = 123456;
        long longVar = 3124567891L; // Must end with l or L
        float floatVar = 123.45f; // Must end with f or F
        double doubleVar = 12345.6789d; // d or D is optional
        double doubleVar2 = 125.67;
        boolean booleanVar = true;
        char charVar = 65; // code for A
        char charVar2 = 'C';

        // Some incorrect declaration
        // long longVar2 = 34287; Must end with l or L
        // float floatVar2 = 123.45; Must end with f or F

        System.out.println("byteVar = " + byteVar);
        System.out.println("shortVar = " + shortVar);
        System.out.println("intVar = " + intVar);
        System.out.println("longVar = " + longVar);
        System.out.println("floatVar = " + floatVar);
        System.out.println("doubleVar = " + doubleVar);
        System.out.println("doubleVar2 = " + doubleVar2);
        System.out.println("booleanVar = " + booleanVar);
        System.out.println("charVar = " + charVar);
        System.out.println("charVar2 = " + charVar2);
    }
}
```

WHICH DATA TYPE IS BEST FOR STORING INTEGER NUMBERS?

It completely depends on an integer number. As a good programming style always prefer lower size data type over higher size data type if the value that need to be stored comes within the range of that data type. For eg. any value from range -128 to 127 can be stored in **short**, **byte**, **int**, **long** data type but prefer **short** data type over others as it takes less space in memory.

NON PRIMITIVE DATA TYPE

Non-primitive data types are generally created by the programmer. In java every class or interface acts like a data type. **Classes, interfaces, arrays** etc defined by java or by programmer is basically the non-primitive data type. **Non primitive data types** are built using primitive data types, for example classes with primitive variables. **Variables of non-primitive** data type don't contain the value directly, instead they contain a **reference**(address) to an object in memory. That is why non primitive types are also called as **reference types**. If you access a non-primitive variable, it will return a reference not the value. The objects of non-primitive type may contain any type of data, primitive or non-primitive. Except primitive data type everything is non-primitive data type in java. **If I create a class MyFirstProgram, would it be a non-primitive data type? Yes**, **MyFirstProgram** will be a non-primitive data type.

Is string a primitive data type in java? No, String is a non-primitive data type as it stores references.

Why String is not a primitive data type? Because String is a class defined in java language, it has it's own methods to manipulate and operate over object of String class. Every class in java, whether it is defined by java or by programmer is a non-primitive data type.

NON PRIMITIVE DATA TYPE PROGRAM IN JAVA

```
class NonPrimitiveDataTypeExample {
    public static void main(String[] args) {
        // String is a non primitive data type define in Java
        String nonPrimStr = "String is a non primitive data type";
        System.out.println("nonPrimStr = "+ nonPrimStr);
        // Integer is a non primitive data type define in Java
        Integer intVal = new Integer(10);
        System.out.println("intVal = "+ intVal);
        // This class itself is a non primitive data type
        NonPrimitiveDataTypeExample np = new NonPrimitiveDataTypeExample();
        System.out.println("np = "+ np.toString());
    }
}
```

Note: The value of np variable in last line of output could be different in your case as it depends on specific machine. What are examples of java non primitive data types? Any class or interface created by you or already created in java are non-primitive data types. Some of the examples of non-primitive data type in java are String, StringBuilder, Arrays, Integer, Character etc.

DIFFERENCE BETWEEN PRIMITIVE AND NON-PRIMITIVE DATA TYPES IN JAVA

Primitive data types are predefined by java itself while **non primitive data types** are the classes or interfaces defined by the programmer or java. **Variables of primitive** type holds the values directly, if you access a **primitive type variable** you get the value directly. **Variables of non-primitive** types holds the **references(address)**, if you access a **non-primitive type variable** you get a reference. **Primitive data types** are also known as **value types** since the variable of such types holds the values directly while **non primitive data types** are also known as **reference types** since variable of such types holds the references, not the values. Every primitive type variable has a **default value**, if no value

is assigned whereas a **non-primitive type** variable points to null if no object is assigned. Non primitive type variables can call methods of that type(class or interface) while it's not possible with primitive variables.

WHAT IS OPERATOR IN JAVA

In computer programming languages, **an operator** is a special symbol which tells the computer to perform specific operation on one, two or three operands and then return a result.

WHAT IS OPERAND IN JAVA?

An **operand** is simply a value on which the operation is performed. This value can be in the form of constant, variable, expression or any method which returns a value. E.g. in expression $2+3$, 2 and 3 are operands and $+$ is an operator that performs addition operation. Also in expression $a+b$, a and b are operands and $+$ is an operator.

WHAT IS AN EXPRESSION IN JAVA?

An **expression** is a construct made up of variables, operators and method invocations, these expressions are constructed as per the syntax of the programming language and evaluates to a single value. For example $a+b$, $a+b*c$, $a = 10$, $c = a + b$, $a > b$, $\text{sqrt}(4)$, $(a+b)*(c-d)$ etc. An expression may contain sub-expressions within it.

DIFFERENT TYPES OF OPERATORS IN JAVA

There are following types of operators in java, you can refer the specific tutorial of these operator types to see the details like what all operators are, what they do and their example programs.

- Assignment operator (=)
- Arithmetic Operators (+, -, *, /, %)
- Unary Operators (+, -, ++, --, !)
- Relational Operators (==, !=, >, >=, <, <=)
- Conditional Operators (&&, ||, ?:)
- Bitwise Operators (~, &, ^, |)
- Bit Shift Operators (<<, >>, >>>)

Operators can also be categorized by the number of operands they operate on. Most operator requires one or two operands except ternary operator which requires three operands. So we can categorize above operators in three types as well: **UNARY OPERATOR**, **BINARY OPERATOR** **TERNARY OPERATOR**

WHAT IS UNARY OPERATOR IN JAVA?

An operator which operates on **one operand** is known as **unary operator**. For example $+$, $-$, $++$ etc are unary operators, usage example $+2$, -3 , $++a$ etc.

WHAT IS BINARY OPERATOR IN JAVA?

An operator which operates on **two operands** is known as **binary operator**. For example $+$, $-$, $*$ etc are binary operators, usage example $2+3$, $5-3$, $a*b$ etc.

WHAT IS TERNARY OPERATOR IN JAVA?

An operator which operates on **three operands** is known as ternary operator. There is only one **ternary operator** in java which is $?:$, also known as if then else operator.

JAVA OPERATORS LIST

The table below shows the list of all operators in java which you can use in your programs. For this table let's assume we have operands or variables a , b and c .

OPERATOR SYMBOL	OPERATOR NAME	EXAMPLE	TYPE OF
=	Assignment operator	a = b;	Assignment Operator
+	Additive operator	a+b	Arithmetic Operator
-	Subtraction operator	a-b	
*	Multiplication operator	a*b	
/	Division operator	a/b	
%	Remainder or Modulo operator	a%b	
+	Unary plus operator	+a, +5	Unary Operator
-	Unary minus operator	-a, -3	
++	Increment operator	a++, ++a	
--	Decrement operator	a--, --a	
!	Logical complement operator	!(a>b)	
==	Equal to operator	a==b	Relational Operator
!=	Not equal to operator	a!=b	
>	Greater than operator	a>b	
>=	Greater than or equal to operator	a>=b	
<	Less than operator	a < b	
<=	Less than or equal to operator	a<=b	
&&	Conditional-AND operator	(a==1 && b==5)	Conditional Operator
 	Conditional-OR operator	(a==1 b==5)	
?:	Ternary operator	c = a>b ? a : b	

~	Unary bitwise complement operator	~a	Bitwise Operator
&	Bitwise AND	a&b	
^	Bitwise exclusive OR operator	a^b	
	Bitwise inclusive OR operator	a b	
<<	Signed left shift operator	a << 2	Bit Shift Operator
>>	Signed right shift operator	a >> 2	
>>>	Unsigned right shift operator	a >>> 2	

Note : Apart from above operators, the **instanceof** keyword in java is also considered as an operator. It compares an object to a specified type(**class**). **Note :** Spaces are not allowed inside multi-character(eg. &&, >=, == etc) operators. For example a > = b, a = = b, a & & b etc are incorrect and will result in compilation error.

HOW MANY OPERATORS DOES JAVA HAVE?

There are 28 operators in java including the **instanceof** operator.

If an expression has many operators, then it must be evaluated as per the operator precedence in java.

OPERATOR PRECEDENCE IN JAVA WITH EXAMPLE

When we solve an expression with many operators, we follow some rules. E.g. an expression **2+3*4**, we do **multiplication first** before addition since the **multiplication operator** has higher precedence than addition operator. The same applies in java. In java, **operator precedence is a rule that tells us the precedence of different operators**. Operators with **higher precedence** are evaluated before those operators with lower precedence. E.g., in an expression **a+b*c**, the operator ***** will be evaluated before **+** operator, since the **operator *** has higher precedence than **+** operator. Table below shows the precedence of operators in decreasing order,

JAVA OPERATOR PRECEDENCE TABLE

Operators	Precedence	Example
Postfix	<i>expr++ expr--</i>	<i>a++ , a--</i>
Unary	<i>++expr --expr +expr -expr ~ !</i>	<i>++a , --a , !a</i>
Multiplicative	<i>* / %</i>	<i>a*b , a/b , a%b</i>
Additive	<i>+ -</i>	<i>a+b , a-b</i>
Shift	<i><< >> >>></i>	<i>a<<2 , a>>1</i>
Relational	<i>< > <= >= instanceof</i>	<i>a<2 , a>1</i>
Equality	<i>== !=</i>	<i>a==b , a!=b</i>
Bitwise AND	<i>&</i>	<i>a&b</i>
Bitwise exclusive OR	<i>^</i>	<i>a^b</i>
Bitwise inclusive OR	<i> </i>	<i>a b</i>
Logical AND	<i>&&</i>	<i>a&&b</i>
Logical OR	<i> </i>	<i>a b</i>
Ternary	<i>? :</i>	<i>a = a>2 ? a : b</i>
Assignment	<i>= += -= *= /= %= &= ^= = <<= >>= >>>=</i>	<i>a=b, a+=b, a/=b, a>>=2</i>

A programmer should **remember** bellow rules while evaluating an expression in java.

- The operands of operators are evaluated from left to right. For example, in expression **++a + b--**, the operand **++a** is evaluated first then **b--** is evaluated.
- Every operand of an operator (except the conditional operators **&&**, **||**, and **? :**) are evaluated completely before any part of the operation itself is performed. For example in expression **++a + b--**, the addition operation will take place only after **++a** and **b--** is evaluated.
- The left-hand operand of a binary operator are evaluated completely before any part of the right-hand operand is evaluated.
- Order of evaluation given by parenthesis **()** get's preference over operator precedence.

- The prefix version of ++ or -- evaluates/uses the incremented value in an expression while postfix version of ++ or -- evaluates the current value, then increases/decreases the operand value.
- All binary operators are evaluated from left to right except assignment operator.

What changes the precedence of the operators in Java? Parentheses "()" are used to alter the order of evaluation. E.g. in an expression **a+b*c**, if you want the addition operation to take place first, then rewrite the expression as **(a+b)*c**.

Java Program of Operator Precedence

```
class OperatorPrecedence {
    public static void main (String[] args) {
        int result = 0;
        result = 5 + 2 * 3 - 1;
        System.out.println("5 + 2 * 3 - 1 = " +result);
        result = 5 + 4 / 2 + 6;
        System.out.println("5 + 4 / 2 + 6 = " +result);
        result = 3 + 6 / 2 * 3 - 1 + 2;
        System.out.println("3 + 6 / 2 * 3 - 1 + 2 = " +result);
        result = 6 / 2 * 3 * 2 / 3;
        System.out.println("6 / 2 * 3 * 2 / 3 = " +result);
        int x = 2;
        result = x++ + x++ * --x / x++ - --x + 3 >> 1 | 2;
        System.out.println("result = " +result);
    }
}
```

1.ASSIGNMENT OPERATOR IN JAVA WITH EXAMPLE

Assignment operator is one of the simplest and most used operators in java programming language. As the name itself suggests, the assignment operator is used to assign value inside a variable. In java we can divide assignment operator in two types:

- Assignment operator or simple assignment operator
- Compound assignment operators

WHAT IS ASSIGNMENT OPERATOR IN JAVA

The = operator in java is known as assignment or simple assignment operator. It assigns the value on its right side to the operand(variable) on its left side. For example:

```
int a = 10; // value 10 is assigned in variable a
double d = 20.25; // value 20.25 is assigned in variable d
char c = 'A'; // Character A is assigned in variable c
a = 20; // variable a is reassigned with value 20
```

The left-hand side of an assignment operator must be a variable while the right side of it should be a value which can be in the form of a constant value, a variable name, an expression, a method call returning a compatible value or a combination of these.


```

int a = 10, b = 20; // right side can be a constant value
int c = a; // right side can be a variable
int d = a+b; // right side can be an expression
int e = sum(a,b) // right side can be a method(sum) call
// Following are incorrect declarations
a + b = 20; // Left side of = operator can't be an expression
1 = a; // Left side of = operator must be a variable

```

A value at right side of assignment operator must be compatible with the data type of left side variable, otherwise compiler will throw compilation error. Following are incorrect assignment:

```

int a = 2.5; // value 10.5 can't be assigned in int variable
char c = 2.5; // value 2.5 can't be assigned in char variable

```

Another important thing about assignment operator is that, it is evaluated from **right to left**. If there is an expression at right side of assignment operator, it is evaluated first then the resulted value is assigned in left side variable.

```

int a = 10, b = 20, c = 30;
int x = a + b + c; // value of x will be 60
a = b = c; // value of a and b will be 30

```

Here in statement **int x = a + b + c;** the expression **a + b + c** is evaluated first, then the resulted value(**60**) is assigned into **x**. Similarly in statement **a = b = c**, first the value of **c** which is **30** is assigned into **b** and then the value of **b** which is now **30** is assigned into **a**. A variable on the **left side** of an assignment operator can also be a non-primitive variable. E.g. if we have a class **MyFirstProgram**, we can assign object of **MyFirstProgram** class using **=** operator in **MyFirstProgram** type variable.

```

MyFirstProgram mfp = new MyFirstProgram();
Is == an assignment operator ?

```

No, it's not an assignment operator, it's a relational operator used to compare two values. **Is** **assignment operator** a **binary operator**. **Yes**, as it requires two operands.

ASSIGNMENT OPERATOR PROGRAM IN JAVA

```

class AssignmentOperator {
    public static void main (String[] args) {
        int a = 2;
        int b = a;
        int c = a + b;
        int d = sum(a,b);
        boolean e = a>b;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}

```

```

        System.out.println("e = " + e);
    }
    static int sum(int x, int y) {
        return x+y;
    }
}

```

JAVA COMPOUND ASSIGNMENT OPERATORS

The assignment operator can be mixed or compound with other operators like addition, subtraction, multiplication etc. We call such assignment operators as compound assignment operator. E.g.:

```

int a = 10, b = 20;
a += 10; // is same as a = a + 10; // += is a compound assignment
b *= 5;  // is same as b = b * 5; // *= is a compound assignment

```

Here the statement **a += 10;** is a short version of **a = a + 10;** the operator **+=** is addition compound assignment operator. Also, **b *= 5;** is short version of **b = b * 5;** the operator ***=** is multiplication compound assignment operator. The compound assignment can be in more complex form as well, like below:

```

a -= a+b; // is same as a = a - (a+b);
b *= a*b+c; // is same as b = b * (a*b+c);

```

Operator	Example	Same As
=	a = 10	a = 10
+=	a += 5	a = a + 5
-=	a -= 3	a = a - 3
*=	a *= 6	a = a * 6
/=	a /= 5	a = a / 5
%=	a %= 7	a = a % 7
&=	a &= 3	a = a & 3
=	a = 3	a = a 3
^=	a ^= 2	a = a ^ 2
>>=	a >>= 3	a = a >> 3
>>>=	a >>>= 3	a = a >>> 3
<<=	a <<= 2	a = a << 2

How many assignment operators are there in Java? Including simple and compound assignment we have total **12** assignment operators in java as given in above table. **What is shorthand operator in Java?** Shorthand operators are not new, they are just a shorter way to write something that is already available in java language. E.g. the code **a += 5** is shorter way to write **a = a + 5**, so += is a shorthand operator. In java all compound assignment operators and the **increment / decrement** operators are basically shorthand operators.

COMPOUND ASSIGNMENT OPERATOR PROGRAM IN JAVA

```
class CompoundAssignmentOperator {
    public static void main (String[] args) {
        int a = 10;
        a += 10;
        int b = 100;
        b -= 20;
        int c = 3;
        c *= 10;
        short s = 200;
        s &= 100;
        short s2 = 100;
        s2 ^= 10;
        byte b2 = 127;
        b2 >>= 3;

        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("s = " + s);
        System.out.println("s2 = " + s2);
        System.out.println("b2 = " + b2);
    }
}
```

What is the difference between **+=** and **=+** in Java? An expression **a += 1** will result as **a = a + 1** while the expression **a =+ 1** will result as **a = +1**. The correct compound statement is **+=**, not **=+**, so do not use the later one.

2.ARITHMETIC OPERATORS IN JAVA WITH EXAMPLES

Arithmetic operators perform arithmetic operations e.g. **addition, subtraction, multiplication** and **division**. Table below shows the list of all arithmetic operators. Consider **a** and **b** as two operands or variables.

Operator	Name	Example	Description
+	Additive operator	a+b	Adds a and b
-	Subtraction operator	a-b	Subtracts b from a
*	Multiplication operator	a*b	Multiplies a and b
/	Division operator	a/b	Divides a by b
%	Remainder operator	a%b	Divides a by b and returns the remainder

The symbol " %" is also the part of arithmetic operator, it divides one operand by another operand and returns the remainder as it's result. It is also known as **modulo** operator. The code below shows how to use these operators in java.

```
int a = 10, b = 20;
int c = a + b; // value of c = 30
c = b - a; // c = 10
c = a * b; // c = 200
c = b / a; // c = 2
c = 15 % 2; // c = 1
```

Can we use spaces before and after arithmetic operators? Yes, that won't be a problem. For example expressions `5+10`; `5 + 10`; `5 + 10`; are same and will run without any error. Each operand in an arithmetic operation can be a constant value, a variable/expression, a function call returning a compatible value or a combination of all these.

```
int a = 10, b = 20;
int c = 5 + 10; // operands can be a constant value
c = a + b; // operands can be a variable/expression
c = (a - b) + c; // operands can be a variable/expression
c = a + sum(5,10); // operands can be a function call
```

The addition(+) operator in java is also used for **string concatenation**. Two strings in java can be concatenated using + operator like below:

```
String s1 = "herbert", s2 = "java";
String s3 = s1 + s2; // s3 = "herbertjava"
```

Results of arithmetic operations must be assigned to a variable, not doing so results in compilation error.

```
int a = 10, b = 20;
a + b; // Compilation error, result must be assigned in a variable eg. a = a + b;
```

The operands in an arithmetic operation can be of different data types as well, except the **boolean** data type. The result of such operation must be assigned in a compatible data type.

```
int a = 10, double b = 20.5;
double d = a + b; // d = 30.5
d = a * b; // d = 205
```

```
char c = 'a' + 5; // c = f
```

Arithmetic operators can also be combined with **assignment operator** to create a compound assignment like below:

```
int a = 10, b = 20;
a += 10; // is same as a = a + 10; // += is addition compound assignment operator
b *= 5; // is same as b = b * 5; // *= is multiplication compound assignment operator
```

The **arithmetic operations** must be performed with **primitive data type variables** only. Performing any arithmetic operations on **non primitive data type** variables will result as compilation error, except one exception which is the addition operator which can be used with the String data type only for string concatenation.

```
String s1 = "herbert", s2 = "java";
String s3 = s1 - s2; // Compilation error
s3 = s1 * s2; // Compilation error
s3 = s1 + s2; // s3 = "herbertjava"
```

How many arithmetic operators are there in java? Are **5** arithmetic operators in java, + - * / and %.

JAVA MODULO OR REMAINDER OPERATOR

The % operator in java is known as modulo or remainder operator. This operator divides one operand by another operand and returns **the remainder** as it's result. E.g. in expression **8%3**, 8 is divided by 3 and the remainder 2 is returned as result.

```
int a = 10 % 3 // a = 1
a = 10 % 2; // a = 0
a = 23 % 7; // a = 2
```

The modulo operator can be used with other primitive data types as well except the **boolean** data type.

