www.beginnersbook.com

www.javatpoint.com

www.studytonight.com

www.w3schools.com

# Java Tutorial

Our core Java programming tutorial is designed for students and working professionals. Java is an object-oriented, class-based, concurrent, secured and general-purpose computer-programming language. It is a widely used robust technology.

## What is Java

Java is a **programming language** and a **platform**.
Java is a high level, robust, object-oriented and secure programming language.
**Platform**: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

---

## Java Example

Let's have a quick look at Java programming example. A detailed description of hello Java example is available in next page.

```
1. class Simple{
2.    public static void main(String args[]){
3.     System.out.println("Hello Java");
4.    }
5. }
```

---

## Application

According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games, etc.

---

## Types of Java Applications

There are mainly 4 types of applications that can be created using Java programming:

### 1) Standalone Application

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

### 2) Web Application

An application that runs on the server side and creates a dynamic page is called a web application. Currently, Servlet, JSP, Struts, Spring, Hibernate, JSF, etc. technologies are used for creating web applications in Java.

### 3) Enterprise Application

An application that is distributed in nature, such as banking applications, etc. is called enterprise application. It has advantages of the high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.

### 4) Mobile Application

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

## Java Platforms / Editions

There are 4 platforms or editions of Java:

### 1) Java SE (Java Standard Edition)

It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, String, Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

### 2) Java EE (Java Enterprise Edition)

It is an enterprise platform which is mainly used to develop web and enterprise applications. It is built on the top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, JPA, etc.

### 3) Java ME (Java Micro Edition)

It is a micro platform which is mainly used to develop mobile applications.

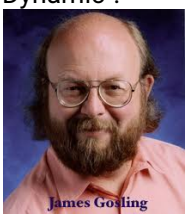
### 4) JavaFX

It is used to develop rich internet applications. It uses a light-weight user interface API.

## History of Java

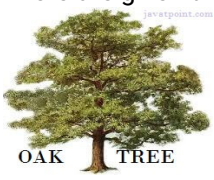
**The history of Java** is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of java starts with Green Team. Java team members (also known as **Green Team**), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was suited for internet programming. Later, Java technology was incorporated by Netscape.

The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted and Dynamic".




Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc.

There are given the significant points that describe the history of Java.




1) **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

2) Originally designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called **"Greentalk"** by James Gosling, and file extension was .gt.

4) After that, it was called **Oak** and was developed as a part of the Green project.

### Why Java named "Oak"?

5) **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like U.S.A., France, Germany, Romania, etc.

6) In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

### Why Java Programming named "Java"?

7) **Why had they chosen java name for java language?** The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA", etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say.

According to James Gosling, "Java was one of the top choices along with **Silk**". Since Java was so unique, most of the team members preferred Java than other names.

8) Java is an island of Indonesia where first coffee was produced (called java coffee).

9) Notice that Java is just a name, not an acronym.

10) Initially developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

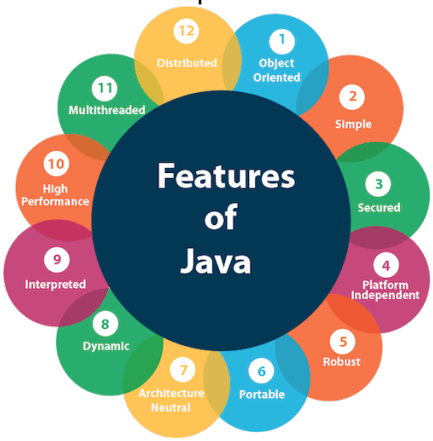
11) In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

12) JDK 1.0 released in(January 23, 1996).

### Features of Java

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as java *buzzwords*.

A list of most important features of Java language is given below.




1. Simple
2. Object-Oriented
3. Portable

4. Platform independent
5. Secured
6. Robust
7. Architecture neutral
8. Interpreted
9. High Performance
10. Multithreaded
11. Distributed
12. Dynamic

### Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun, Java language is a simple programming language because:

- Java syntax is based on C++ (so easier for programmers to learn it after C++).
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

---

### Object-oriented