```
double a = 10.5 % 3; // a = 1.5
double b = 10.5 % 3.5; // b = 0.0
float f = 20.5f % 3.5f; // f = 3.0
```

ARITHMETIC OPERATORS PROGRAM IN JAVA

```
class ArithmeticOperator {
    public static void main (String[] args) {
        int num1 = 20, num2 = 10, result;
        result = 5 + 10;
        System.out.println("5 + 10 = " + result);
        result = num1 + num2;
        System.out.println(num1 + " + "+num2+" = " + result);
        result = num1 + add(2,3);
        System.out.println("result with method call = " + result);

        result = num1 - num2;
        System.out.println(num1 + " - "+num2+" = " + result);

        result = num1*num2;
        System.out.println(num1 + " * "+num2+" = " + result);

        result = num1/num2;
        System.out.println(num1 + " / "+num2+" = " + result);
```

```

        result = num1 % 7;
        System.out.println(num1 + " % 7 = " + result);
        result += 2; // Equivalent to result = result+2;
        System.out.println("result = " + result);
    }
    static int add (int a, int b) {
        return a + b;
    }
}

```

Modulo operator is also useful in some cases, for example using modulo operator we can check or find, *if a number is even or odd, the last digit of a number, if a number is divisible by another number* etc. The program below shows how we can check or find these things using modulo operator.

JAVA MODULO OPERATOR PROGRAM

```

class ModuloOperator {
    public static void main (String[] args) {
        int num1 = 118, num2 = 3;
        // Check if a number is even/odd
        if(num1 % 2 == 0)
            System.out.printf("%d is an even number %n", num1);
        else
            System.out.printf("%d is an odd number %n", + num1);
        // To find the last digit of a number
        int last_digit = num1 % 10;
        System.out.printf("Last digit of number %d is %d %n", num1, last_digit);
        // Check if a num1 is divisible by num2
        if(num1 % 3 == 0)
            System.out.printf("%d is divisible by %d", num1, num2);
        else
            System.out.printf("%d is not divisible by %d", num1, num2);
    }
}

```

3 UNARY OPERATORS IN JAVA WITH EXAMPLES.

Unary simply means **one**, so unary operators are the operators which operates on only one operand. These operators are also quite used by java programmers. The table below shows the list of all unary operators in java. For this table let's assume we have an integer variable `a` and a boolean variable `isValid` :

Operator	Name	Example	Description
+	Unary plus operator	+a, +5	Indicates positive value
-	Unary minus operator	-a, -3	Negates an expression
++	Increment operator	a++, ++a	Increments value of a by 1
--	Decrement operator	a--, --a	Decrements value of a by 1
!	Logical complement operator	!isValid	Inverts the value of a boolean

JAVA UNARY PLUS AND MINUS OPERATOR

Unary plus operator indicates that a number is positive. By default, numbers are positive, the **number 2** and **+2** are same, so it doesn't matter whether you have used or not used this operator with a variable / value. The unary minus operator negates a variable/value, that means this operator makes a positive number as negative and a negative number as positive. An operand in a unary plus and minus operator can be a constant value, a variable, an expression or a function call returning a compatible value.

```
int a = +10, b = -20; // operand can be a constant value
int c = -a; // operand can be a variable
int d = -(a+b); // operand can be an expression
int e = -sum(a,b); // operand can be a method(sum) call
```

JAVA LOGICAL COMPLEMENT OPERATOR

The logical complement operator(!) inverts the boolean value, if the boolean value is **true**, it becomes **false** and if it is **false**, it becomes **true**. The operand in logical complement operator must be a boolean value, a boolean variable, a boolean expression or a method call returning a boolean value. The operand cannot be a constant value.

```
boolean b = !true; // operand can be a boolean value // b = false
boolean c = !b; // operand can be a boolean variable // c = true
boolean d = !(10 > 5); // operand can be an expression // d = false
boolean e = !isValid(); // operand can be a boolean method(isValid) call
boolean f = !10; // Incorrect, operand can't be a constant value
```

Increment and decrement operators in Java

The unary increment(++) operator increments the value of a variable by 1 while unary decrement(-- operator decrements the value of a variable by 1. For example **a++** increases the value of **a** by 1 and is same as **a = a + 1**, similarly **a--** decreases the value of **a** by 1 and is same as **a = a - 1**. Both these operators can be used with a variable only, not with any constant value, expression or method calls.

```

int a = 2, b = 3;
a++; // is same as a = a + 1;
a--; // is same as a = a - 1;
// Following is incorrect declaration
int d = 2++; // increment/decrement can't be used with constant value
d = (a+b)++; // increment/decrement can't be used with expression
d = sum(a,b)++; // increment/decrement can't be used with method

```

A unary **increment / decrement** operator can be applied **before(prefix)** or **after(postfix)** the variable name. Both versions, **++a** and **a++** will increase the value of **a** by one. The next section explains more detail about both the operators.

PRE AND POST INCREMENT AND DECREMENT OPERATORS IN JAVA

If the **unary ++ operator** is used before the variable name, it is called **pre-increment operator** while if it is used after the variable name, it is called **post increment operator**. Also, if the unary **-- operator** is used before the variable name, it is called **pre-decrement operator** while if it is used after the variable name, it is called **post decrement operator**. E.g. in code **++a**; the operator **++** is a **pre-increment operator** while in code **a++**; **operator ++** is post increment operator, the same applies with **--** operator as well. In **pre increment/decrement**, the value of the variable is incremented/decremented first then it is assigned or evaluated, that is why it is called pre-increment/decrement. While in post increment / decrement the current value of a variable is assigned or evaluated, after that the increment / decrement happens, that is why it is called post increment/decrement. Let's know this by the example below :

```

int a = 2, b = 3, c = 4, d = 5;
int p = a++; // p = 2; first assignment happens then the increment
p = a; // a = 3, p = 3; // a was increased in previous line
int q = ++b; // q = 4; first increment happens then the assignment
int r = c--; // r = 4, first assignment happens then the decrement
r = c; // c = 3, r = 3; // c was decreased in previous line
int s = --d; // s = 4; first decrement happens then the assignment

```

To understand it clearly let's see step by step how this expression is evaluated. Just remember in java an expression is evaluated from left to right.

```

a++ + --a - a-- * ++a; // original expression
2 + --a - a-- * ++a; // current value of a is 2, so a++ is replaced with
2 + 2 - a-- * ++a; // at --a, the value of a was 3(was incremented with f
2 + 2 - 2 * ++a; // current value of a is 2 so a-- is replaced with 2, th
2 + 2 - 2 * 2; // Now current value of a is 1 so ++a will be replaced as
2 + 2 - 4; // first multiplication will happen as it has higher precedence
0 // So final value will be 0 which will be assigned in p

```

The diagram below demonstrates how the value of variable `a` is evaluated.



What are prefix and postfix operators in Java? The **pre increment / decrement operator** are also known as **prefix operators** in java while the **post increment / decrement operator** is also known as **postfix operators** in java. What is the difference between **pre increment and post increment**? As mentioned above, in pre increment the value of the variable is increment first then it used or assigned in an expression while in post increment the current value of the variable is used or assigned in an expression then it is incremented.

JAVA PROGRAM OF UNARY OPERATORS

```

class UnaryOperator {
    public static void main(String[] args) {
        int result = +5;
        System.out.println("result = "+result);
        result--;
        System.out.println("result = "+result);
        result++;
        System.out.println("result = "+result);
        result = -result;
        System.out.println("result = "+result);
        int i = 5;
        i++;
        System.out.println("i = "+i); // prints 6
        ++i;
        System.out.println("i = "+i); // prints 7
        System.out.println("i = "+ ++i); // prints 8
        System.out.println("i = "+ i++); // prints 8
        System.out.println("i = "+i); // prints 9
        System.out.println("i = "+ --i); // prints 8
    }
}

```

```

        boolean isValid = false;
        System.out.println("isValid = "+ !isValid);
    }
}

```

The program below demonstrates the **pre and post increment/decrement operator** in java, you can evaluate the value of different expressions in this example by yourself first then match your result with the output to see if you have understood it properly.

JAVA PROGRAM FOR INCREMENT AND DECREMENT OPERATORS

```

class IncrementDecrement {
    public static void main(String[] args) {
        int a = 2, result;
        result = a++ + --a - a-- * ++a;
        System.out.println("result = "+result+", a = "+a);
        result = ++a * --a + a-- - a++;
        System.out.println("result = "+result+", a = "+a);
        result = --a * ++a / ++a - a++ + a--;
        System.out.println("result = "+result+", a = "+a);
    }
}

```

RELATIONAL AND CONDITIONAL OPERATORS IN JAVA

Lets see the details about **relational and conditional operators** in java. We shall cover details like what these operators are, how to use them, list of all relational/conditional operators and java programs of these operators.

RELATIONAL OPERATORS IN JAVA

Relational operators are used to relate or compare if one operand is greater than, less than, equal to, or not equal to another operand. Operators like **>**, **>=**, **<** etc are called as relational operators while operator **==** is called as equality operator. All relational operators along with equality operator returns either **true** or **false**. Table below shows the list of all relational operators in java.

List of Relational operators in Java

Operator	Name	Example	Description
<code>==</code>	Equal to Operator	<code>a==b</code>	Checks if <code>a</code> and <code>b</code> are equal
<code>!=</code>	Not equal to Operator	<code>a!=b</code>	Checks if <code>a</code> and <code>b</code> are not equal
<code>></code>	Greater than Operator	<code>a>b</code>	Checks if <code>a</code> greater than <code>b</code>
<code>>=</code>	Greater than or equal to Operator	<code>a>=b</code>	Checks if <code>a</code> greater than or equal to <code>b</code>
<code><</code>	Less than Operator	<code>a < b</code>	Checks if <code>a</code> less than <code>b</code>
<code><=</code>	Less than or equal to Operator	<code>a<=b</code>	Checks if <code>a</code> less than or equal to <code>b</code>

```
int a = 10, b = 20; boolean b2;
b2 = 5 > 6; // operand can be a constant value, b2 = false
b2 = a < b; // operand can be variables, b2 = true
b2 = a <= (b-10); // operand can be an expression, b2 = true
b2 = a >= sum(5,10); // operand can be a method(sum) call, b2 = false
```

Relational operators are mostly used with **conditional** and **looping** statements like `if`, `if else`, `for`, `while` etc.

```
int a = 10, b = 20; boolean b2;
if(a > b)
while(a > b)
```

There can be spaces before and after all relational operator. Also remember that, in multi-character relational operators like `>=`, `<=`, there should not be any space between the characters of the operator. For example `= =`, `> =`, `< =` would result in compilation error.

```
int a = 10, b = 20;
if(a>b) or if(a > b) // Correct, spaces are allowed before and after the operator
if(a>=b) or if(a >= b) // Correct use of multi-character operator
if(a > = b) // Incorrect use, spaces not allowed within multi-character operator
```

All relational operators can be used with primitive data types only except equality(`==`) operator which can be used with non primitive data types as well. The equality operators can be used to compare if two objects have same references(address).

```
String str = new String("refresh java");
String str2 = new String("Refresh Java");
String str3 = str;
if(str == str2) // condition will result as false
if(str == str3) // condition will result as true
if(str > str2) // Incorrect, operator > can't be use with non primitive ty
```

The result of a **relational operator** is always a **boolean** value that is **true or false**. So, they must be assigned to a **boolean type variable** only, assigning in any other data type variable will result in compilation error.

```
class RelationalOperator {
    public static void main(String[] args) {
        int num1 = 3;
        int num2 = 5;
        if(num1 == num2)
            System.out.println("num1 == num2");
        if(num1 != num2)
            System.out.println("num1 != num2");
        if(num1 > num2)
            System.out.println("num1 > num2");
        if(num1 >= num2)
            System.out.println("num1 >= num2");
        if(num1 < num2)
            System.out.println("num1 < num2");
        if(num1 <= num2)
            System.out.println("num1 <= num2");
    }
}
```

What is the difference between = and == operators in java? Operator = is an assignment operator which is used to assign value in a variable while == is an equality operator which is used to compare if two variables/values are equal or not.

CONDITIONAL OPERATORS IN JAVA

Following operators are conditional operators in java:

Operator	Name	Example	Description
&&	Conditional AND Operator	(a==1 && b==5)	Conditional-AND operator, returns true if both expression returns true , else false
 	Conditional OR Operator	(a==1 b==5)	Conditional-OR operator, returns true if any of the expression returns true , else false

The result of conditional **AND** and conditional **OR** operators is also a boolean value which is **true** or **false**. **What is Logical AND and Logical OR operators in java?** The conditional-AND (&&) and conditional-OR (||) operators in java is also known as **Logical AND(&&)** and **Logical OR(||)** operators respectively. Both these operators are also binary operators, since they operate on two operands / expressions. The left and right expressions of these operators must be a boolean variable/value, expression or a function returning a boolean value. The operands can not be a constant value like 5, 10 etc.

```
int a = 10, b = 20, boolean b2 = true;
if(b2 && a > 5) // operand can be a boolean variable or expression
if(a > 5 || b > 20) // operands can be an expression
if(a > 5 && isValid()) // operands can be an expression or function returning a boolean value
if(5 && 10) // Incorrect, operands can not be a constant value.
```

The block below shows what would be the result of && and || operators when it's left and right expressions/conditions returns **true** or **false**.

```
true && true => true;      true || true => true;
true && false => false;    true || false => true;
false && true => false;    false || true => true;
false && false => false;   false || false => false;
```

In conditional && and || operator the second expression is evaluated only if needed. For example if first expression in && operator returns **false**, the second expression will not be evaluated because no matter what is the result(**true** or **false**) of second expression, the && operator will return **false**. Similarly in case of || operator if first expression returns **true**, the second expression will not be evaluated.

```
int a = 10, b = 20;
if(a > 12 && b > 15) // Here b > 15 won't be checked as a > 12 returns false
if(a > 5 || b > 20) // Here b > 20 won't be checked as a > 5 returns true
```

There should not be any space between the characters of conditional AND and OR operators. For example & & or | | will result in compilation error.

```
int a = 10, b = 20;
if(a > 12 & & b > 15) // Incorrect, there should not be any space in &&
if(a > 5 | | b > 20) // Incorrect, there should not be any space in ||
```

These operators can also be repeated any number of times inside an expression and they can be mixed as well.

```
int a = 10, b = 20;
if(a > 5 && a < 20 && b > 15) // returns true, as all conditions are true
if((a > 10 && a < 20) || b > 15) // returns true, as b > 15 returns true
```

JAVA TERNARY OPERATOR

The ternary operator(`?:`) is another conditional operator in java. This operator is thought as a short form of **if-then-else statement**. It is known as **ternary operator** since it uses three operands. Line below shows how to use this operator.

variable = someCondition ? Expression1 : Expression2;

Here `someCondition` is a boolean expression that returns either `true` or `false`. You can read this operator as "if `someCondition` is `true`, assign the value of `Expression1` into the `variable` else assign the value of `Expression2` into the `variable`."

WHAT IS BOOLEAN EXPRESSION IN JAVA?

An expression which returns either `true` or `false` is known as boolean expression. For example `a > b`, `a == b`, `(a + b) < c` etc are boolean expressions.

Here Expression1 and Expression2 can be a constant value, a variable or an expression like `a+b`, `a*b+c`, `a>b`, "hello" etc. The value of these expressions must match with data type of the variable.

```
int a = 10, b = 20;
int result = a > b ? 5 : 1; // Expression can be a constant value
int result2 = a > b ? a : b; // Expression can be a variable
int result3 = a > b ? (a-b) : (b-a);
String str = (a == b) ? "Equal" : "Not Equal";
```

JAVA PROGRAM OF CONDITIONAL OPERATOR

```
class ConditionalOperator {
    public static void main(String[] args) {
        int num1 = 3, num2 = 5, num3 = 10;

        if(num1 == 3 && num2 == 5)
            System.out.println("num1 = 3 AND num2 = 5");
        if(num1 > 0 && num2 > 1 && num3 >= 10)
            System.out.println("num1 > 0 AND num2 > 1 and num3 >= 10");
        if(num1 == 3 || num2 == 5)
            System.out.println("num1 is 3 OR num2 is 5");
        if((num1 > 0 || num2 > 1) && num3 > 10)
            System.out.println("Print me if condition is true");

        int result = num1 < num2 ? num1 : num2;
        String str = num1 == num2 ? "num1 and num2 is equal" : "num1 and num2 is not equal";
        System.out.println("result = "+result);
        System.out.println("str = "+str);
    }
}
```

JAVA BITWISE AND BIT SHIFT OPERATORS WITH EXAMPLES

Bitwise and **Bit shift operators** operates on bits(0 and 1). They operates on binary representation of operands. These operators can be used with integer(`byte`, `short`, `int` and `long`) data type variables or values only and they always returns a numerical value. Let's see these operator types one by one.

BITWISE OPERATORS IN JAVA

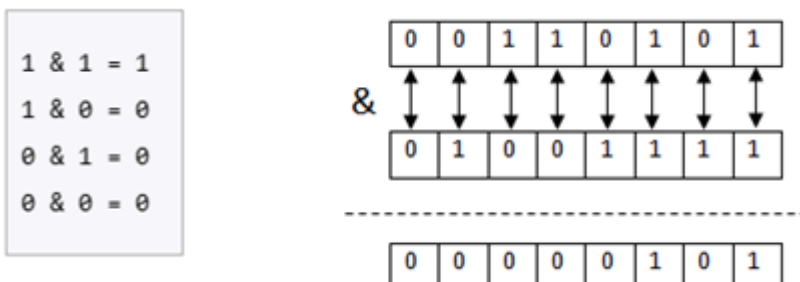
As the name itself suggests, bitwise operator performs operation bit by bit wise. Table below shows the list of all bitwise operators in java. Consider *a* and *b* as two integer type variables.

Operator	Name	Example	Description
&	Bitwise AND Operator	<i>a</i> & <i>b</i>	Performs bitwise AND operation between bits of <i>a</i> and <i>b</i>
	Bitwise OR Operator	<i>a</i> <i>b</i>	Performs bitwise OR operation between bits of <i>a</i> and <i>b</i>
^	Bitwise XOR Operator	<i>a</i> ^ <i>b</i>	Performs bitwise exclusive OR operation between bits of <i>a</i> and <i>b</i>
~	Bitwise complement Operator	~ <i>a</i>	Inverts the bits of variable <i>a</i>

All the bitwise operators except complement operator are binary operators. The bitwise complement operator is a unary operator, as it needs only one operand to perform the operation. It belongs to bitwise operator type because it works on bits.

Bitwise AND operator in Java

Bitwise AND(&) operator performs bitwise AND operation on corresponding bits of both the operands. It returns 1 if both the bit's are 1 else it returns 0. For example & operation between two byte variable with value as 53(00110101) and 79(01001111) will result in 5(0000101).

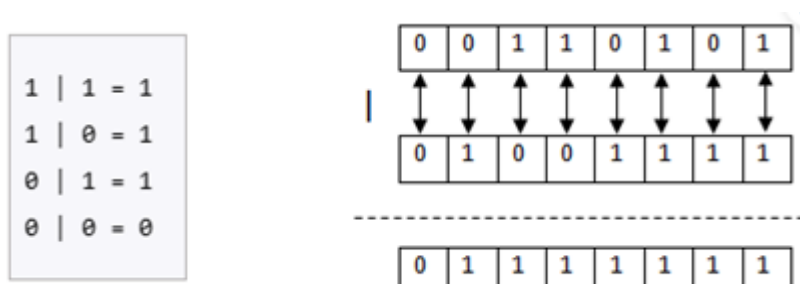


WHAT IS THE DIFFERENCE BETWEEN & AND && OPERATOR IN JAVA?

The & operator performs bitwise AND operation on bit representation of both operands and returns an integer value while logical AND(&&) operator performs operation on two boolean expressions and returns boolean value(true or false) as result.

Bitwise OR operator in Java

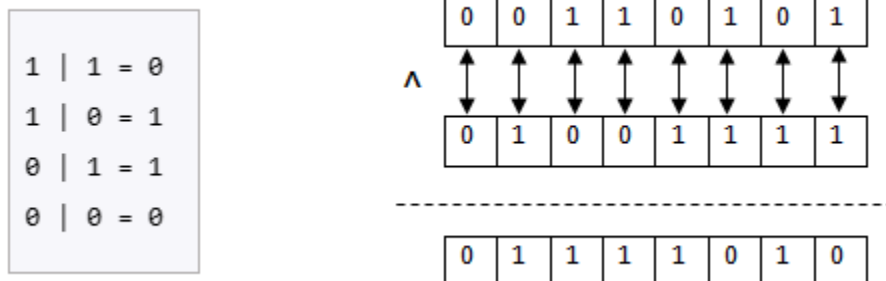
Bitwise OR(|) operator performs bitwise OR operation on corresponding bits of both the operands. It returns 1 if any of the bit's is 1 else it returns 0. For example | operation between two byte variable with value as 53(00110101) and 79(01001111) will result in 127(01111111).



What is the difference between bitwise OR(|) and logical OR(||) operator in java? The | operator performs bitwise OR operation on bit representation of both operands and returns an integer value while logical OR(||) operator performs operation on two boolean expressions and returns boolean value(true or false) as result.

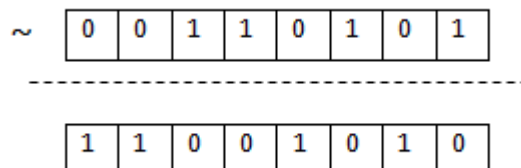
Bitwise Exclusive OR operator in Java

Bitwise XOR(^) operator performs bitwise exclusive OR operation on corresponding bits of both the operands. It returns 1 if both bit's are different else it returns 0. For example **XOR** operation between two byte variable with value as 53(00110101) and 79(01001111) will result in 122(01111010).



BITWISE COMPLEMENT OPERATOR

A **unary bitwise complement operator ~** inverts a bit pattern of an operand. It converts 0 as 1 and 1 as 0. For example a byte data type contains 8 bits, using this operator with a byte variable with value whose bit pattern is "00110101" would be changed as "11001010".



BITWISE OPERATOR PROGRAM IN JAVA

```
class BitwiseOperatorDemo {
    public static void main(String[] args) {
        byte num1 = 53;
        byte num2 = 79;
        int result;
        result = num1 & num2;
        System.out.println("num1 & num2 = " + result);
        result = num1 | num2;
        System.out.println("num1 | num2 = " + result);
        result = num1 ^ num2;
        System.out.println("num1 ^ num2 = " + result);
        result = ~num1; /* 11001010(-54, 2's complement form) = ~00110101 */
        System.out.println("~num1 = " + result );
    }
}
```

}

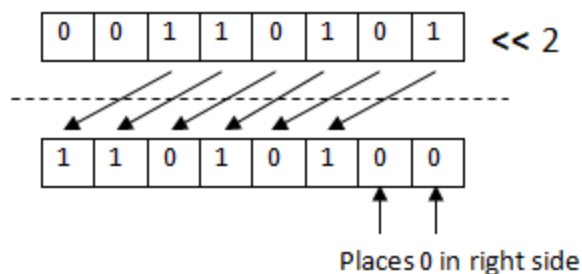
BIT SHIFT OPERATOR IN JAVA

As the name itself suggests, the bit shift operators shifts the bits of an operand to left or right depending on the shift operator used. In bit shift operator, the bit pattern(operand) is given in left-side while the number of positions to shift by is given in right side of the operator. All bit shift operators are binary operators in java as they require two operands to operate. Table below shows the list of all bit shift operators in java.

Operator	Name	Example	Description
<<	Signed left shift Operator	a << 2	Left shifts the bits of a by 2 position
>>	Signed right shift Operator	a >> 3	Right shifts the bits of a by 3 position.
>>>	Unsigned right shift Operator	a >>> 2	Right shifts the bits of a by 2 position and places 0's in left side.

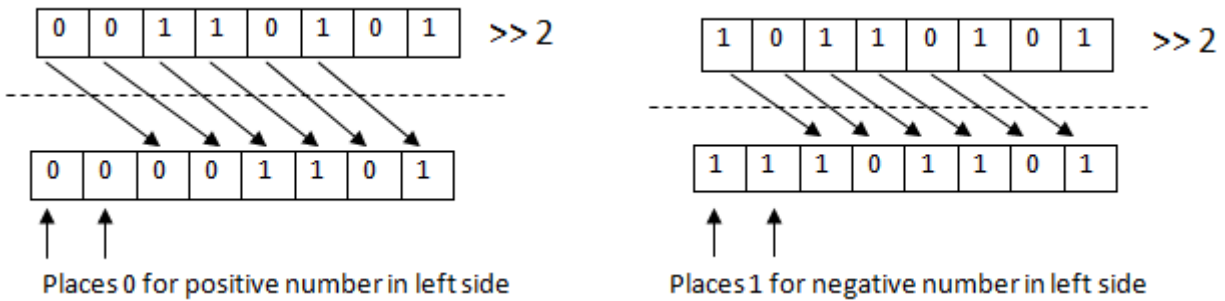
SIGNED LEFT SHIFT OPERATOR IN JAVA

The **signed left shift operator** "<<" shifts a bit pattern to the left by the number of positions given in the right side of this operator. It places **0** on the right side for every shift. Java stores number's in 2's complement notation, where the most significant bit(leftmost bit) is used for sign bit. In java, if sign bit is **0**, it means the number is positive and if it is **1**, it means the number is negative. Using this operator a negative number can become a positive number and vice-versa. If a **0** comes into sign bit after shifting the bits of a negative number, then it becomes a positive number. Similarly if **1** comes into sign bit after shifting the bits of a positive number, then it becomes a negative number. These conditions are indications of **underflow** and **overflow** respectively.



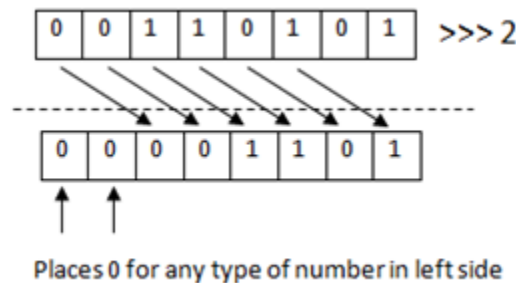
SIGNED RIGHT SHIFT OPERATOR IN JAVA

The **signed right shift operator** ">>" shifts a bit pattern to the right by the number of positions given in the right side of this operator. The signed right shift operator places **0**(for +ve number) or **1**(for -ve number) in left side for every shift. That is why using this operator with a positive number will always be positive while a negative number will always be negative.



UNSIGNED RIGHT SHIFT OPERATOR IN JAVA

The **unsigned right shift operator** ">>>" shifts a **bit pattern** to the right by the number of positions given in the right side of this operator and places a **0** into the leftmost position in every shift, no matter whether the number is negative or positive. That is why this operator always returns a positive number.



WHAT IS THE DIFFERENCE BETWEEN SIGNED AND UNSIGNED RIGHT SHIFT OPERATOR?

The only difference between **signed** and **unsigned** right shift operator is, signed right shift operator places **0** or **1** into the leftmost position depending on whether the number is positive or negative while the unsigned right shift operator always places **0**, no matter the number is positive or negative. The result of >>> operator is always a positive number while result of >> operator is positive for positive numbers and negative for negative numbers.

BIT SHIFT OPERATORS PROGRAM IN JAVA

```
class BitShiftOperatorDemo {
    public static void main(String[] args) {
        byte num1 = 53;
        byte num2 = -75;
        byte result;

        result = (byte)(num1 << 1); /* 01101010 = 00110101<<2 */
        System.out.println("num1 << 1 = " + result);

        result = (byte)(num1 << 2); /* 11010100 = 00110101<<2 */
        System.out.println("num1 << 2 = " + result);

        result = (byte)(num1 >> 2); /* 00001101 = 00110101>>2 */
        System.out.println("num1 >> 2 = " + result);

        result = (byte)(num1 >>> 2); /* 00001101 = 00110101>>>2 */
        System.out.println("num1 >>> 2 = " + result);
    }
}
```


OPERATOR PRECEDENCE IN JAVA

When we solve an expression containing multiple operators we follow some rules. For example in an expression $2+3*4$, we do the multiplication first before addition because the multiplication operator has higher precedence than addition operator. The same applies in java. Operator precedence is a rule that tells us the precedence of different operators. Operators with higher precedence are evaluated before operators with lower precedence. For example in expression $a+b*c$, the operator $*$ will be evaluated before $+$ operator, because operator $*$ has higher precedence than $+$ operator.

Table below shows the precedence of operators in **decreasing order**, the operators appearing first in table have higher precedence than operators appearing later. Operators appearing in same row of table have equal precedence. When operators of equal precedence appear in the same expression, a rule governs the evaluation order which says, all binary operators except for the assignment operator are evaluated from left to right while assignment operator is evaluated from right to left. For example in expression $2+3-4$, 2 and 3 is added first then 4 is subtracted from it. Consider a and b as two operands for the examples given in below table.

JAVA OPERATOR PRECEDENCE TABLE

OPERATORS	PRECEDENCE	EXAMPLE
Postfix	$expr++ \quad expr--$	$a++ , a--$
Unary	$++expr \quad --expr \quad +expr \quad -expr \quad \sim$ $!$	$++a , --a , !a$
Multiplicative	$* \quad / \quad \%$	$a*b , a/b , a\%b$
Additive	$+ \quad -$	$a+b , a-b$
Shift	$<< \quad >> \quad >>>$	$a<<2 , a>>1$
Relational	$< \quad > \quad <= \quad >= \quad instanceof$	$a<2 , a>1$
Equality	$== \quad !=$	$a==b , a!=b$

Bitwise AND	<code>&</code>	<code>a&b</code>
Bitwise exclusive OR	<code>^</code>	<code>a^b</code>
Bitwise inclusive OR	<code> </code>	<code>a b</code>
Logical AND	<code>&&</code>	<code>a&&b</code>
Logical OR	<code> </code>	<code>a b</code>
Ternary	<code>? :</code>	<code>a = a>2 ? a : b</code>
Assignment	<code>= += -= *= /= %= &= ^= = <<= >>= >>>=</code>	<code>a=b, a+=b, a/=b, a>>=2</code>

Which operator has highest precedence in java? The postfix operator(eg. `a++`, `a--`) has the highest precedence in java. **Which operator has the lowest precedence in java?** Assignment operator and its different forms has the lowest precedence in java.

A programmer should remember bellow rules while evaluating an expression in java.

- The operands of operators are evaluated from left to right. For example in expression `++a + b--`, the operand `++a` is evaluated first then `b--` is evaluated.
- Every operand of an operator (except the conditional operators `&&`, `||`, and `? :`) are evaluated completely before any part of the operation itself is performed. For example in expression `++a + b--`, the addition operation will take place only after `++a` and `b--` is evaluated.
- The left-hand operand of a binary operator are evaluated completely before any part of the right-hand operand is evaluated.
- Order of evaluation given by parenthesis `()` get's preference over operator precedence.
- The prefix version of `++` or `--` evaluates/uses the incremented value in an expression while postfix version of `++` or `--` evaluates the current value, then increases/decreases the operand value.
- All binary operators are evaluated from left to right except assignment operator.

JAVA PROGRAM OF OPERATOR PRECEDENCE

```

class OperatorPrecedence {
    public static void main (String[] args) {
        int result = 0;
        result = 5 + 2 * 3 - 1;
        System.out.println("5 + 2 * 3 - 1 = " +result);
        result = 5 + 4 / 2 + 6;
        System.out.println("5 + 4 / 2 + 6 = " +result);
        result = 3 + 6 / 2 * 3 - 1 + 2;
        System.out.println("3 + 6 / 2 * 3 - 1 + 2 = " +result);
        result = 6 / 2 * 3 * 2 / 3;
        System.out.println("6 / 2 * 3 * 2 / 3 = " +result);
        int x = 2;
        result = x++ + x++ * --x / x++ - --x + 3 >> 1 | 2;
        System.out.println("result = " +result);
    }
}

```

NOTE: Nested parentheses in an expression are evaluated from the innermost parentheses to the outermost parenthesis. If evaluation of the left-hand operand of a binary operator does not complete properly, no part of the right-hand operand will be evaluated. Arguments in a method are evaluated from left to right.

OVERFLOW, UNDERFLOW, WIDENING AND NARROWING IN JAVA

important concepts related with **variables and data types** like **Overflow**, **Underflow**, **Widening** and **Narrowing** in java. These are some useful concepts that a programmer must be aware of.

OVERFLOW AND UNDERFLOW IN JAVA

Every **data type** in java has a range of values that can be assigned into the variable of that data type. For example a **byte** data type variable can store values from -128(min) to 127(max), if you try to store a value which is not in this range then overflow or underflow takes place.

```

byte b1 = (byte)128; // Overflow, since assigned value is greater than max range of byte data type
byte b2 = (byte)(-129); // Underflow, since assigned value is less than min range of byte data type

```

So **overflow** and **underflow** in java is a condition where an operation returns a result that crosses the **range of data type**. If a value crosses the **maximum prescribed size** of a data type, it is called an **Overflow** and if the value crosses the minimum prescribed size, it is called as **Underflow**. Java does not throw any error or exception in any of the above cases. For example, an **int** is of 32 bit in Java, any value that crosses 32 bits gets rolled over, which means after incrementing **1** on max value of **int**(2147483647), the returned value will be -2147483648 which is minimum value of **int**. Also, after decrementing **1** from -2147483648, the result will be 2147483647 which is maximum value of **int**. For float data types(**float** and **double**), overflow will result in **infinity** while underflow results in **0.0**

OVERFLOW AND UNDERFLOW PROGRAM IN JAVA

```

class OverflowAndUnderflow {
    public static void main(String args[]) {
        int intOverflow = 2147483647 + 1;
        System.out.println("int overflowed value = " + intOverflow);

        int intUnderflow = -2147483648 - 1;
    }
}

```

```

    System.out.println("int underflowed value = " + intUnderflow);
    // Compilation error in below code, Constant values are checked at compile time for size limit
    // int overflow = 2147483648; // Crossed maximum prescribed value for int data type
    double doubleOverflow = 1e308 * 10;
    System.out.println("double overflowed value = " + doubleOverflow);
    double doubleUnderflow = 4.9e-324 / 100000;
    System.out.println("double underflowed value = " + doubleUnderflow);
}
}

```

WHAT IS THE DIFFERENCE BETWEEN OVERFLOW AND UNDERFLOW IN JAVA?

Overflow is a condition where a value assigned to a variable crosses the maximum range of variable's data type while underflow is the condition when the value that we assign to the variable crosses the minimum range of variable's data type. Refer the example above to understand it via program. **Does overflow and underflow happens with Non-Primitive data types as well?** **No**, as it doesn't contain value, it contains only reference.

WIDENING AND NARROWING IN JAVA

Widening is a process in which a smaller data type value is converted to wider/larger data type value. It is done by java itself that is why we call it implicit/automatic conversion. This is also known as **Upcasting**. Automatic conversion of a subclass reference variable to a superclass reference variable is also part of widening. Let's understand this by an example :

```

byte b1 = 20;
int i = b1;

```

Here in line 2 the **byte** variable(b1) value is converted into **int** data type value. This is basically called widening and it is done by java itself. In java, the following type of conversion comes under widening :

Widening or Upcasting. It happens automatically(Implicit Conversion)

byte → short → char → int → long → float → double

Subclass to superclass conversion is also widening. Refer inheritance for subclass/superclass

Subclass → Superclass

So if you assign a **byte** variable into **short, char, int, long, float** or **double** variable, it will be widening, similarly assigning any **short** variable to **char, int, long, float** or **double** variable will also be widening and so on. Some specific type of conversion given above may result in loss of precision, refer [this](#) link for more information.

Is int to double widening? Yes, **int** to **double** is widening, since the size of **double** data type is greater than **int** data type. **Narrowing** is just opposite of widening, it's a process in which a larger data type value is converted to smaller data type value. This is done explicitly by the programmer. This is also known as **Downcasting**. Explicit conversion of a superclass reference variable to a subclass reference variable is also part of narrowing. Let's understand this by an example:

```

int i = 20;
byte b1 = (byte)i;

```

Here in line 2 the `int` variable(`i`) value is converted into `byte` data type value. This is basically called narrowing and it is done explicitly as you can see I have used `(byte)` before variable `i` to convert the `int` value into `byte` value. In java, following type of conversion basically comes under narrowing :

Narrowing or Downcasting. It is done Explicitly by the programmer

`double` → `float` → `long` → `int` → `char` → `short` → `byte`

Superclass to subclass conversion is also narrowing.

Superclass → **Subclass**

So if you assign a `double` variable into `float`, `long`, `int`, `char`, `short` or `byte` variable, it will be narrowing, similarly assigning any `float` variable to `long`, `int`, `char`, `short` or `byte` variable will also be narrowing and so on. Narrowing conversion may lose information about the overall magnitude of a numeric value and may also lose precision and range.

WIDENING AND NARROWING PROGRAM IN JAVA

```
class WideningAndNarrowing {
    public static void main(String [] args) {
        byte b=20;
        int i=b; //byte to int widening
        long l=b; //byte to long widening
        double d=b; //byte to double widening
        System.out.println("int value after widening : "+ i);
        System.out.println("long value after widening : "+ l);
        System.out.println("double value after widening : "+ d);

        double d2 =20.5;
        byte b2 = (byte)d2; //Narrowing double to byte
        // long l2=d2; //compile time error, must be explicitly casted
        long l2= (long)d2; //Narrowing double to long
        float f2= (float)d2; //Narrowing double to float
        System.out.println("Narrowing double value to byte : "+ b2);
        System.out.println("Narrowing double value to long : "+ l2);
        System.out.println("Narrowing double value to float : "+ f2);
    }
}
```

What is the difference between widening and narrowing in java? In **widening**, the smaller data type value is automatically converted/accommodated into larger data type. It happens automatically, that means it is done by java itself. **Narrowing** is just opposite of widening, in this the larger data type value is converted/accommodated in smaller data type. It needs to be done explicitly by the programmer itself. **What is type casting in java?** Widening and narrowing is also known as type casting in java.

NOTE: Every time you run Overflow or underflow program, it will return same result. Every programming language has it's own way of handling overflow and underflow conditions. Overflow and Underflow deals on min/max values of data types while Widening and Narrowing deals on conversion of data type.

JAVA SIMPLE PROGRAMS

Let us write some basic programs like sum, average, circle and rectangle area etc in java. We shall covers some key concepts about **variables** and **data types**, let's see the basic programs first followed by the key concepts. The program below shows how to add two number and also shows how to find average of two number. Similarly, you can develop your own program to add multiple numbers or find the average of multiple numbers.

Addition program in java | Average program in java

```
class AdditionAndAverage {
    public static void main(String [] args) {
        int firstNum = 1234;
        int secondNum = 5678;
        int sum = firstNum + secondNum; // Adding two number
        int avg = (firstNum + secondNum)/2; // Calculating average of two
        System.out.println("sum = "+sum);
        System.out.println("Average = "+avg);
    }
}
```

The program below shows how to multiply and divide two numbers in java. Be careful to use the correct data type for storing the result of these operations. For example if division of two number results as a decimal(eg. 20.25, 245.3 etc) type value, then store it in any of the float(float, double) data types.

Multiplication program in java | Java division program

```
class MultiplicationAndDivision {
    public static void main(String [] args) {
        int firstNum = 100;
        int secondNum = 5;
        int result = firstNum * secondNum; // Multiplying two number
        int result2 = firstNum / secondNum; // Dividing two number
        double result3 = (double)200 / 6; // Division which returns decim
        System.out.println("multiplication = "+result);
        System.out.println("Division = "+result2);
        System.out.println("result3 = "+result3);
    }
}
```

Program below shows how to find area of a circle for given radius and area of rectangle for given length and width.

Area of circle java program | Area of rectangle java program

```

class CalculateArea {
    public static void main(String [] args) {
        float radius = 10.5f;
        float pi = 22/7f;
        int length = 10;
        int width = 20;
        float circleArea = pi*radius*radius; // Calculating area of circle
        int rectArea = length*width; // Calculating area of rectangle
        System.out.println("area of circle = "+circleArea);
        System.out.println("area of rectangle = "+rectArea);
    }
}

```

SOME KEY POINTS ABOUT VARIABLES & DATA TYPES

Following are some of the key points that we should be aware of :

- By default, operations on integer variables (eg 5+15, 2+3*5) returns an **int** type value, if result needs to be assigned into any other integer data type(eg. **byte** or **short**) variable, it must be casted explicitly.
- The value in a **char** variable must be a single character, assigning multiple character will result in compilation error.
- A **boolean** variable value must be given as **true** or **false**. it should not be like 'true', "true", "false" et.

WHAT DO \n AND \t DO IN JAVA PROGRAM?

The **\n** and **\t** are part of escape sequence characters in java. In java **\n** is used to insert a newline in the text at this point which means anything that you write after **\n** in a string that will be printed in a new line. While **\t** is used to insert a tab in the text at this point which means anything written after **\t** in a string will be printed after a tab space. Refer the program below. Along with **\n** and **\t** there are several more escape sequence characters in java.

Java program of \n and \t escape character

```

class VariableAndDataType {
    public static void main(String [] args) {
        byte a = 20;
        byte b = 10;
        byte sum = (byte)(a+b); // casting explicitly
        // char ch = 'abc'; // Can not assign multiple characters
        char ch = 'a';
        // boolean b2 = 'false' , c ="false"; // Can not use '', or ""
        boolean b2 = false;
        System.out.println("sum = " +sum);
        System.out.println("ch = " +ch);
        System.out.println("b2 = " + b2);
        System.out.println("First line \nSecond line");
        System.out.println("refresh \t java");
        System.out.println("Displaying \" in string.");
    }
}

```

CHPT 3:JAVA SELECTION OR CONTROL FLOW STATEMENTS**●IF and IF-ELSE**

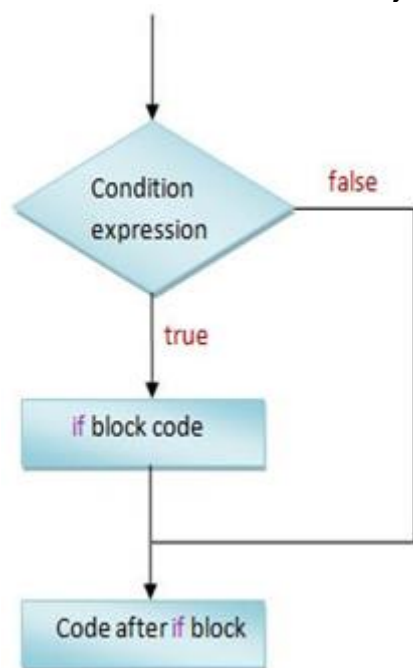
Sometime as a you may want to execute **one or more lines of code** on basis of certain condition. For an example, you may want that if a number **num1** is greater than **0**, print a message in console as *"num1 is a positive number"* by executing code **System.out.println("num1 is a positive number");** To do such scenarios, java provides a keyword known as **if keyword** which allows programmers to execute a set of statements on certain condition. The **if** statement asks the computer to execute the instructions given inside it's block only when the condition is **true**. The **syntax** of **if** keyword is:

```
if(condition) {
    // java code to run inside if block
}
```

Here, a condition is a **boolean expression** which returns either **true** or **false**. If it returns **true**, the code written inside **if** block gets executed and if it returns **false**, **if** block code will not be executed. The boolean expressions could look like **a>0**, **(a+b)>10**, **(a-b)>0** etc. These such expressions returns either **true** or **false**.

WHAT IS AN IF BLOCK IN JAVA?

Everything that comes inside matching **curly brackets { }** after **if** keyword is the part of **if** statement which is also called as **if** block in java.



Flowchart of if statement in Java

```
class IfStatement {
    public static void main(String [] args) {
        int num1 = 20;
        if(num1 > 0) {
            System.out.println("num1 is a positive number");
            System.out.println("Value of num1 = "+num1);
        }
        if(num1 < 0) {
            System.out.println("num1 is a negative number");
            System.out.println("Value of num1 = "+num1);
        }
        System.out.println("statement after if block");
    }
}
```

If statement program in Java

Can I write many conditions in an if statement in java? Yes, you can use many conditions in **if statement** by using **logical AND(&&)** and **logical OR(||)** operator. For example **if(a > 10 && a < 20)** is a valid boolean condition. **Can we use if statement without curly braces?** Yes, we can if we have only one line of code for the **if statement**. if we don't use the **curly braces** after **if statement**, the **if condition** will be applied only on the first line of code after **if statement**. Refer the program given below. **What happens if we put semicolon after if statement in Java?** Though it compiles well,

the **if statement** will end up there itself. It means the **if condition** won't be applied on the statements of **if block**, so these statements will be executed always. Refer the program given below:

```
class IfStatementUsage {
    public static void main(String [] args) {
        int a = 15, b = 10;
        if(a > 0) // no curly braces
            System.out.println("a > 0"); // if condition is applied on this line only
            System.out.println("a = "+a);

        if(a > 10 && a < 20)
            System.out.println("a > 10 and < 20");
        if(a > 0 || b > 0)
            System.out.println("a > 0 or b > 0");
        if(a > 0 && a < 20 && a > b)
            System.out.println("a > 0 and < 20 and a > b");
        if(a < 0); { // semicolon ends the if statement
            System.out.println("Printed even the if condition is false");
        }
    }
}
```

● IF-ELSE (IF-THEN-ELSE)

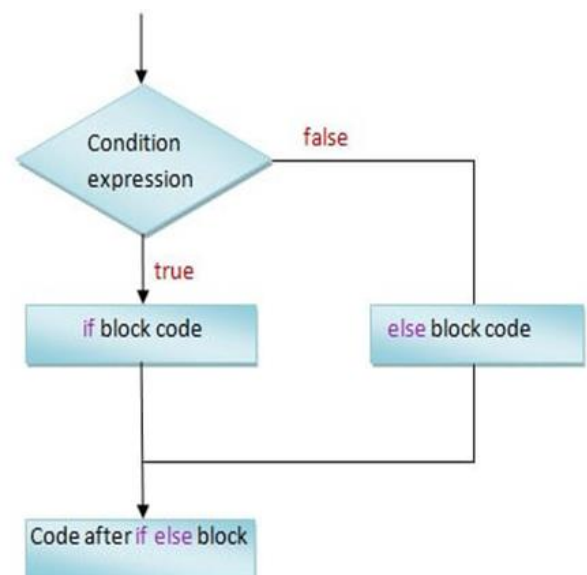
You may want to execute a set of statements when a condition is **true** and if that is not **true** then you want to execute some other set of statements. To do this, java offers the **if then else statement**. The **else block** gets executed only when the condition in **if statement** is **false**. You can also read this as "if something is **true** execute this **else** execute that". The **syntax** of **if-then-else** statement is:

```
if(condition) {
    // java code to run inside if block
}
else {
    // java code to run inside else block
}
```

IF ELSE FLOWCHART IN JAVA

IF ELSE PROGRAM

```
class IfElseStatement {
    public static void main(String [] args) {
        int num1 = 20, num2 = 15;
        if(num1 == num2) {
            System.out.println("num1 is equal to num2");
        }
        else {
            System.out.println("num1 is not equal to num2");
        }
        System.out.println("code after if else block");
    }
}
```



```

    }
}

```

JAVA IF...ELSE IF CONDITION AND NESTED IF ELSE

You may want to use **set of conditions** where your **first condition** is **false** then only move to check the next condition, and if that is also **false** then only move to check the next condition and so on. To do such scenarios, java offers an **if** then **else if statement**. You can specify another state with the **else if** statement which is run only when it's previous **if** or **else if** condition is **false**. If the **else if** condition returns **true**, then the **else if** block is executed, if it returns **false**, computer will move to next **else if** statement and evaluates the condition expression. This process goes on until any of the expression returns **true** or all the **else if** statements are finished. Once any of the expression returns **true**, the code inside that conditional statement is executed and all other condition statements after that are not executed. If all conditional statements returns **false**, then the final **else** block(if any) will be executed. The **syntax** of **if-then-else if** statement is:

```

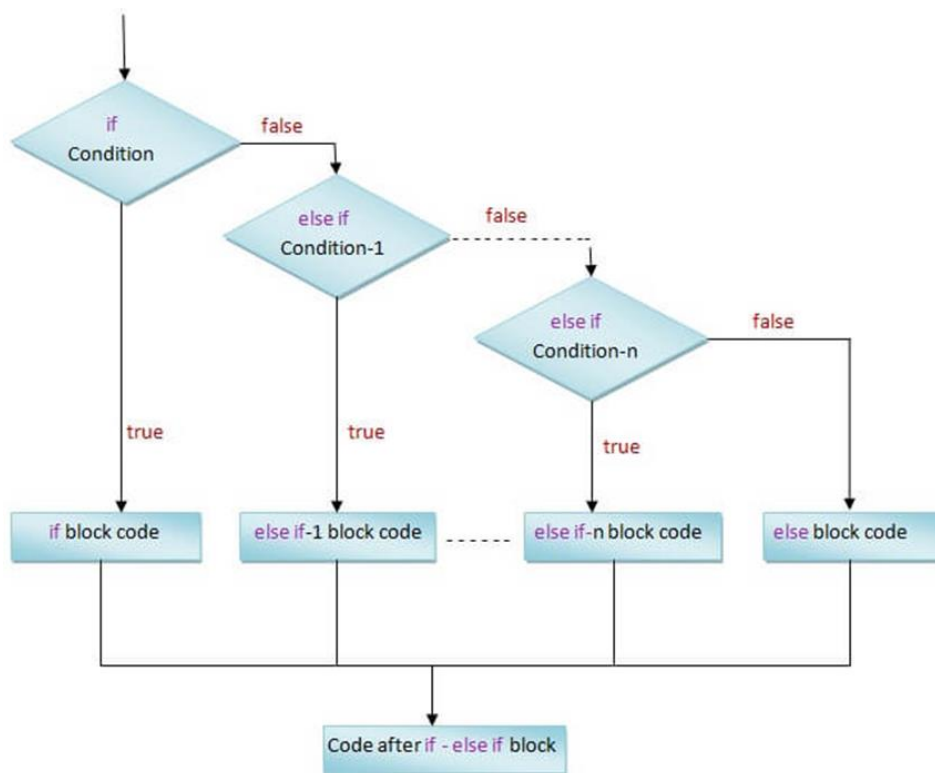
if(condition) {
    // code to be executed inside if block
}
else if(condition-1) {
    // code to be executed inside this else if block
}
else if(condition-2) {
    // code to be executed inside this else if block
}
.
.
.
else if(condition-N) {
    // code to be executed inside this else if block
}
else {
    // code to be executed inside else block, it's optional
}

```

The last **else** statement in **if else-if** ladder is optional, it's programmer's choice whether to use or not.

What is if-else if ladder statement in Java? The **if.. else if** statement given above is also called as **else if ladder** which is mostly used when programmer wants to select one block of code among many. In **if.. else if** ladder only one block(the one whose condition is **true**) of code is executed.

FLOWCHART OF IF...ELSE IF STATEMENT



IF .. ELSE IF PROGRAM IN JAVA

```

public class IfElseIfStatement {
    public static void main(String [ ] args) {
        int num1 = 20, num2 = 20;
        if(num1 > num2) {
            System.out.println("num1 is greater than num2");
        }
        else if(num1 == num2) {
            System.out.println("num1 is equal to num2");
        }
        else {
            System.out.println("num1 is less than num2");
        }
        System.out.println("code to be executed after else if ladder");
    }
}
  
```

Can we use an if inside an else if in Java? Yes, we can use if statement inside else if statement, the program will run successfully.

● NESTED IF-ELSE

Nested if-else statement in Java

Java allows you to place or put an **if else block** inside another **if or else block**. This is called a **nested if else statement**. You can do any nesting level in a program which means you can put if else block inside another if or else block up to any number of times. Inner if block is executed only when outer if condition is true.

```

if(condition-1) {
    // code to be executed inside first if block
    if(condition-2) {
        // code to be executed inside second if block
        if(condition-3) {
            // code to be executed inside third if block
            .....
        }
    }
    else {
        // code to be executed inside else block
    }
    // code to be executed inside second if block
}
// code to be executed inside first if block
}
else {
    if(condition-4) {
        // code to be executed inside this if block
    }
}
}

```

```

public class NestedIfElseStatement {
    public static void main(String [] args) {
        int num1 = 20;
        if(num1 >= 10) {
            if(num1 < 100) {
                System.out.println("num1 is a double digit
number");
            }
            else {
                System.out.println("num1 is not a double digit
number");
            }
        }
        else {
            System.out.println("num1 is less than 10");
        }
        System.out.println("code after nested if else block");
    }
}

```

NESTED IF ELSE PROGRAM IN JAVA

●FOR LOOP IN JAVA

for loop in Java

Usually, each line of code in a program is executed only once. Sometime we meet a situation where in we want to repeat the execution of one or more line of code more than once. To do such requirements java, offers **looping statements**. **Looping statements** ask a PC to repeat an execution of instructions written inside these statements as far as a given condition is **true**. Below are java looping statements to repeat the execution of statements in a program.

1. **for** loop.

2. **while** loop.

3. **do while** loop

We have to print digits from **1** to **10**, one tactic could be, we can write a program which includes 10 print statements, but what about if you need to print digits from 1 to 100 or more?. Java looping statements can help you to accomplish the same in an easy way. Java for loop can be used to repeat execution of statements for a fixed number of times or till a condition is true. The basic syntax of for loop is:

```

for(initialization; condition; increment/decrement)
{
    // code to be executed inside for loop
}

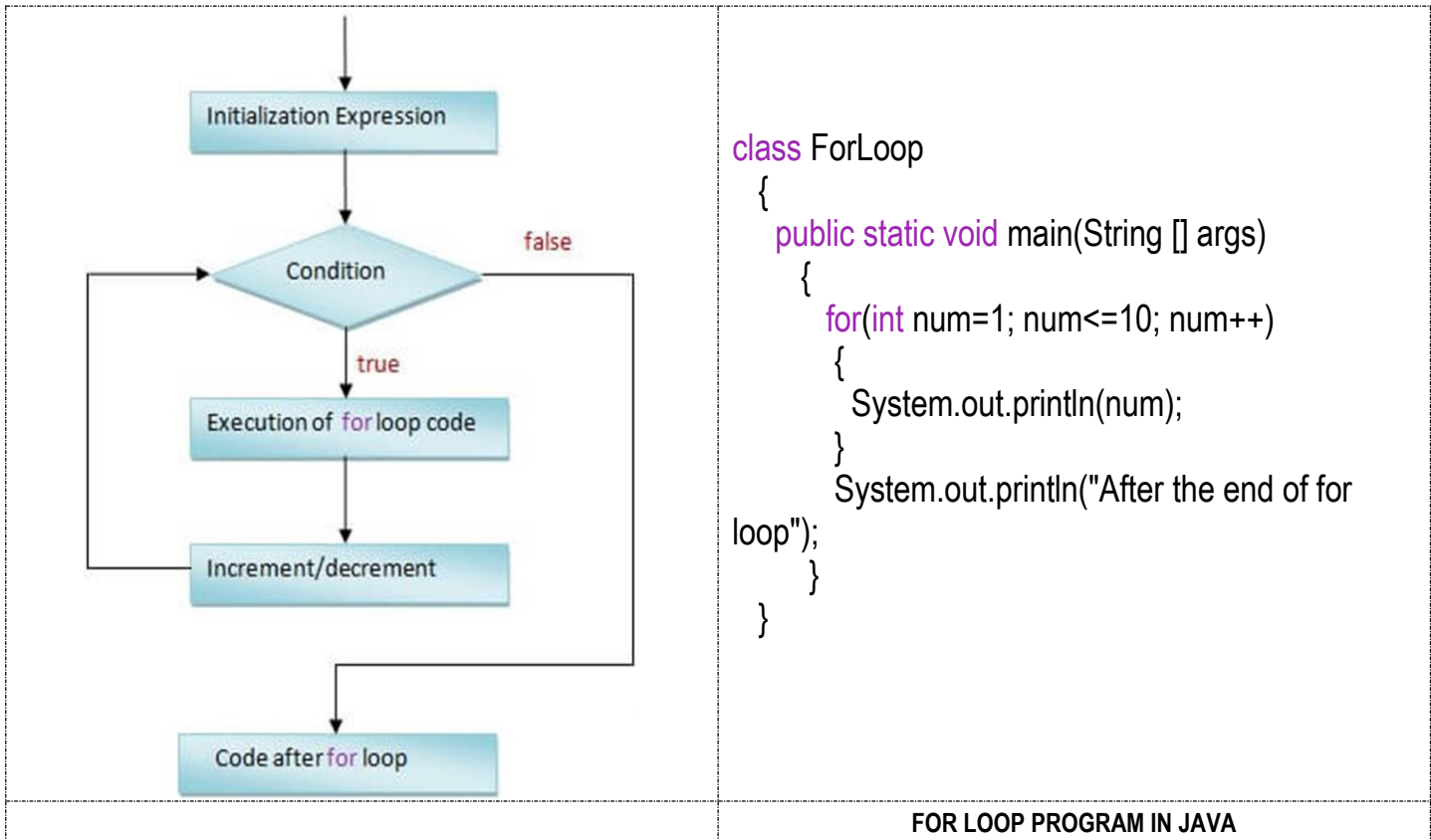
```

All that is inside the () after the **for** loop are non-compulsory expressions which do specific tasks. **Initialization** expression is used to initialize variables, objects **int i = 0**, **int count=10** etc. **Condition** expression is used to test the condition, for example **i<=10**, **count>0** etc. **Increment / decrement** expression is used to change in variable or object value, for example **i++**, **count--** etc. All that comes inside matching **curly brackets { }** after **for** keyword is the part of **for** statement which is also called as **for** block or **for** loop body.

HOW A FOR LOOP WORKS

Initialization expression is the first expression executed as the loop begins and it is executed only once. Then the **condition expression** is then evaluated, it's a **boolean expression** which returns either **true** or **false**. If the condition expression returns as **true**, Statements inside the body of **for** loop is executed. Then the increment/decrement expression is executed. Again, the condition expression is evaluated. If it returns as **true**, again the statements inside the body of **for** loop is executed and then increment/decrement expression is executed. This process goes on until the condition expression returns as **false**. If the condition expression returns as **false**, **for** loop terminates.

CONTROL FLOW DIAGRAM OF FOR LOOP



Can I use many variables in a for loop expressions? Yes you can use them in a **for** loop expressions. For eg. **for (int x=0, y=10; x<10 && y>0; x++, y--)** is a valid java **for** loop. You can initialize many variables of same data type only in initialization expression.

Infinite for loop in Java

What if a condition expression written in **for** loop statement always returns as **true**, the body of **for** loop will keep executing, this is basically called as infinite **for** loop.

```

class InfiniteForLoop
{
    public static void main(String [] args)
    {
        for(int num=1; num>0; num++)
        {
            System.out.println(num);
        }
    }
}
  
```

}

In above program, **num>0** will always return **true** since **num** increases in every iteration. So, **for** loop will keep running **infinite/endless** number of times. In **for** loop syntax, all the expressions are optional, programmer can leave all of them as blank. If all of them leaved as blank, this again becomes infinite **for** loop.

```
for(;;)
{
    // code to be executed inside for loop
}
```

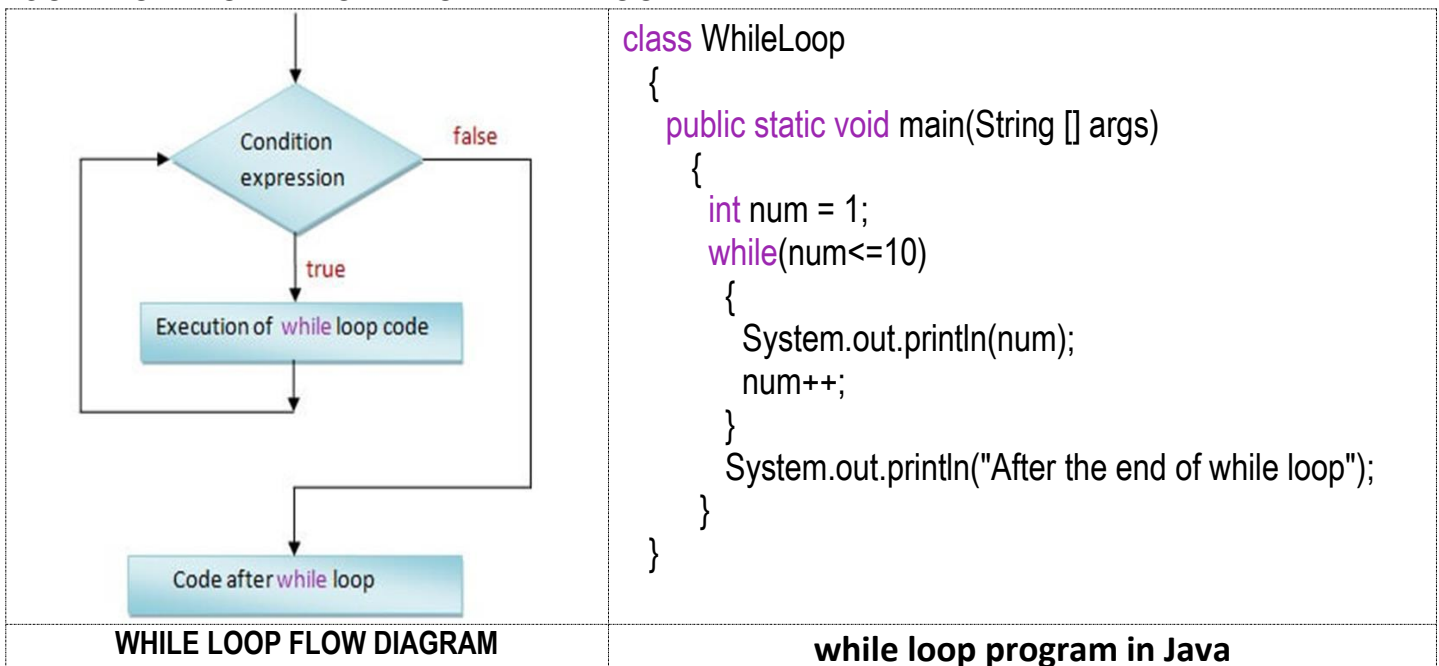
while loop in Java

while is another **looping statement** in java which allows us to repeat the execution of one or more line of code as far as a condition is **true**. The computer keeps repeating the execution of statements inside the **while** loop until the condition get's **false**. Programmer can also read this as "execute given statements **while** something is **true**". The **syntax** of **while** loop is :

```
while(condition)
{
    // code to be executed inside while loop
}
```

Here the condition is a **boolean expression** which must return either **true** or **false**. If it returns **true**, the code written inside the **while block** gets executed and if it returns **false**, the **while** block code won't be executed. Everything that comes inside **matching curly brackets { }** after **while** keyword is the part of **while** statement which is also called as **while** block or **while** loop body.

CONTROL FLOW DIAGRAM OF WHILE LOOP



Can I test multiple conditions in while statement? Yes, you can test multiple conditions using **&&** or **||** operator. For e.g. **while(a>0 && b<10)** is a valid statement.

Infinite while loop

If the condition in **while** loop always returns as **true**, it becomes an **infinite loop**.

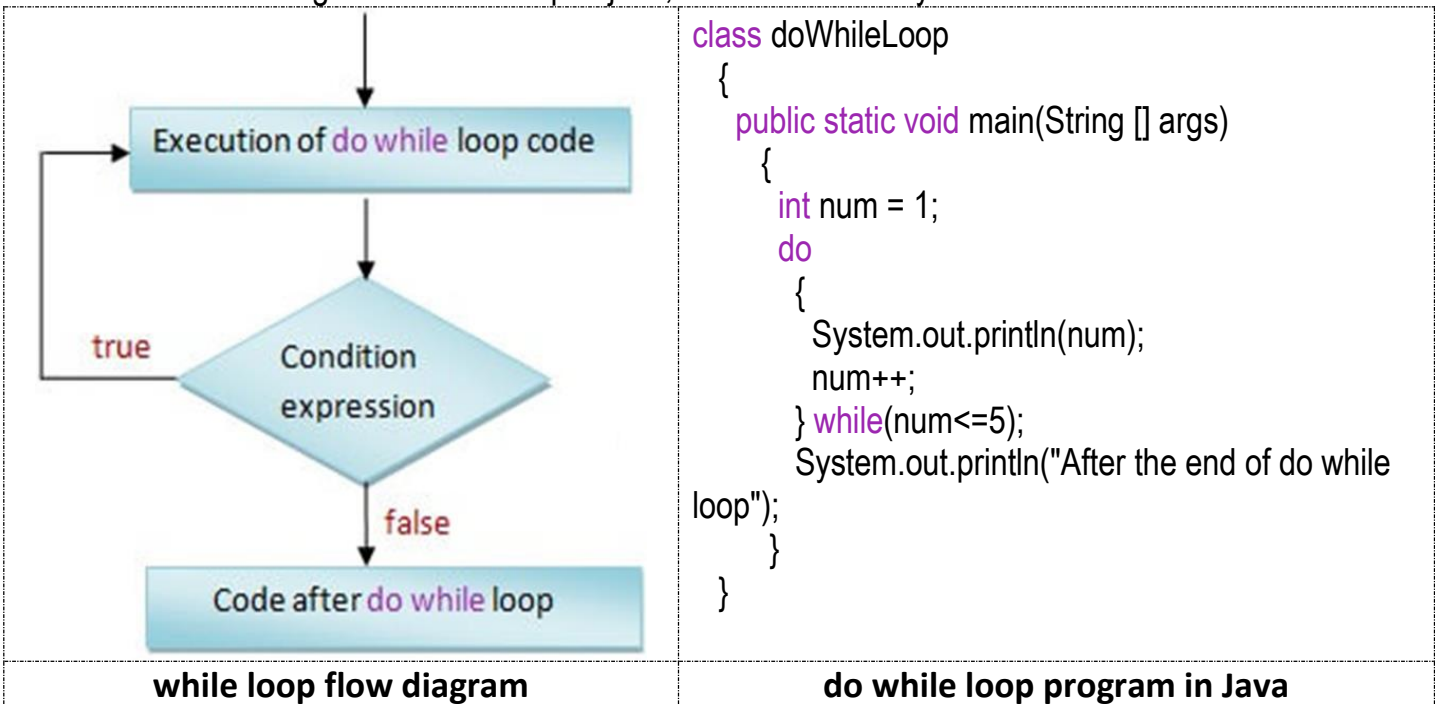
```
while(true)
{
    // code to be executed inside while loop
}
```

do while loop in Java

do while is also a looping statement in java which allows to repeat the execution of one or more line of code as far as a condition is **true**. It's so similar to the **while loop**, the only difference is that in **do while loop** the statements inside **do while** block is executed first then condition expression is tested. Therefore, code within the **do** block is always executed **at least once**. The **syntax** of **do while** loop is:

```
do
{
    // code to be executed inside do while loop
}while(condition);
```

Note the **semicolon(;) at the end of while statement**, it's the part of declaration, programmer usually misses this while using the **do while** loop in java, which results in syntax error.



DIFFERENCE BETWEEN WHILE AND DO WHILE LOOP IN JAVA

In a **while loop**, the condition expression is checked first. If it's **true** then a code inside the while loop gets executed else it won't be run. In a **do while loop**, a code inside a loop is executed first then the condition is checked. If the condition is **true** then the code inside a loop will be executed again otherwise it won't be executed. So, in a **do while** loop the code inside the loop is executed at least once no matter whether the condition is **true** or **false** which is not the case with **while** loop.

WHEN TO USE FOR, WHILE AND DO WHILE LOOP?

When you know that your code inside a loop should be executed a fixed number of times, use **for loop**. When you are not sure about the **repetition number**, go for **while loop**, in a situation where you want your code should be executed at least once, go for **do while loop**. Just for an example if you

want to print a table of a number, you should know that the iteration / repetition must be done 10 times, use **for loop** and let us suppose you want to know how many times a random integer number is divisible by 2, go for **while** loop e.g. 256 is divisible 8 times, 12 is divisible 2 times only etc.

java nested loops

Java programming allows programmers to place **one loop statement** inside the body of another loop statement. E.g., a **for loop** can be placed inside **other for loop**, a **while** loop can be placed inside other **while** loop, a **while** loop can be placed inside other **for** loop etc. The process of placing one loop inside other loop is called **nesting**, and such loops are called as nested loops. The **syntax** of nested loops are:

<pre> for(init; condition; inc/dec) { // outer loop code for(init; condition; inc/dec) { // inner loop code } // outer loop code } </pre>	<pre> while(condition1) { // outer loop code while(condition2) { // inner loop code } // outer loop code } </pre>
---	---

Points about nested loops which will help you to understand more about nesting of loops.

- For every iteration of outer loop, inner loops complete all its iteration.
- A single loop can contain multiple loops at same level inside it.
- It's perfectly legal to place, **for** inside **while**, **while** inside **for**, **for** and **while** in the same level etc.

How many times I can do nesting of loops? The process of repeating the nesting can be done any number of times.

```

class NestedForLoop
{
    public static void main(String [] args)
    {
        for(int i=1;i<=3;i++)
        {
            System.out.println("Outer loop iteration i = "+i);
            for(int j=1;j<=3;j++)
            {
                System.out.println("j = "+j);
            }
        }
    }
}

```

```

class NestedWhileLoop
{
    public static void main(String [] args)
    {
        int i=0,j=0;
        while(i<3)
        {
            j=0;
            while(j<3)
            {
                System.out.print(i+" "+j+" ");
                j++;
            }
            System.out.println("");
            i++;
        }
    }
}

```

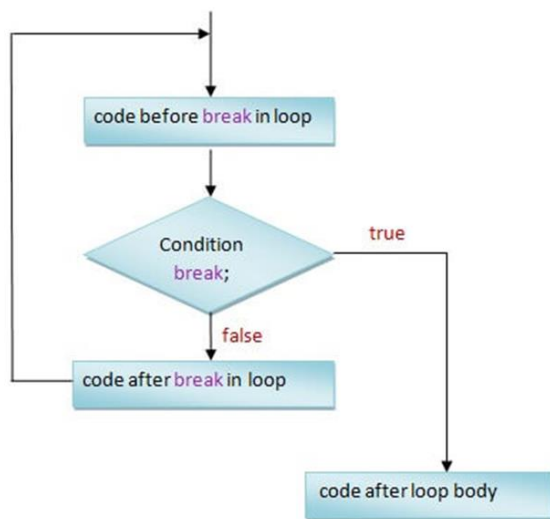
Nested for loop in Java

Nested while loop in Java

JAVA BREAK AND LABELED BREAK STATEMENT

Once the execution of a **loop statement** in a program starts, it keeps repeating the execution of loop instructions as far as a given condition is **true**. Sometimes you may want to **break** the execution flow in between. To **break** the execution flow of a loop, java provides a keyword called as **break**. It can be used inside **for**, **while**, **do while** and **switch** statements only. As soon as a **break** statement encountered in a loop, the execution of that loop gets terminated and the control flow jumps to the next statements after the loop body. It's mostly used along with conditional statements (**if**, **else**). The **syntax** of **break** statement is:

break;



break statement flow diagram

```

class BreakStatement
{
    public static void main(String [] args)
    {
        for(int i=1; i<=5; i++)
        {
            if(i==4)
            {
                break;
            }
            System.out.println("i = "+i);
        }
        System.out.println("line after for loop");
    }
}
  
```

break statement java program

When the value of **i** equals to **4**, a **break** statement is run and control flow jumps to line **(System.out.println("line after for loop");)** after the **for** loop, that is why value **4** and **5** is not printed.

What if I use break statement outside of loop or switch statement? Your program won't compile, it will throw compilation error. **Does break break Out all loops? No**, It breaks only the current loop in which it is used. If **break** statement is used inside any inner loop, it breaks only inner loop once it is executed and the execution flow moves to the next line after inner loop. The outer loop will be executed as usual.

```

class InnerLoopBreakStatement
{
    public static void main(String [] args)
    {
        for(int i=1; i<=3; i++)
        {
            System.out.println("Outer loop i = "+i);
            for(int j=1; j<=5; j++)
            {
                if(j==3)
                {
                    break;
                }
            }
        }
    }
}
  
```

```

    public static void main(String [] args)
    {
        firstLabel:
        for(int i=1; i<=3; i++)
        {
            System.out.println("Outer loop i = "+i);
            secondLabel:
            for(int j=1; j<=5; j++)
            {
                if(i==2 && j==2)
                {
                    break firstLabel;
                }
            }
        }
    }
}
  
```

<pre> break; } System.out.println("j = "+j); } } System.out.println("line after outer for loop"); } </pre>	<pre> break firstLable; } System.out.println(" j = "+j); } } System.out.println("line after labeled break statement"); } </pre>
Inner break loop	labeled break statement

LABELLED BREAK STATEMENT

A **label** is an identifier, from java SE 5 and above where we can declare a collection of statements with a label identifier. Labeled **break statement** allow you to break the execution of label statements.

JAVA CONTINUE AND RETURN KEYWORDS

Sometime you may want a specific repetition of a loop, not to run few or all instructions of loop body. To solve such needs, java provides **continue keyword** which allows you to skip the current iteration of a loop and move to the next iteration. If you want to skip the execution of a code inside a loop for a specific iteration or repetition, **you use the continue statement**. Only the code written after the **continue statement will be skipped**, code before the **continue** statement will execute as usual for that iteration. The **continue** statement can only be used inside **for**, **while** and **do while** loops of java. In case of **break** statement, the control flow exits from the loop, but in case of **continue** statement, the control flow will not be exited, only the current iteration of loop will be skipped. It is mostly used along with conditional statements(**if**, **else**). The **syntax** of **continue** statement is:

<p style="text-align: center;">continue;</p> <pre> graph TD Start(()) --> Before[code before continue in loop] Before --> Cond{Condition
continue;} Cond -- true --> Start Cond -- false --> After[code after continue in loop] After --> Start </pre>	<pre> class ContinueStatement { public static void main(String [] args) { for(int i=1; i<=5; i++) { if(i==4) { continue; } System.out.println("i = "+i); } System.out.println("line after for loop"); } } </pre>
Continue statement flow diagram	Continue statement Java Program

CONTROL FLOW DIAGRAM OF CONTINUE STATEMENT (ABOVE)

Once the **continue statement** is run inside **for** loop, the control flow moves back to **for** loop to evaluate increment/decrement expression and then condition expression. If **continue** statement is executed

inside **while** and **do while** loop, the control flow jumps to condition expression for evaluation. As you can see in above program, for **i==4** the **continue** statement is executed which skips the current iteration and move to the next iteration, that is why value **i = 4** is not printed. If **continue** statement is used inside any inner loop, it skips current iteration of inner loop only, the outer loop will be executed as usual.

<pre> class InnerLoppContinueStatement { public static void main(String [] args) { for(int i=1; i<=3; i++) { System.out.println("Outer loop i = "+i); for(int j=1; j<=2; j++) { if(i==2) { continue; } System.out.println(" j = "+j); } } System.out.println("line after outer for loop"); } } </pre>	<pre> class LabeledContinueStatement { public static void main(String [] args) { firstLable: for(int i=1; i<=3; i++) { System.out.println("Outer loop i = "+i); secondLabel: for(int j=1; j<=3; j++) { if(i==2) { continue firstLable; } System.out.println(" j = "+j); } } System.out.println("line after end of firstLable "); } } </pre>
Inner loop continue statement	labeled continue statement java program

LABELLED CONTINUE STATEMENT

A Labeled continue statement allows programmer to skip the current iteration of that label.

Java Program of labeled continue statement

JAVA RETURN STATEMENT

As the **return** keyword itself suggest that it is used to return something. Java **return** statement is used to return some value from a method. Once a **return** statement is run, the control flow is exited from current method and returns to the calling method. The **syntax** of **return** statement is:

```

return expression;
return;

```

Who gets the value that is returned by method? The caller of a method. This value can be assigned in some variable. The **return** statement given above has two forms:

- First that returns a value. To return a value, just place the value or expression after **return** keyword. for eg. **return 10**, **return a+b**, **return "refresh java"** etc. The data type of the returned value must match the return type of it's method declaration.
- Second that doesn't returns a value. To use, just place **return** keyword. When the return type of method is declared as **void**, you can use this form of **return** as it doesn't return a value, or you can also leave out to use **return** keyword for such methods.

JAVA PROGRAM OF RETURN STATEMENT

```

class ReturnStatement

```



```

{
    public static void main(String [ ] args)
    {
        int sum = sum(10,20);
        System.out.println("Sum = "+sum);
    }
    public static int sum(int a, int b)
    {
        return a+b;
    }
}

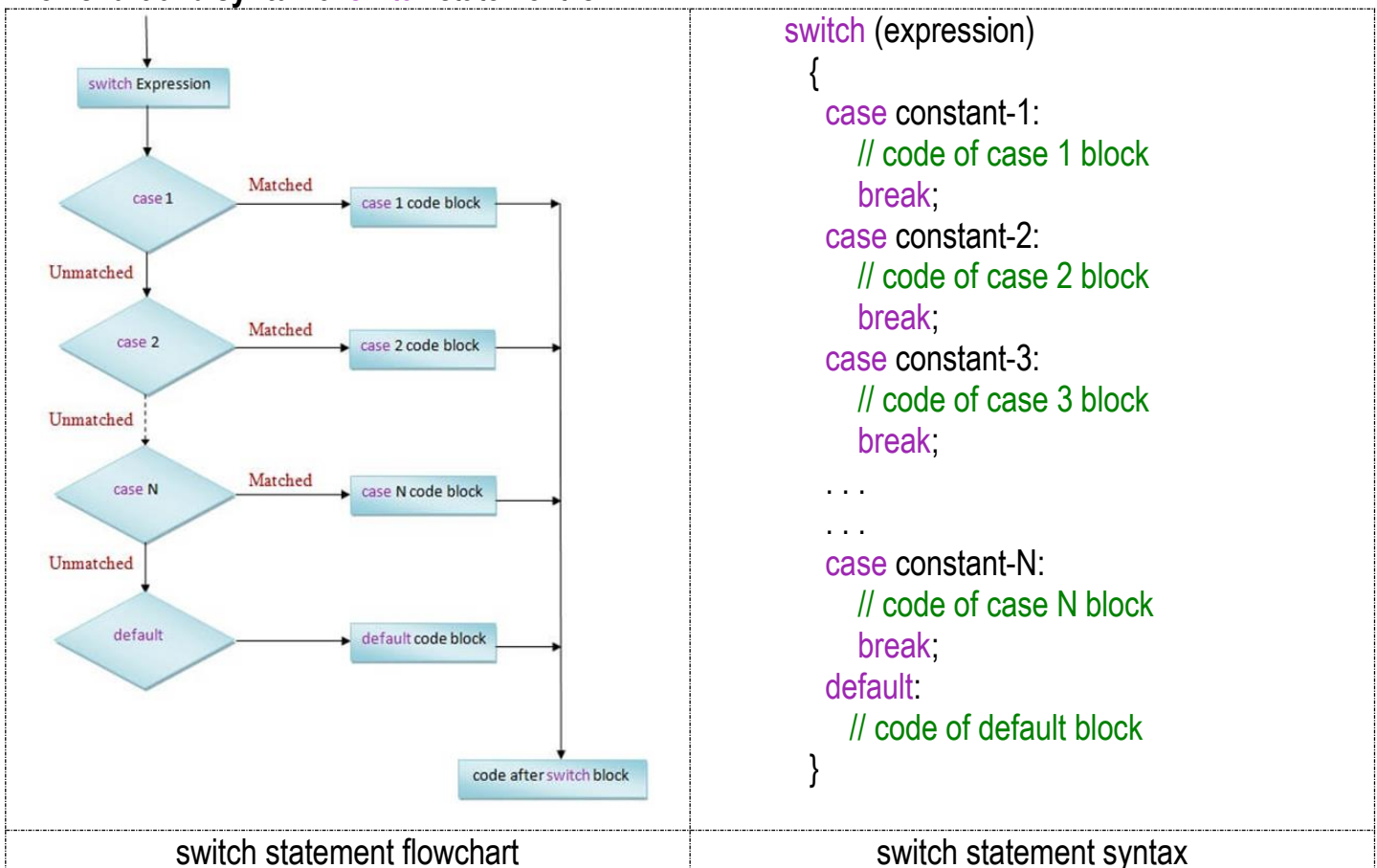
```

In above program, **a** and **b** are integers, so **a+b** will also be an integer which is equivalent to return type of method sum which is also an **int**.

NOTE: continue statement cannot be used outside of a loop, if used it will result in compile time error. Use continue statement more precisely inside while and **do while block**, if not used precisely, it may result as infinite loop. Ensure all letters of the **continue keyword** are **small**, cannot use **Continue**, **contiNue**, **CONTINUE** etc. Ensure all letter of return keyword is small, cannot use **Return**, **RETURN** etc. return keyword can be used anywhere inside method, but not outside method.

JAVA SWITCH CASE STATEMENT

Sometime you may need to select one block of code from many blocks of codes. To do this java gives a **switch statement** which allows you to select a block from many blocks of statements. The **flowchart** and **syntax** of **switch** statement is:



A **switch** statement evaluates its expression and then selects the matching block. The working of **switch** statement is very similar to **if-else** if ladder, which means the same thing can be achieved using **if-else** if statement. **Does the switch statement move directly to matching case? No**, It starts matching from top to bottom, the one that is matched, the code of that block is executed.

A programmer should remember below points while using **switch statement in java.**

- The expression used in a switch statement should be of type byte, short, char, int, enums and String(Java SE 7 and above). Wrapper classes Character, Byte, Short and Integer can also be used.
- A programmer can have any number of case statements within a switch block. Each case statement must be followed by the value to be compared to and a colon(:).
- Each case value must be unique. Duplicating a case value will result in compile time error.
- The type of value for every case must be same as data type of switch statement expression. Each case value must be a constant or a literal. You cannot use variables, for example case num1, case num2 are incorrect declaration.
- There can be multiple lines of code(statements) for each case.
- When a case is matched, all statements of that case will run until a break statement is reached.
- When a break statement is reached, the switch block terminates, and the flow of control jumps to the next line after the switch block.
- Use of break statement is optional, If not used, all the cases(including codes inside it) after the matching case will execute until a break statement or end of switch statement is reached.
- The default case in switch statement is optional. It can be used to perform any tasks when no cases is matched.

SWITCH CASE PROGRAM IN JAVA

```
class SwitchStatement
{
    public static void main(String [] args)
    {
        int dayNo = 4;
        String dayName;
        switch (dayNo)
        {
            case 1:
                dayName = "Sunday";
                break;
            case 2:
                dayName = "Monday";
                break;
            case 3:
                dayName = "Tuesday";
                break;
            case 4:
                dayName = "Wednesday";
                break;
```

```

    case 5:
        dayName = "Thursday";
        break;
    case 6:
        dayName = "Friday";
        break;
    case 7:
        dayName = "Saturday";
        break;
    default:
        dayName = "Invalid day";
        break;
}
System.out.println("day name = "+dayName);
}
}

```

NOTE: Conventionally you should use default statement in last, though it's not mandatory but prefer to follow this. **Keywords** are case-sensitive, you cannot use like Switch, SWITCH, switch etc. Every letter should be small letter. return statement can also be used inside switch cases, once executed the flow will be exited from the method itself. Expressions like $a+b$, $a+b*c$ etc. can also be used in switch statement.

JAVA PROGRAMS OF CONTROL FLOW STATEMENTS

Let see some basic useful programs as beginners. These programs use the control flow statements in some-way. These programs will help us to **build different logics** for different types of programs. Let's see the first program which shows how to find if a number is even or odd.

```

class EvenOddProgram
{
    public static void main(String [ ] args)
    {
        int num1 = 22;
        if(num1 % 2 == 0)
        {
            System.out.println(num1+ " is an even number");
        }
        else
        {
            System.out.println(num1+ " is an odd number");
        }
    }
}

```

Java Program of even odd number

```

class NumberTable
{
    public static void main(String [ ] args)
    {
        int num1 = 8;
        for(int i=1; i<=10;i++)
        {
            System.out.println(num1*i);
        }
    }
}

```

Java code to print multiplication table of a number

```

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

```

```

class FindingGrade

```

```

{
    int num1 = 40, num2 = 20, num3 = 50;
    if(num1 > num2)
    {
        if(num1 > num3)
            System.out.println("Greatest number = "+num1);
        else
            System.out.println("Greatest number = "+num3);
    }
    else
    {
        if(num2 > num3)
            System.out.println("Greatest number = "+num2);
        else
            System.out.println("Greatest number = "+num3);
    }
}

```

Java Program to find greatest of three number

```

{
    public static void main(String [] args)
    {
        int marks = 65;
        if(marks >= 80)
            System.out.println("A+ grade");
        else if(marks >= 60 && marks < 80)
            System.out.println("A grade");
        else if(marks >= 40 && marks < 60)
            System.out.println("B grade");
        else if(marks >= 30 && marks < 40)
            System.out.println("C grade");
        else
            System.out.println("D grade");
    }
}

```

Java Program for finding a grade

A % returns a remainder. If a number is divisible by **2**, it's an **even number**. **22** is divisible by **2**, and returns a remainder **0** which means **22** is an **even number**. Just change the **num1** as **23**, the output will be "**23 is an odd number**." As a beginner you want to write a program which prints **a multiplication table of a number**. Program above shows how to print multiplication table of a number, just change the **num1** as any other integer number, the program will display the table of that number.

Sometime you will need to write a program to find the greatest of three number, program below shows how to find the greatest of three number. You can change the value of **num1**, **num2**, **num3** the program will print the greatest of all three numbers. This is one approach to find the greatest number, there could be other approaches as well.

JAVA ARRAY'S INTRODUCTION

We all know that java provides **primitive data types** to store single values like 20, 100, 20.5 etc in a **variable**. But what if you want to store many values of the **same data type** like 20, 30, 40 or 10.5, 20.4, 30.6 etc in a **single variable**, one style could be, to create **many variables** and assign single values in each variable. Another easy way is to use **arrays** provided by java. An **array in java is a container which allows us to store many values of same data type in one variable**. The **syntax** of declaring an array is:

```

dataType[] arrayName;
dataType arrayName[];

```

You can use any of the above two forms to declare an array in java. For example:

```
int[ ] intArray; // declares an array of integer with name as intArray
int intArray2[ ]; // declares an array of integer with name as intArray2
float[ ] floatArray; // declares an array of float with name as floatArray
char[ ] charArray; // declares an array of char with name as charArray
```

The `[]` is used with the data types or array name suggests that it's an array. The **data type** of an array can be **primitive or non-primitive**, while the name of the array is given by you. **The name must follow the rules and convention given in Identifier-naming-convention section.** Arrays of **primitive data types** stores values while arrays of **non-primitive data types** store object references.

WHICH APPROACH I SHOULD PREFER TO DECLARE AN ARRAY?

Though you can use any of the above form, **it's good practice to use first one as it is more meaningful.** The code `int[] intArray` itself suggest that variable `intArray` is an `int` type array. Java convention also discourage to use the second form which is `int intArray[]`. The code given below shows how to declare an array of **non primitive data type**. Assuming that you have already created a class `MyFirstProgram`. To declare an array of `MyFirstProgram`, use below syntax:

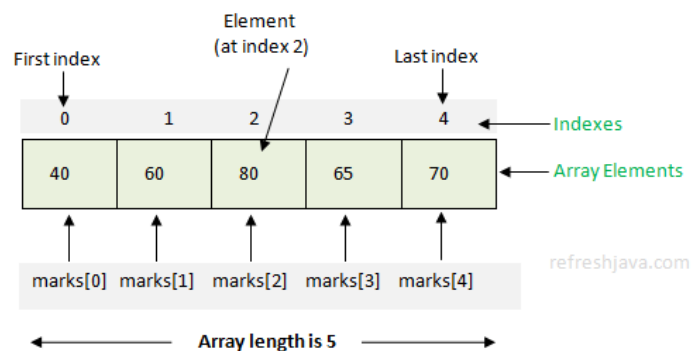
```
MyFirstProgram[] mfpArray; // declares an array of MyFirstProgram class with name
as mfpArray
```

HOW TO INITIALIZE ARRAY IN JAVA

Java arrays initializes array values in a **continuous memory location** where each memory location is given an index. You can assign or access the value to that memory location using its index. The **index** starts from **0** and goes as **0,1,2,3,4....** The first approach to create or initialize an array in memory is by using **new** keyword. The code below initializes an array in memory with size as 5 and then assigns values in it using its indexes.

```
int[] marks = new int[5];
marks[0] = 40; // initialize first element
marks[1] = 60; // initialize second element
marks[2] = 80; // initialize third element
marks[3] = 65; // initialize fourth element
marks[4] = 70; // initialize fifth or last element

// initializing in one line
int[] marks = new int[]{40,60,80,65,70};
```



Java Array Initialization

HOW TO ACCESS VALUES OF ARRAY

Each item in an array is called an element, and each element of array is **accessed** by its index. The index of first element is 0.

```
marks[0] // To access element 40
marks[1] // To access element 60
marks[2] // To access element 80
```

```
marks[3] // To access element 65
marks[4] // To access element 70
Min or start index of an array = 0;
Max or last index of an array = length of array - 1;
```

WHAT IS LENGTH OR SIZE OF AN ARRAY?

The **maximum** or **total number** of elements that can be assigned into the array is known as **length** or **size of an array**. You can get the length of an array using the **length property of array**. The second and shortcut approach to initialize an array in memory is by directly assigning array values into array variable like below :

```
int[] intArray2 = {20,40,60}; // initializes an int array with size as 3
int[] intArray3 = {
    20,30,40,
    50,60,70,
    80,90,100
}; // initializes an int array with size as 9
// Do not use comma (,) after last element.
```

The length of array in above declaration is determined by the number of values given inside the **{ }** and separated by **comma (,)**. You can assign elements of array in one line or multiple line as seen above.

Can I increase the length or size of an array after initialization? No, once an array is created or initialized, the **length of array is fixed**. You can't **increase** or **decrease length** of the array after initialization. For e.g. if you create any array of length 5 as `int[] marks = new int[5];` You cannot access or assign value at 5th index(`marks[5]`) which is 6th element, if you access `marks[5]`, java will throw a runtime exception.

You can also assign one array into another array like below. Just remember in this case if you make any changes in element of one array, that will be reflected in other as well.

```
int[] intArray2 = {20,40,60};
int[] intArray3 = intArray2; // intArray3 refers intArray2
```

What if I don't assign a value to a particular index? If a programmer doesn't specify a value to a particular index of an array, java will itself assign a value to that index as per the **data type** of array. For eg. 0 for **int**, 0.0 for **double**, false for **boolean** etc. For non-primitive data type, it assigns **null** for that index. **Can I pass array from one method to other? Yes**, You can pass array from one method to other method as you pass normal variables.

ARRAY PROGRAM IN JAVA

```
class ArrayInJava
{
    public static void main(String [] args)
    {
        int[] marks = new int[5];
        marks[0] = 40;
        marks[1] = 60;
        marks[2] = 80;
```

```

marks[3] = 65;
marks[4] = 70;

System.out.println("Element at index 0 = "+marks[0]);
System.out.println("Element at index 1 = "+marks[1]);
System.out.println("Element at index 2 = "+marks[2]);
System.out.println("Element at index 3 = "+marks[3]);
System.out.println("Element at index 4 = "+marks[4]);

// Accessing array elements using for loop;
System.out.println("Accessing array element using for loop .....");
for(int i=0; i<5; i++)
{
    System.out.println("Element at index "+i+" = "+marks[i]);
}
}
}

```

Output:

```

Element at index 0 = 40
Element at index 1 = 60
Element at index 2 = 80
Element at index 3 = 65
Element at index 4 = 70

Accessing array element using for loop .....
Element at index 0 = 40
Element at index 1 = 60
Element at index 2 = 80
Element at index 3 = 65
Element at index 4 = 70

```

The program below calculates the average of given integer numbers of an array. It also shows how to use the **length** property of array which returns the length of an array.

```

class ArrayDemo
{
    public static void main(String [] args)
    {
        int[] numbers = {40,60,80,65,70};
        double avgNumber;
        int total = 0;
        // finding sum of all numbers of array;
    }
}

```



```

    for(int i=0; i < numbers.length; i++)
    {
        total = total+numbers[i];
    }
    avgNumber = total/5;
    System.out.println("Average of numbers of array = "+avgNumber);
}
}

```

NOTE: Arrays with single [] brackets is also known as one dimensional array. Arrays discussed in this tutorial is single dimension arrays, for multidimensional arrays refer next section. Array elements starts from index 0, not 1. If you access array variable name, java will return reference(address) of that variable. Accessing any elements outside array index will throw `ArrayIndexOutOfBoundsException` at runtime. A method can return an array as well to calling method.

JAVA MULTIDIMENSIONAL ARRAYS

In java you can define an array whose elements are itself an array. Are called **multidimensional array**. A **multidimensional array** is an array of arrays which simply means the elements of such arrays will itself be an array. They are of different types for example **two dimensional(2D)**, **three dimensional(3D)**, **four dimensional(4D)** etc. We will see the two- and three-dimensional arrays.

JAVA TWO DIMENSIONAL (2D) ARRAYS

This is the simplest form of multidimensional array. A **2D array** is an array of one-dimensional arrays which means each element of two-dimensional array will be a single dimensional array. A 2D array has two dimension which is also called as row and column of **2D array**. This type of array has **two indexes** row index and column index. You can access or assign value in **2D array** using the combination of row and column indexes. **Row** and **column indexes** start from 0. **2D array** is similar to **matrixes** in mathematics where we have rows and columns.

HOW TO DECLARE 2D ARRAYS

The **syntax** of declaring 2D array is:

```
DataType[][] arrayName;
```

Example:

```

int[][] intArray; // declares a 2D array of integer with name as intArray
double[][] doubleArray; // declares a 2D array of double with name as doubleArray
char[][] charArray; // declares a 2D array of char with name as charArray

```

The **double brackets []** indicates that it's a 2D array. You can use this bracket after the data type or variable name as given in the previous tutorial. The data type can be primitive or non-primitive while the name of array is given as per the programmer's choice.

HOW TO INITIALIZE 2D ARRAYS

```

// First approach
int[][] matrix = new int[2][3];

```

```

matrix[0][0] = 10;
matrix[0][1] = 20;
matrix[0][2] = 30;
matrix[1][0] = 15;
matrix[1][1] = 80;
matrix[1][2] = 50;

```

// 2nd approach

```
int[][] a = {{15,20,25,30},{20,30,40,50},{60,65,70,80}};
```

In **first approach** the **value** given in **first bracket []** represents **number of rows** while **value** given in **second bracket []** represents **number of columns**. So in code **new int[2][3]**, number of **rows** is **2** and number of **column** is **3**. In **second approach**, each array inside the declaration represents a **row** while elements inside each array represents the number of columns. So, the **number of rows** in **2nd approach** is **3** and number of columns is **4**. Each **array(rows)** in second approach can have different number of **values(columns)**. E.g., The 1st array may have 4 elements, 2nd array may have 2 elements and 3rd array have 3 elements. Such arrays are called **jagged arrays**. Following example represents a jagged array.

```
int[][] a = {{15,20,25,30},{20,30},{60,65,70}};
```

Since a 2D array is an array of 1D arrays, you can think of the arrays given in {{15,20,25,30},{20,30,40,50},{60,65,70,80}}; like below

```
{a[0],a[1],a[2]};
```

Here a[0], a[1], a[2] are arrays itself which points the corresponding arrays. Similarly in declaration **int[][] matrix = new int[2][3]**; matrix[0] and matrix[1] will be an array. The image below displays how 2D array can be represented in rows and columns.

		Column1	Column2	Column3	Column4		
Row indexes		0	1	2	3	Column indexes	
Row 1	0	a[0][0] 15	a[0][1] 20	a[0][2] 25	a[0][3] 30	← Array a[0]	
Row 2	1	a[1][0] 20	a[1][1] 30	a[1][2] 40	a[1][3] 50	← Array a[1]	
Row 3	2	a[2][0] 60	a[2][1] 65	a[2][2] 70	a[2][3] 80	← Array a[2]	

To **access** values in 2D arrays, you use it's row and column indexes like **a[0][0]**, **a[0][1]**, **a[0][2]** and so on. In above example a[0][0] will return 15, a[0][1] will return 20 and a[0][2] will return 25.

What is the length of 2D array? The length of 2D array is the total number of arrays declared inside it. In other way the total number of rows inside a 2D array is the length of 2D array. For example in

declaration `int[][] a = {{15,20,25,30},{20,30,40,50},{60,65,70,80}};` length is 3 while in declaration `new int[2][3]` length is 2.

What is the total number of elements that can be stored in 2D array? To find the total number of elements that can be stored in 2D arrays, you can multiply number of rows and columns. For example in declaration `int[][] matrix = new int[4][5]`, total number of elements that can be stored is $4 \times 5 = 20$.

2D ARRAY PROGRAM IN JAVA

```
class TwoDArray
{
    public static void main(String [] args)
    {
        int[][] matrix = new int[2][2];
        matrix[0][0] = 10;
        matrix[0][1] = 20;
        matrix[1][0] = 30;
        matrix[1][1] = 40;

        System.out.println("Element at index 00 = "+matrix[0][0]);
        System.out.println("Element at index 01 = "+matrix[0][1]);
        System.out.println("Element at index 10 = "+matrix[1][0]);
        System.out.println("Element at index 11 = "+matrix[1][1]);

        int[][] a = {{15,20,25},{20,30,40},{50,60,70}};

        System.out.println("\nElement at index 00 = "+a[0][0]);
        System.out.println("Element at index 01 = "+a[0][1]);
        // Accessing array elements using for loop;
        System.out.println("\nAccessing array element using for loop .....");
        for(int i=0; i < a.length; i++)
        {
            for(int j=0; j < a[i].length; j++)
            {
                System.out.println("Element at index "+i+" "+j+" = "+a[i][j]);
            }
        }
    }
}
```

JAVA THREE DIMENSIONAL OR 3D ARRAYS

A three-dimensional array is an array of 2D arrays, which means each element in 3D array will be a 2D array. Each element in this array is also accessed by its indexes.

HOW TO DECLARE 3D ARRAYS

The syntax of declaring 3D array is:

```
DataType[][][] arrayName;
```

Examples:

```
int[][][] intArray; // declares a 3D array of integer with name as intArray
```

```
double[][][] doubleArray; // declares a 3D array of double with name as doubleArray
```

Every declaration in this array is same as 2D array except one more `[]` bracket. The triple `[][][]` indicates that it's a 3D array.

HOW TO INITIALIZE 3D ARRAYS

// First approach

```
int[][][] matrix = new int[2][2][2];
```

```
matrix[0][0][0] = 10;
```

```
matrix[0][0][1] = 20;
```

```
matrix[0][1][0] = 30;
```

```
matrix[0][1][1] = 40;
```

```
matrix[1][0][0] = 80;
```

```
matrix[1][0][1] = 90;
```

```
matrix[1][1][0] = 15;
```

```
matrix[1][1][1] = 25;
```

// 2nd approach

```
int[][][] a = {{{15,20},{30,40}},{25,50},{60,80}}};
```

What is the length of 3D array? The length of 3D array is the total number of 2D arrays declared inside it. For example in declaration `int[][][] a = {{{15,20},{30,40}},{25,50},{60,80}}};` length is **2** while in declaration `new int[2][3][4]` length will be **2**.

What is the total number of elements that can be stored in 3D array? To find the total number of elements that can be stored in 3D arrays, just multiply the values given in `[]` brackets. For example in declaration `int[][][] matrix = new int[4][5][2];` total number of elements that can be stored is $4 \times 5 \times 2 = 40$.

3D ARRAY PROGRAM IN JAVA

```
class ThreeDArray
{
    public static void main(String [] args)
    {
        int[ ][ ][ ] a = {{{15,20},{30,40}},{25,50},{60,80}}};

        System.out.println("Element at index 000 = "+a[0][0][0]);
        System.out.println("Element at index 001 = "+a[0][0][1]);
        // Accessing 3D array elements using for loop;
        System.out.println("\nAccessing 3D array elements using for loop .....");
        for(int i=0; i < a.length; i++)
        {
            for(int j=0; j < a[i].length; j++)
            {
                for(int k=0; k < a[i][j].length; k++)
                {
                    System.out.println("Element at index "+i+" "+j+" "+k+" = "+a[i][j][k]);
                }
            }
        }
    }
}
```

JAVA FOR EACH LOOP

If you have to **traverse(iterate)** an array or collection elements using normal **for loop**, you may have to write the **for-loop expressions** like **initialization**, **condition**, **increment** or **decrement** which is a bit painful and sometimes wrong declaration may result as infinite loop. To simplify this, java offers another looping statement known as **for-each loop**. The **for-each loop** is an enhanced form of normal **for** loop which is used to traverse elements of **arrays** or **collection**. It was introduced in java 5 just to simplify the traversing of arrays and collection elements. The **Syntax** of for-each loop is:

```
for(DataType item : array/collection variable )
{
    // Code to be executed once for each element in array/collection.
}
```

Here **item** is the given **variable name** by a programmer while the **data type** of this variable should be the same or compatible with data type of the array/collection variable. Array/collection variable is the array or collection variable whose elements need to be **traverse**. You must read the **colon(:)** as "**in**", so that the loop above is read as "for each item in array/collection variable".

What is collection in java? is a group of data structures(classes and interfaces). Variable of these data structures can hold many elements like arrays do. E.g. ArrayList, LinkedList, HashSet, HashMap etc are some of the collection data structures. **Why it is called for-each loop?** Since the loop iterates through each element of array/collection. **When should I prefer for-each loop over normal for loop?** When you just want to traverse each element of array/collection one by one.

FOR EACH LOOP PROGRAM IN JAVA

```
class ForEachLoop
{
    public static void main(String [] args)
    {
        int[] marks = new int[]{40,60,80,65,70};
        System.out.println("Accessing array element using normal for loop .....");
        for(int i =0; i < marks.length; i++)
        {
            System.out.println(marks[i]);
        }
        System.out.println("Accessing array element using for-each loop .....");
        for(int num : marks)
        {
            System.out.println(num);
        }
    }
}
```

We see you don't have to write the expressions like initialization, condition, increment/decrement which is used in normal **for** loop and it's also easy to write and more readable as well. So, you should prefer to

use the **for-each loop** over normal **for** loop while traversing array/ collection elements. You should read this as, "*for each num in marks*" execute the code inside the body.

How for each loop works

- It picks one element from array/collection.
- Assigns that element in variable given in left side
- Executes the body of for-each loop
- Then again moves to first step for the next element or finishes if all the elements are traversed.

foreach vs for loop

You cannot **modify** the elements of array/collection using the for-each loop since don't get the indexes while the same can be done in a normal **for** loop. E.g. below thing cannot be achieved using the for-each loop.

```
for(int i = 0; i < marks.length-1; i++)
{
    if(i==3)
    {
        marks[i] = 70;
    }
}
```

You can traverse an array/collection from **last to first** using normal **for** loop as seen below, but the same cannot be achieved using for-each loop as it iterates only forward over the array/collection.

```
for(int i = marks.length-1; i >= 0; i--)
{
    System.out.println(marks[i]);
}
```

You can access **many elements** in a **single iteration** using the **for** loop as given below, but the same cannot be achieved using for-each loop.

```
for(int i = 0; i < marks.length-1; i++)
{
    if(marks[i] == marks[i+1])
    {
    }
}
```

Prefer the normal **for loop** over **for-each loop** whenever **indexes** are needed while iteration as for-each loop doesn't keep track of indexes.

STRINGS IN JAVA INTRODUCTION

String is one of the most widely used **data type** in java. **Words** that we usually use in conversations like "hello", "refresh java", "Hello world", "It's easy to learn java", name of a person, address of a person etc are some of the examples of **string in java**. In java, a **string is set of characters**, for example the string "hello" is a sequence of characters 'h', 'e', 'l', 'l', 'o'. Similarly, each string in java is a sequence of characters. **Java** provides **String class** to create and manipulate strings in programs. This class is defined in **java.lang** package which comes automatically in your java software(**JDK or JRE**).

Since a class in java acts as a **non primitive data type**, so **String** is also a non-primitive data type. You can use **String** class as a data type to store string values. The **Syntax** of declaring a string is:

```
String var_name = "java by herbert";
String var_name = new String("Hello ISBAT");
```

Here **String** is the **data type**, the **var_name** is the name of variable given you and the value of the string variable is given inside **" "**. You need to use the **double quotes(" ")** to create the string values in java. **Which approach should I prefer to declare a string, "" or new keyword?** Though you can use any of the above approach but creating a string using **new** operator is not commonly used and is not recommended as it creates an extra object inside heap memory.

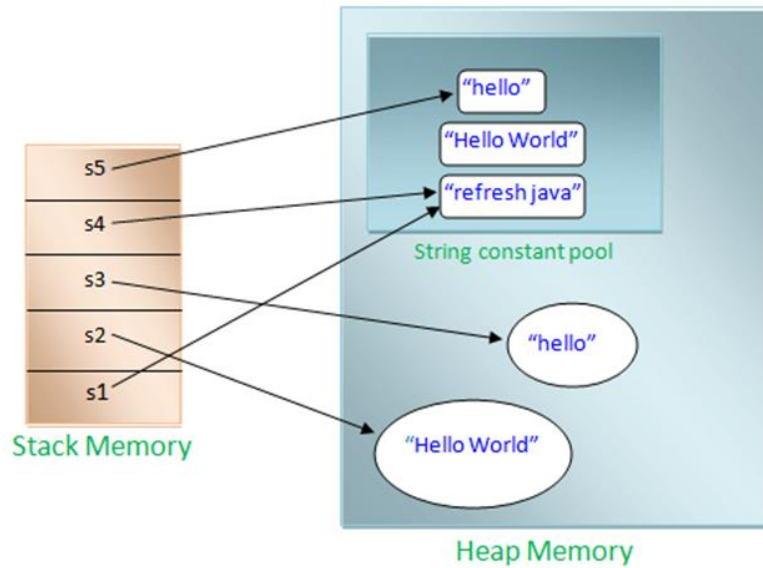
STRING CLASS PROGRAM IN JAVA

```
class StringDemo
{
    public static void main(String args[])
    {
        // Creating String variables
        String strVar = "refresh java";
        String strVar2 = new String("Hello World");
        char[] charArray = {'h', 'e', 'l', 'l', 'o'};
        String strVar3 = new String(charArray);
        String strVar4 = strVar;
        // Printing the values of String variables
        System.out.println("strVar = "+strVar);
        System.out.println("strVar2 = "+strVar2);
        System.out.println("strVar3 = "+strVar3);
        System.out.println("strVar4 = "+strVar4);
    }
}
```

JAVA STRING MEMORY ALLOCATION

String values in java are stored inside a special memory called a **string pool** or **string constant pool** which exist inside the **heap memory**. This memory is used to allocate space for string values only. **String values** given inside the **" "** in a program are made in **a string pool**. To understand memory allocation of strings, let's consider the following string variables declared inside a program.

1. String s1 = "refresh java";
2. String s2 = new String("Hello World");
3. String s3 = new String("hello");
4. String s4 = "refresh java";
5. String s5 = "hello";



Let's understand how java assigns memory for above strings variables.

- Line 1, java checks if string literal "ISBAT java" is already in string pool or not, since it's not there, java creates this string literal in string pool and assigns the reference(address) of this in **s1**.
- Line 2 is creating object **s2** using **new** keyword, so java will create it inside the heap memory with value as "Hello World" and it will create the string literal("Hello World") inside the string pool as well.
- Line 3 is creating object **s3** using **new** keyword, so it will also be created inside the heap memory with value as "hello" and the same string literal("hello") would also be created inside string pool.
- Line 4, java again checks the string literal "refresh java" inside the string pool, since it's already there in pool, so it won't create it again. Java assigns the reference of existing string literal in **s4**.
- Line 5, java again checks the string literal "hello" inside string pool, since it's already there in pool, so it won't create it again. Java assigns the reference of existing string literal in **s5**.

Why does java use a special memory for string? Because a String is one of the most useful data type with improved memory usage and application performance. So, java uses string pool memory as it doesn't create unnecessary or duplicate objects, instead it uses the existing string objects. **What is string literal in java?** A series of characters in your program that are enclosed in double quotes("") is a string literal. E.g. "hello world", "refresh java", "java is easy to learn" are string literals in java. Whenever it encounters a string literal in your code, the compiler creates a String object with its value in string constant pool. String literals are string constants in java. **How can I check if two string variables or literal have same reference(address)?** You can use **==** operator to compare if two string variables or literals have same address. E.g. **s1==s2, s1==s4** etc.

STRING CONCATENATION IN JAVA

This is a common concept in java programmers. **String concatenation** is a way of concating(adding) two or more string values. Java provides the feature of adding two or more string variable or values using plus(+) operator. E.g. you can add two string values "refresh" and "java" as "refresh"+"Java" bwhich results as a new string "refreshJava". Java also provides the **concat** method in **String** class to concat two string variables or values. You can refer next tutorial about this method.

JAVA PROGRAM TO CONCATENATE TWO STRING

```
class StringConcatDemo
{
```

```

public static void main(String args[])
{
    String s1 = "ISBAT";
    String s2 = "Java";
    s1 = s1+s2;
    s2 = " ISBAT " + s2 + " tutorial";
    System.out.println("s1 = "+s1);
    System.out.println("s2 = "+s2);
}
}

```

A JAVA STRING IS IMMUTABLE

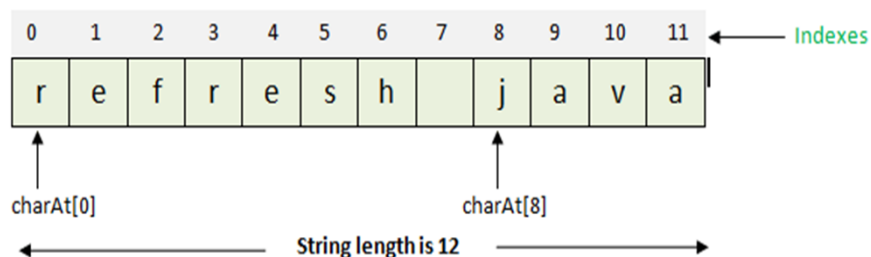
Immutable simply means something that is not changeable, once a string value in java is created, you can't change that value. In the above program, the code `s1+s2` creates a new string " ISBAT Java" and assigns the reference of this in `s1` but the original string "refresh" will still be there in string pool memory, that is why string in java is immutable.

Why a string is immutable in Java? Since immutable strings cannot be changed, the same string can be referenced in multiple programs or applications which significantly improves the memory usage and application performance.

NOTE: String pool, string constant pool and string literal pool and string intern pool are same things. **Digits** can also be stored as string in java, e.g. `String str = "12345";` is a valid string. A '+' operator, which does **addition** on **primitives** (i.e. **int** and **double**), is overloaded to perform concatenation on String objects. A compiler, uses the append method of **StringBuffer/StringBuilder** class to implement string concatenation.

USEFUL STRING CLASS METHODS IN JAVA

You may want to do some types of operations on strings like "**compare two strings, find or replace a particular character/word in a string, add two strings, extract a part of a given string**" etc. As java designer, there are a number of methods in **String class** which you can use in your program to perform such type of operations on strings. To understand the **String class** methods better, let's first understand how a **string value** in java is stored in memory. **String in java is an array of characters, so string values in java is stored as an array of characters where each characters have an index number.** Some **String class methods** use indexes to perform operations on strings. The image below shows how a string value "refresh java" is stored as an array of characters in string pool memory.



As you see, each characters is given an **index number** and you can access the character at a given index using **charAt() method of String class** which returns the character of a **particular index**. The **length of a string is the total number of characters in string**. To understand the String class methods better let's consider we have three string variables **s1**, **s2** and **s3** as given below.

```
String s1 = "refresh java";
String s2 = "Hello World";
String s3 = "learn java on refresh java tutorial";
```

HOW CAN I CALL A METHOD OF STRING CLASS IN A PROGRAM? To access a method of String class, you use the **dot(.) operator** with **a string variable** or a **string literal**, e.g. `s1.length()`, `s1.equals(s2)`, `"Hello".equalsIgnoreCase("hello")`, `s1.charAt(8)` etc.

Table below shows the list of methods which are used often with string. You can refer oracle documentation of String class to get the detail of all methods.

NAME	DESCRIPTION	EXAMPLE
int length()	It returns the length of string. The length is equal to the total number of characters in string.	<code>int length = s1.length();</code> <code>//length = 12</code>
boolean equals(Object anObject)	Used to compare two strings. It returns true if both string represents same sequence of characters else false.	<code>s1.equals(s2);</code> <code>// returns false</code> <code>s1.equals("refresh java");</code> <code>// returns true.</code>
boolean equalsIgnoreCase(String anotherString)	Used to compare two strings, ignoring the case(upper or lower case). It returns true if both the string are of same length and represents same sequence of character ignoring the case else returns false.	<code>s1.equalsIgnoreCase("Refresh JAVA");</code> <code>// returns true</code>
boolean startsWith(String prefix)	checks if a string starts with the string represented by prefix.	<code>s1.startsWith("refresh");</code> <code>// returns true.</code> <code>s1.startsWith("Refresh");</code> <code>// returns false.</code>

int indexOf(String str)	Used to search if a substring(represented by str) exist in the given string. Returns the index of first occurrence of substring in the given string if found else returns -1.	s1.indexOf("java"); // returns 8 s1.indexOf("hello"); // returns -1
int indexOf(String str, int fromIndex)	Used to search if substring(represented by str) exist in the given string, but it starts searching from the index given by fromIndex. It returns starting index of that substring if found else returns -1.	s3.indexOf("java", 10); // returns 22 s3.indexOf("easy", 25); // returns -1
int lastIndexOf(String str)	Used to search the last occurrence of substring(represented by str) in the given string. It returns the index of last occurrence of substring if found else returns as -1.	s3.lastIndexOf("java"); // returns 22
char charAt(int index)	Returns the character at specified index in given string.	char charVar = s1.charAt(8); // charVar = 'j'
String substring(int fromIndex)	Returns a string that is a substring of the given string. The substring starts with the character at fromIndex position and extends to the end of given string.	String str = s1.substring(8); // str = "java", s1 will still be same that is "refresh java" String str2 = s3.substring(27); // str2 = "tutorial", s3 will still be same.
String substring(int fromIndex, int endIndex)	Returns a string that is a substring of the given string. The substring starts with the character	String str = s1.substring(0,7); // Return characters from 0 to 6 str = "refresh"

	at fromIndex position and extends to the (endIndex-1)	String str2 = s3.substring(14,26); // str2 = "refresh java"
String toLowerCase()	Converts all the characters of given string to lower case and returns that string.	String str = s2.toLowerCase(); // str = "hello world" s2 will still be same that is "Hello World".
String toUpperCase()	Converts all the characters of given string to upper case and returns that string.	String str = s1.toUpperCase(); // str = "REFRESH JAVA", s1 will still be same.
String trim()	Used to remove whitespace() at the beginning and end (not in between) of a given string. Returns a string without any whitespace in end and beginning.	String str = " hello world" .trim(); // str = "hello world"
String concat(String str)	Concatenates(adds) the string represented by str to the end of given string	String strVar = s1.concat("tutorial"); // strVar = "refresh java tutorial", s1 will still be same.
char[] toCharArray()	Converts the given string to a new character array and returns that char array.	char [] charArray = s1.toCharArray(); // charArray will contain an array of characters(r,e,f,r,e,s,h,j,a,v,a)
String replace(char oldChar, char newChar)	Returns a string resulting from replacing all occurrences of character represented by oldChar with character represented by newChar in	String str = s1.replace('r', 'h'); // str = "hefhesh java", s1 will still be same.

	the given string.	
String replaceFirst(String regex, String replacement)	Used to replace first occurrence of string represented by regex with string represented by replacement in given string. regex can be a regular expression as well.	String str = s3.replaceFirst("java","program"); // str = "learn program on refresh java tutorial"
String replaceAll(String regex, String replacement)	Used to replace all occurrence of string represented by regex with string represented by replacement in given string, regex can be a regular expression as well.	String str = s3.replaceAll("java","program"); // str = "learn program on refresh program tutorial"
String[] split(String regex)	Used to split a given string on the basis of regex. It returns a string array.	String[] str = s3.split(" "); // str will contains strings "learn" "java" "on" "refresh" "java" "tutorial"

STRING CLASS METHODS PROGRAM IN JAVA

```

class StringClassMethods
{
    public static void main(String [] args)
    {
        String s1 = "refresh java";
        String s2 = "Hello World";
        String s3 = "learn java on refresh java tutorial";
        String str;

        // Getting the length of a string
        System.out.println("length of string s1 = "+s1.length());
        // Comparing two strings
        System.out.println(s1.equals("refresh java"));
        System.out.println("is s1 equals s2 = "+s1.equals(s2));
        // Checking if a string starts with a given substring
        System.out.println("is s1 starts with word refresh = "+s1.startsWith("refresh"));
        // Searching a substring in a string
        System.out.println("index of word java in s3 = "+s3.indexOf("java"));
        System.out.println("index of word hello in s3 = "+s3.indexOf("hello"));
    }
}

```

```

// Getting the character at given index in a string
System.out.println("character at index 8 in string s1 = "+s1.charAt(8));
// Extracting a substring from a given string
str = s1.substring(8);
System.out.println("Substring str = "+str);
// After extracting a substring, value of s1 will still remain same.
System.out.println("Value of s1 = "+s1);
// Converting a given string to lower case.
str = s2.toLowerCase();
System.out.println("Lower case str = "+str);
// Concatenating a string to given string.
str = s1.concat(" tutorial");
System.out.println("Concatenated str = "+str);
// Concatenating a string to given string using + operator.
str = str + " online";
System.out.println("+ operated value of str = "+str);
// Replacing a word in a given string.
str = s3.replaceFirst("java","program");
System.out.println("Replaced value str = "+str);
// Splitting a given string using space( ).
String[] strArray = s3.split(" ");
for(String strVar : strArray)
    System.out.print(strVar+" ", );
}
}

```

DIFFERENCE BETWEEN EQUALS() AND == IN JAVA

Both equals() method and == operator can be used to compare string variables or string literals in java but there is a difference between both. equals() method compares the content(the sequence of characters) of string variables or string literals. It returns **true** if both the content are same else returns **false** while == operator compares the references(address) of string variables or string literals. It returns **true** if both the variables or literals points to same reference else returns **false**. If the comparison of two strings using == operator returns **true** then equals() method for same comparison will definitely return **true** while vice-versa may or may not return **true**. The program given below shows the difference of equals() method and == operator.

```

class StringComparison
{
    public static void main(String [] args)
    {
        String str = "refresh java";
        String str2 = new String("refresh java");
        String str3 = "refresh java";
        String str4 = new String("refresh java");
        String str5 = str2;
    }
}

```



```

        System.out.println("Comparison using equals() method .....");
        System.out.println("str equals() str2 >> "+str.equals(str2));
        System.out.println("str equals() str5 >> "+str.equals(str5));
        System.out.println("str2 equals() str4 >> "+str2.equals(str4));
        System.out.println("str2 equals() str5 >> "+str2.equals(str5));

        System.out.println("Comparison using == operator .....");
        System.out.println("str == str2 >> "+(str == str2));
        System.out.println("str == str3 >> "+(str == str3));
        System.out.println("str2 == str4 >> "+(str2 == str4));
        System.out.println("str2 == str5 >> "+(str2 == str5));
        System.out.println("str4 == str5 >> "+(str4 == str5));
    }
}

```

STRINGBUFFER AND STRINGBUILDER IN JAVA

Apart from the **String** class java has two other classes to handle string values, **StringBuffer** and **StringBuilder**. The classes were added with some purpose, the difference between both and when to use **StringBuffer** or **StringBuilder**. **String** in java is **immutable**, which means you cannot change string value once created. When we do string manipulation like *concatenation*, *substring*, *replace* etc, it generates a new string and leaves the original string as is in memory which becomes **a garbage value**. For an example the code given below will generate a garbage value as "refresh java" in memory.

```

String str = "refresh java";
str = str + " tutorial";

```

The concatenation act above creates a **new string** as "refresh java tutorial" in string pool memory and assigns its reference in str but this operation won't change the original string "refresh java", it will still exist in pool memory as a garbage value.

WHAT IS A GARBAGE VALUE?

Any **value** or **object** **inside the memory that is not being referred by program or application is known as garbage**. The process of collecting and destroying such values from memory is known as **garbage collection**. The string manipulation operations doesn't cost much if it is not happening frequently, but if it's happening often and in many areas in your application, this may cause **serious memory leak** and **performance** issue as it creates a lot of **garbage** in memory. To solve this problem, java gives two more classes, **StringBuffer** and **StringBuilder**. String values stored in **StringBuffer** or **StringBuilder** data type **are mutable**. Meaning, if you apply any operation on **String values** or **objects stored in StringBuffer and StringBuilder classes** which are **mutable**, then you will change the original string itself rather than creating a new string. As a result, it will not generate garbage in memory. The **Syntax** of declaring StringBuffer and StringBuilder is:

```

StringBuffer var_name = new StringBuffer("StringBuffer value");

StringBuilder var_name = new StringBuilder("StringBuilder value");

// Example

```

```
StringBuffer strbuff = new StringBuffer("refresh java");
StringBuilder strbdr = new StringBuilder("java is easy");
```

Here **StringBuffer** and **StringBuilder** are the data type(classes), **var_name** is the name of variable given as per the programmers choice and the value of variable is given inside **"**. Java considers string literal(value given in **"**) as **String** type, so you cannot assign string literal directly to StringBuffer and StringBuilder variable as given below. You must have to use **new** keyword to create StringBuffer and StringBuilder.

// Both line will throw compilation error.

```
StringBuffer strbuff = "refresh java";
StringBuilder strbdr = "java is easy";
```

You can also not use **+** operator to **concat string** on **StringBuffer** and **StringBuilder** variables.

```
StringBuffer strbuff = new StringBuffer("refresh java");
StringBuilder strbdr = new StringBuilder("java is easy");
```

// Both line will throw compilation error.

```
strbuff = strbuff + " another value";
strbdr = strbdr + " another value";
```

How can I add another string value in StringBuffer or StringBuilder variable? You need to use **append()** method of these classes to add another string. Refer the program given below to see the use of this method.

StringBuffer and StringBuilder program in Java

```
class StringBufferAndBuilder
{
    public static void main(String args[])
    {
        String str = "refresh java";
        String str2 = str;
        StringBuffer strbuff = new StringBuffer("StringBuffer on refresh java");
        StringBuffer strbuff2 = strbuff;
        StringBuilder strbuild = new StringBuilder("StringBuilder on refresh java");
        StringBuilder strbuild2 = strbuild;

        str = str + " tutorial";
        strbuff = strbuff.append(" tutorial");
        strbuild = strbuild.append(" tutorial");

        System.out.println("str = "+str);
        System.out.println("str2 = "+str2);
        System.out.println("strbuff = "+strbuff);
        System.out.println("strbuff2 = "+strbuff2);
        System.out.println("strbuild = "+strbuild);
        System.out.println("strbuild2 = "+strbuild2);
    }
}
```

As you see, both **String** variable have different value while both the **StringBuffer** and **StringBuilder** variable have same value, that's because **String** is immutable. Once you add a string literal in **String** variable, java creates a new string object and assigns the reference of it in **String** variable while in case of **StringBuffer** and **StringBuilder**, the same string is modified.

USEFUL STRINGBUILDER AND STRINGBUFFER METHODS IN JAVA

The table below shows some useful methods of a **StringBuffer** and a **StringBuilder** classes which is usually used for string manipulation. Both classes have the same methods, the only difference is that methods of **StringBuffer** class are **synchronized**. Table assumes you have a variable `s1` of type **StringBuffer**.

NAME	DESCRIPTION	EXAMPLE
append(String str)	It appends the string represented by <code>str</code> at the end of given string.	<code>s1 = new StringBuffer("refresh");</code> <code>s1.append("Java");</code> <code>// s1 = "refreshJava"</code>
insert(int offset, String str)	Used to insert string represented by <code>str</code> at index specified by <code>offset</code> in given string.	<code>s1 = new StringBuffer("refresh");</code> <code>s1.insert(2, "java");</code> <code>// s1 = "rejavafresh"</code>
delete(int startIndex, int endIndex)	It deletes characters from <code>startIndex</code> to (<code>endIndex</code> -1) from given string.	<code>s1 = new StringBuffer("refresh java");</code> <code>s1.delete(2,7);</code> <code>// s1 = "re java"</code>
replace(int start, int end, String str)	It replaces characters from <code>start</code> to (<code>end</code> -1) index with string represented by <code>str</code> in given string.	<code>s1 = new StringBuffer("refresh java");</code> <code>s1.replace(0,7,"tutorial");</code> <code>// s1 = "tutorial java"</code>
reverse()	It reverses the sequence of characters of given string.	<code>s1 = new StringBuffer("refresh java");</code> <code>s1.reverse();</code> <code>// s1 = "avaj hserfer"</code>
setCharAt(int index, char ch)	Used to replace the character at given index with character represented by <code>ch</code> in given string.	<code>s1 = new StringBuffer("refresh");</code> <code>s1.setCharAt(0,'s');</code> <code>// s1 = "sefresh"</code>

CONVERSION FROM STRING TO STRINGBUFFER/STRINGBUILDER AND VICE-VERSA

You cannot directly assign a **String** variable in **StringBuffer**/**StringBuilder** or vice-versa. You need to use conversion like below to store it in other string data type variable.

```
String str = "refresh java";
// Both line will throw compilation error.
// StringBuffer strbuff = str;
// StringBuilder strbdr = str;
// Conversion from String to StringBuffer and StringBuilder.
```

```
StringBuffer strbuff = new StringBuffer(str);
StringBuilder strbdr = new StringBuilder(str);

// Both line will throw compilation error.
// String str2 = strbuff;
// String str3 = strbdr;
// Conversion from StringBuffer/StringBuilder to String.
String str2 = strbuff.toString();
String str3 = strbdr.toString();
```

STRING VS STRINGBUFFER VS STRINGBUILDER

- The String object is **immutable** while objects of StringBuffer and StringBuilder are **mutable**.
- StringBuffer is **thread safe** as its methods are **synchronized** while StringBuilder is not, so use StringBuffer in multi-threaded environment where thread safety is needed.
- StringBuffer is **not as faster as** StringBuilder for string manipulation operations, since methods of StringBuffer class is synchronized.
- StringBuilder was added in **java 1.5**, before that there was only String and StringBuffer classes for string handling.
- In single threaded environment, prefer String over StringBuilder if you don't need mutable objects as String is easier and more convenient to use. Use StringBuilder if mutable objects needed.

What is multi-thread environment? In a multi-threaded environment multiple programs or different parts(thread) of a program runs simultaneously. **Define synchronized in java?** **synchronized** is a keyword, use to achieve thread safety. A **synchronized** method cannot be accessed simultaneously by multiple threads. In most of the scenarios we don't use String data type in multithreaded environment where thread safety is needed, as it may cause inconsistent behavior. For example if a thread makes any changes in String variable, the other thread may not get the updated value which may cause inconsistent behavior of your program. To avoid such type of problem you should use StringBuffer class where thread safety is needed.

Refer java oracle docs or java source code of StringBuffer and StringBuilder class to get details of all methods. There are many methods in String and StringBuffer/StringBuilder classes that are available in each other but not all. Digits can also be stored as string in StringBuffer and StringBuilder class in java, for example StringBuffer str = new StringBuffer("12345"); is a valid string. The '+' operator, which performs addition on primitives (such as int and double), is overloaded to perform concatenation on String objects. The compiler, internally uses the append method of StringBuffer/StringBuilder class to implement string concatenation.

ARRAY AND STRING PROGRAMS IN JAVA

In this tutorial we will see some of the programs of array and string which are commonly asked with beginners in interviews. These programs will help them to build their logics in using array and string. Let's see the first program which shows how to check if a given number exist inside an array. If it exist, the program will also print it's index in the array.

PROGRAM TO SEARCH AN ELEMENT IN AN ARRAY

```
class SearchNumber
```

```

{
    public static void main(String [] args)
    {
        int [] numbers = {40,60,80,65,70};
        boolean isExist = false;
        int searchNumber = 80;
        int pos = -1;
        for(int i = 0; i < numbers.length; i++)
        {
            if(numbers[i] == searchNumber)
            {
                isExist = true;
                pos = i;
                break;
            }
        }
        if(isExist)
            System.out.println("number "+searchNumber+" exist in the array at index = "+pos);
        else
            System.out.println("number "+searchNumber+" does not exist inside the array");
    }
}

```

The next program shows how to find minimum and maximum element of an array. This is one way to find the minimum and maximum element, there can be other approaches as well.

JAVA PROGRAM TO FIND MAXIMUM AND MINIMUM NUMBER IN AN ARRAY

```

class MinMaxOfArray
{
    public static void main(String [] args)
    {
        int [] numbers = {40,20,80,65,70,90,35,10};
        int maxValue = numbers[0];
        int minValue = numbers[0];
        for(int i=1;i < numbers.length;i++)
        {
            if(numbers[i] < minValue)
                minValue = numbers[i];

            if(numbers[i] > maxValue)
                maxValue = numbers[i];
        }
        System.out.println("Minimum number = "+minValue+" and maximum number = "+maxValue);
    }
}

```

The next program shows how to sort an array of integer elements in ascending order. This is one approach to sort an array, there are other approaches as well to do the same.

JAVA PROGRAM TO SORT AN ARRAY

```
class SortArray
{
    public static void main(String [] args)
    {
        int [] numbers = {40,20,80,65,70,90,35,10};
        int length = numbers.length;
        int temp;
        for(int i = 0; i < length; i++)
        {
            for(int j = i + 1; j < length; j++)
            {
                if(numbers[i] > numbers[j])
                {
                    temp = numbers[i];
                    numbers[i] = numbers[j];
                    numbers[j] = temp;
                }
            }
        }
        System.out.println("Array elements in ascending Order:");
        for(int n : numbers)
            System.out.print(n + " ");
    }
}
```

The next program shows how to reverse a given string and checks if it is a palindrome string. A string is called palindrome if the reverse of the string is same as the original string. Again this is one way to reverse the string, there can be other ways as well to do the same. You can also use reverse() method of StringBuffer/StringBuilder class to reverse a given string directly but our intention is to learn the logic.

JAVA PROGRAM TO REVERSE A STRING AND PALINDROME STRING

```
class ReverseAndPalindromeString
{
    public static void main(String [] args)
    {
        String str = "abcba";
        String revStr = "";
        for(int i = str.length() - 1; i >= 0; i--)
        {
            revStr = revStr + str.charAt(i);
        }
        System.out.println("Reverse string = "+revStr);
        if(revStr.equals(str))
        {
            System.out.println("It is a palindrome string");
        }
    }
}
```



```

        System.out.println("The given string is a palindrome string");
    else
        System.out.println("The given string is not a palindrome string");
    }
}

```

Let's write one more program which counts total number of occurrence of a given character inside a given string. The program shows two different approaches to count the total occurrence.

PROGRAM TO FIND OCCURRENCE OF A CHARACTER IN A STRING IN JAVA

```

class CharacterCount
{
    public static void main(String[] args)
    {
        String str = "Java is easy to learn";
        char c = 'a';
        int count = 0;
        // First Approach
        char[] charArray = str.toCharArray();
        for(char ch : charArray)
        {
            if(ch == c)
                count++;
        }
        System.out.println("Total occurrence of character 'a' using 1st approach = "+count);

        // Second Approach
        int count2 = str.length() - str.replace("a", "").length();
        System.out.println("Total occurrence of character 'a' using 2nd approach = "+count2);
    }
}

```

WHAT IS A CLASS IN JAVA

Java is an **object-oriented programming** language. **Classes** and **objects** are key concepts to understanding **object-oriented programming** which is based on these two terms. Let's see what a **class is** and similarly discuss objects. Programmatically, **a class is a collection of variables, constructors, methods, blocks** which are optional class components. Class is defined using a **class** keyword followed by the **class name**. The name of the class is used to refer that class within or outside the class. The basic **syntax** of declaring a class is:

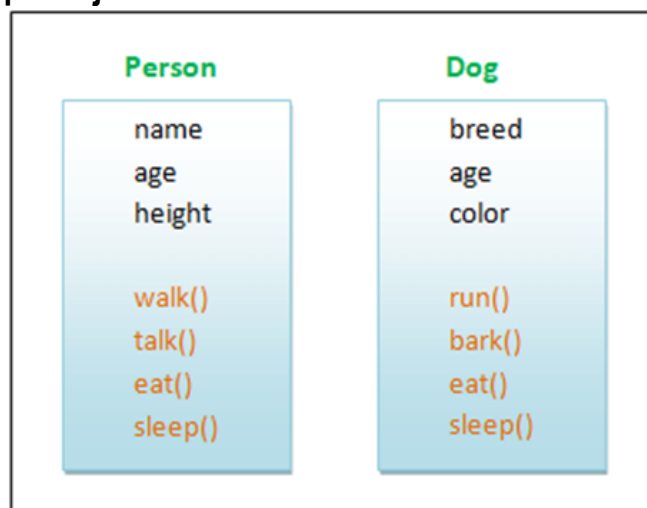
```
class ClassName
{
    // variables, constructors,
    // method declarations
}
```

Here **class** is a keyword, used to define a class in java. **ClassName** is the name of class, given by the programmer. After the class name, it's the **class body** given inside **{ }**. **Variables, methods, constructors** etc defined inside the **balanced { }** after the class name are the part of that class.

Do we have any convention to write the class name? The convention is that a class name and any subsequent word in class name must start with uppercase, e.g. MyFirstProgram, HelloWorldProgram, Person, Bicycle etc are valid **class names** of this convention. **Is class keyword case-sensitive?** All keywords in java are **case-sensitive** and they must be in **smallletters**. So, you can't use a **class** keyword as **Class, CLASS, cLass** etc, it must be in small letters. In **object oriented** world, think of a class as a **blueprint** or **design** describes how an **object** of a class behaves and what **data(properties)** it will contain inside it. The methods of a class define the behavior of object while instance variables are the data of the object that it will contain inside.

REAL TIME EXAMPLE OF CLASS

Some **class examples** are **Person, Vehicle, House, Tree, Dog** etc. For an E.g. a **Person class** has properties like **name, age, height** etc. and behaviors like **walk, talk, eat, sleep** etc. Now if you create an object of this class, it will have these **properties** and **behaviors** inside it. Once you created a class, you can create **multiple objects** of that class.



Real world examples of class

Can I give any name to my class? Yes, you can give any name to a class by following the identifier naming convention rules but giving a meaningful name is a good programming style. E.g. class names like `Abc`, `XYZ` are valid names but not meaningful names while class names like `MyFirstProgram`, `HelloWorldProgram`, `Person` are meaningful names.

JAVA CLASS PROGRAM

```
class Person
{
    // Instance variables, describes state/properties of object of this class.
    String name;
    int age;
    int height;
    // methods, describes behaviors of object of this class.
    public void walk() {
        System.out.println("Hi my name is : "+name+", age : "+age+" year,"
            +" height : "+height+" cm. I can Walk");
    }
    public void talk() {
        System.out.println("Hi my name is : "+name+", age : "+age+" year,"
            +" height : "+height+" cm. I can Talk");
    }
}

class ClassDemo
{
    public static void main(String [] args)
    {
        Person p1 = new Person(); // Creating object of class Person
        p1.name = "Rahul";
        p1.age = 20;
        p1.height = 170;
        p1.walk();
        p1.talk();
    }
}
```

In above example we have created a class called *Person* which is basically a prototype, describing that object of this class will have properties as *name*, *age*, *height* and will have behaviors as *walk* and *talk*. You can add more properties(instance variables) and behaviors(methods) in your program. This example creates only a single object of *Person* class. You can create multiple objects of *Person* class and assign different values inside it. **Should I create all classes in single file?** No, that's not a good programming style. In real world programming mostly, we use to create each class in separate files in our application. So next time when you are creating a class, think that you are creating a prototype which will tell that what *properties(state)* and *behaviors* will exist inside the object of that class. This way you will get better idea to design your class. A class is also called as a **blueprint** or a **template** or a **design** or a **Type**. These are just different words use to describe a class in java. **Who loads java**

classes in memory while execution? As soon as you run a program, the class-loaders available in java virtual machine loads classes in memory for execution. In java, these class-loaders are also a program. **Is it mandatory to define variables(instance or static variable) on top of a class?** No, you can define such variables after the method or in end of a class but that's not a good programming style. You should always prefer to declare it at top since it makes a class more readable. **Can a class exist without main method?** Yes, in real world programming only a starting class of your application(a group of classes) will have main method, all other classes won't have main method. **Should I focus more on real world aspect of class?** As a programmer you should focus more on it's programming aspect rather than focusing on real world aspect, programming aspects like creation of class, creation of variables and methods inside the class, creation of objects of class etc. The syntax of declaring a class given in this tutorial is the minimal one which is required. Some of the more keywords that can be declared along with `class` keyword are `public`, `abstract`, `final`, `extends`, `implements`. A class can also have *another class*, *static blocks*, *enum* etc inside the body of it.

NOTE: Every program must have at least one class, it can have more than one class as well. If there are multiple classes in a single program file, the file must be saved by class name declared with `public` keyword, if any. Every class(user defined or defined by java) in java is a non-primitive data type. There should be only one main method having argument type as `String []` in a class. If there are multiple non public classes in a single program file, conventionally the program file should be saved by class name having main method.

WHAT IS AN OBJECT IN JAVA

Objects are key concepts of java, since it is an **object-oriented programming** language. In **OOP**, program focuses on **objects** rather than **logic**. In the real world, you can see many examples of object around you: **person**, **dog**, **laptop**, **mobile**, **bicycle** etc. These **objects** have two features, **state** and **behavior**. E.g. a person has a **state** like *name*, *age*, *height* etc and **behavior** like *walk*, *talk*, *eat* etc while a dog have state like breed, age, color etc and behavior like run, bark, eat etc. Similarly, **software object** also has two **characteristics**: **state** and **behavior**. The *state* of an object is defined by the **instance variables** of a class while behavior is defined by the **methods** of class. So an **object in java is a set of data(instance variables) and methods that acts on those data**. In programming an **object is an instance of a class**. So next time when you are creating a class, think of two things about its objects. "**What possible states can this object have?**" and "**What possible behavior can this object perform?**". This way you will get better idea to design your class. You can think of a software object like below which holds its *state* and *behavior* together with itself.

The **Syntax** given below shows the most basic way of creating an object of a class in java. There are other ways as well but we will not discuss that here.

```
ClassName objName = new ClassName();
```

Here **ClassName** is the name of a class whose object needs to be created and **objName** is the name of object. The object name should be unique and should follow the convention given in identifier naming convention. A keyword `new` is used to create object in java. Using the () with *ClassName* calls the constructor of that class to create and initialize the object in memory. We will discuss about

constructors in later sections. Let's assume we have two classes *Person* and *MyFirstProgram*, code given below shows how to create an object of these classes:

```
// Creates an object of Person class with name as obj.
Person obj = new Person();

// Creates an object of MyFirstProgram class with name as mfp.
MyFirstProgram mfp = new MyFirstProgram();
```

How can I access properties and methods of an object? You just have to apply dot(.) operator with object name followed by property/method name to access the properties and methods of an object. You can also change the value of a property of an object using the same operator. Refer program given below to see the use of this operator. **What if I access object name only?** In java object name is a reference, so accessing the name will return the reference(address) of that object in memory.

OBJECT PROGRAM IN JAVA

```
class Person
{
    // Instance variables, describes state/properties of object of this class
    String name;
    int age;
    int height;
    // methods, describes behaviors of object of this class
    public void walk() {
        System.out.println("Hi my name is : "+name+", age : "+age+" year,"
            +" height : "+height+" cm. I can Walk");
    }
    public void talk() {
        System.out.println("Hi my name is : "+name+", age : "+age+" year,"
            +" height : "+height+" cm. I can Talk");
    }
    public void eat() {
        System.out.println("Hi my name is : "+name+", age : "+age+" year,"
            +" height : "+height+" cm. I can Eat");
    }
}

class ObjectDemo
{
    public static void main(String [] args)
    {
        // Creating object p1 of Person class
        Person p1 = new Person();
        p1.name = "Rahul";
        p1.age = 20;
        p1.height = 170;
        p1.walk();
    }
}
```

```

        p1.talk();
        p1.eat();
        // Creating another object p2 of Person class
        Person p2 = new Person();
        p2.name = "Rohit";
        p2.age = 30;
        p2.height = 180;
        p2.walk();
        p2.talk();
        p2.eat();
    }
}

```

Save above program as **ObjectDemo.java**
 compile as **javac ObjectDemo.java**
 run as **java ObjectDemo**

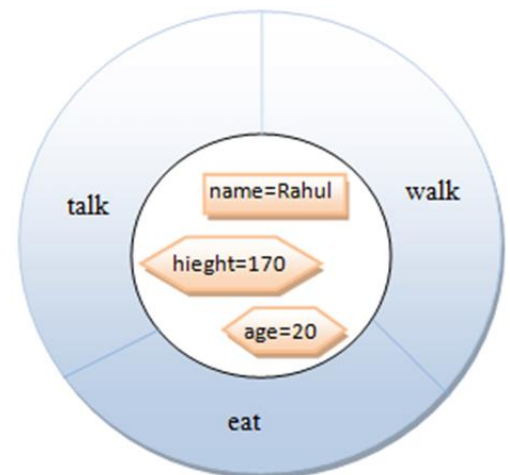
Output:

```

Hi my name is : Rahul, age : 20 year, height : 170 cm. I can Walk
Hi my name is : Rahul, age : 20 year, height : 170 cm. I can Talk
Hi my name is : Rahul, age : 20 year, height : 170 cm. I can Eat
Hi my name is : Rohit, age : 30 year, height : 180 cm. I can Walk
Hi my name is : Rohit, age : 30 year, height : 180 cm. I can Talk
Hi my name is : Rohit, age : 30 year, height : 180 cm. I can Eat

```

Here p1 and p2 are objects of *Person* class. Once the code line `Person p1 = new Person();` executed, java constructors creates the object p1 in heap memory and assigns the default values of the instance variables for this object. After that code lines `p1.name`, `p1.age` and `p1.height` modifies those default values of the instance variables for this object only. Similarly p2 is also created and assigned the values of instance variables. All the objects of a class are created inside heap memory. You can think of object p1 like below.



Who creates objects in java? It's the constructor that creates object of a class inside heap memory. We will see about constructors in later sections. At runtime each object that you have created in your program will be allocated space in heap memory. Each object will have it's own copy of instance variables and methods defined inside the class. Changing the value of a instance variable for one object will not change the value of that variable for other object. **Can an object exist without a class? No**, an object can never exist without a class. Every object must have a class which defines its type. **What happens to objects when program execution completed?**

As soon as the execution of a program completes, all its objects are destroyed/removed from memory. **Should I focus more on real world aspect of object?** You should focus more on its programming aspects rather than on real world aspects, programming aspects like creation of object, accessing its fields and methods, accessing object of other class etc.

★★★

Every object in java has a unique ID, that is not known to external programmers. JVM uses this id internally to identify each object uniquely. Word object and instance are used interchangeably. Fields, properties, attributes, state of an object refers to same thing which is generally instance variables of a class. An object can contain another object inside it. Only non-static variables and methods belong to an object of a class.

WHAT IS A METHOD IN JAVA

A method is a block of instructions(one or more line of codes) given in { } inside a class. This block is referred by a name which is basically the method name. The name of method is used to call(involve) that method within or outside a class. Once a method is called, the code given inside that method is executed. After execution a method may or may not return a value. The most basic or minimal **Syntax** of declaring a method is:

```
// Method without parameter
Return_Type methodName()
{
    // one or more line of code.
    return some_value;
}

// Method with parameter
Return_Type methodName(DataType param1, DataType param2 ...)
{
    // one or more line of code.
    return some_value;
}
```

A method must have a return type which basically tells what type of value this method returns. The **Return_Type** of a method can be **primitive or non-primitive data type**. If a method doesn't return a value, its return type must be **void** or in other way if the return type of a method is **void**, it means that method doesn't return any value. The name of the method is given by the programmer. As per the convention the method name should be in small letters while in multiWord method name, the first letter of any subsequent word should be in capital letter. For example method names like add(), calculateArea(), getName() etc are some examples of this convention. After method name, it's the parameters given inside () which are basically variables. Parameters are optional which means a method may or may not have parameters. You can access these parameters within the method only, not outside the method. The data type of parameters can be primitive or non-primitive. After parameters it's method body given inside { }. Everything given inside { } after method name are the part of that method. Let's see some examples of method declaration.

```
int add(int num1, int num2)
```



```

    {
        // one or more line of code.
        return value; // returns integer value.
    }
    void calculateArea()
    {
        // one or more line of code.
    }

```

Can I give any name to method? Yes you can give any name by following the [identifier naming convention](#) but giving a meaningful name is a good programming style. The name should itself suggest that what this method does. For example method names like xyz(), method123() are valid names but not meaningful names while names like add(), calculateArea(), getColor() are meaningful names.

BUILT-IN VS. USER DEFINED METHODS

Methods that are defined inside the classes included in java software are known as built-in methods. For example println() method, String class methods like charAt(), toLowerCase(), trim() etc are built-in methods. The methods that programmers defined inside their classes is known as user defined methods. It's programmer who gives the name to his method and code inside that method.

Can't I include all code or logic inside a single method? Why I should create different methods?

Yes you can include all logics or code inside a single method but that's not a good programming style. A good programmer always creates different methods for different tasks which makes your program more readable and modular.

PARAMETER VS ARGUMENT

Parameters are variables declared inside the () after a **method name**. Parameters can be accessed inside the method only. In method declaration add(int num1, int num2), variables num1 and num2 are parameters. Arguments are the values that are passed to method while calling the method. For example in a method call add(20,30), 20 and 30 are arguments.

STATIC AND NON STATIC METHOD

A method declared with **static** keyword is known as static method. Static method belongs to class not object which means you don't need to create object to access such methods. You can access static methods with class name itself. Methods defined without **static** keyword is known as non static method, also known as instance method. You need object or instance of the class to access such methods. For example in below program calculateArea() is a static method while add(), firstMethod(), secondMethod() are non static methods.

Static method can access only **static** variables inside its body while non static method can access both **static** and non static variables. We will discuss more about this in later tutorial.

JAVA METHOD PROGRAM

```

class MethodDemo
{
    public static void main(String [] args)
    {
        MethodDemo md = new MethodDemo();
        int sum = md.add(20,30); // calling add() method
        System.out.println("sum = "+sum);
    }
}

```

```

        md.firstMethod(); // calling firstMethod() using object
        MethodDemo.calculateArea(100,50); // calling calculateArea method
using className
    }
    int add(int num1, int num2)
    {
        int sum = num1 + num2;
        return sum;
    }
    void firstMethod()
    {
        System.out.println("Inside first method");
        secondMethod(); // calling secondMethod()
        System.out.println("After calling second method");
    }
    void secondMethod()
    {
        System.out.println("Inside second method");
    }
    static void calculateArea(int length, int width)
    {
        int area = length*width;
        System.out.println("Area = "+area);
    }
}

```

Output:

```

sum = 50
Inside first method
Inside second method
After calling second method
Area = 5000

```

As you can see **static** method `calculateArea()` is called using class name `MethodDemo`. You can call **static** method using object as well, but that's not a good practice, since **static** methods belongs to class not object. You should always prefer to call **static** methods using class name.

What is called method and calling method? A method that calls a given method is known as calling method while the method that is being called is known as called method. E.g. in above program, `main()` is the *calling* method for *called* methods `firstMethod()` and `calculateArea()`.

Can I pass object inside method calling? **Yes** you can pass objects as well inside the method. Any changes made in that object in called method will be reflected in calling method as well.

How method execution happens in Java? As soon as a method is called, the execution of that method get's started. Once the execution of method completed or any **return** keyword encountered, the execution control comes back to the position from where it was called. Once a method is called, java

creates a new stack inside stack memory where all local variables of that method are initialized. For example in above program once line `int sum = md.add(20,30);` executed, the execution of `add` method get's started. As soon as the `return` statement in `add` method is executed, the execution control again comes back to the line `int sum = md.add(20,30);` where the value returned by `add` method is assigned in variable `sum`.

What is method signature?

- The name and the parameters of a method in a method declaration is referred as method signature. Other components like access modifiers, return type etc are not the part of method signature. For example method signature of `add` method in above program is `add(int num1, int num2)`.
- You can define multiple methods with same name having different parameter lists. Java differentiates such methods on the basis of the number of parameters in the list and their types. This is known as method overloading. We will discuss method overloading in later tutorials.
- The syntax of declaring a method given in this tutorial is the minimal one which is required. Apart from this there are couple of more keywords as given below that can be used with method declaration. We will discuss these keywords with methods in later tutorials.
- An access modifier(`public`, `protected`, `private`) can also be used with method declaration. Access modifiers decides the visibility/accessibility of method within or outside the class. For example a `private` method can only be accessed within the class while `public` method can be accessed from outside the class as well.
- `final` keyword can also be used with method declaration. `final` method can not be overridden.
- `abstract` keyword can also be used with method declaration. The implementation or definition of abstract method is defined by sub class.
- A method can also declare an exception using `throws` keyword.



1. Accessing a method means calling that method.
2. Functions in other programming language are equivalent to methods in java programming.
3. There should be only one main method having argument type as `String []` in a class.
4. When you pass an object inside a method, the reference of that object is passed to the method.
5. Method that returns a boolean value, can be called inside a conditional statement as well like `if(IsValid())`, `if(isEmpty())` etc.

CONSTRUCTORS IN JAVA

Constructor in java is used in creation and initialization of object of a class inside memory. It's the constructor that provides the initial values of instance variables for an object inside memory. Constructors declarations are similar as method declarations except that constructors don't have any return type and their name is same as class name. The **syntax** of declaring a constructor is :

```

class MyClassName {

    // No-argument constructor
    Access_Modifier MyClassName() {
        // Initialization code
    }

    // Parameterized constructor
    Access_Modifier MyClassName(DataType param1, DataType param2 ...) {
        // Initialization code
    }
}

```

Example :

```

class Person {
    Person() {
        // Initialization code
    }

    public Person(int param1, String param2) {
        // Initialization code
    }
}

```

Access modifier of a constructor can be **public**, **private**, **protected** or default(no modifier). These modifiers with constructors decides the accessibility(visibility) of the constructor within or outside the class. After access modifier it's the name of constructor which must be same as the class name.

Parameters in constructors are **optional**, a constructor may or may not have parameters. A constructor without parameter is also known as **no-argument** constructor while a constructor with parameter is also known as **parameterized** constructor. The data type of parameters can be primitive or non-primitive. In constructor body generally we initialize instance variables with some values but it's not limited to that only. You can use other statements as well as you do in methods. You can define multiple constructors with different argument lists. Java differentiates constructors on the basis of the number of arguments in the list and their types. This is also known as **constructor overloading**. You cannot write two constructors that have the same number and type of arguments for the same class, because java compiler would not be able to differentiate them. In this case compiler will throw error.

Types of Constructors in Java

Constructor can be divided in three types.

1.No-argument constructor 2.Parameterized constructor 3.Default constructor

We have already covered the definition of first two constructor. We will see how to use them in program example given below. **How constructors are invoked(called)?** Constructors are invoked automatically when the code that creates object of a class using **new** keyword is executed.

CONSTRUCTOR PROGRAM IN JAVA

```

class Person
{
    int age;
    int height;
    String name;
    // No-argument constructor
    public Person() {
        System.out.println("Initialization using no-argument constructor");
        age = 15;
        height = 160;
        name = "Pradeep";
    }
    // Parameterized constructor
    public Person(int a, int h, String n) {
        System.out.println("Initialization using parameterized constructor");
        age = a;
        height = h;
        name = n;
    }
    public static void main(String [] args) {
        Person p1 = new Person(); // invokes no-argument constructor
        p1.print();
        Person p2 = new Person(20,170,"Rahul"); // invokes parameterized constructor
        p2.print();
    }
    void print() {
        System.out.println("age = "+age+", height = "+height+", name = "+name);
    }
}

```

HOW CONSTRUCTOR WORKS IN JAVA

Let's understand this by above program. Once the code `new Person();` is executed, it creates the object of Person class inside memory and invokes the no-argument constructor which executes the given print statement and assigns values of instance variables for this object. Finally the reference of this object is assigned into p1. Similarly code `new Person(20,170,"Rahul");` creates another object inside memory and invokes the matching parameterized constructor which assigns values passed to it in corresponding instance variables for object p2. So both objects will have their specific values of instance variables. You can imagine the object p2 inside memory like below.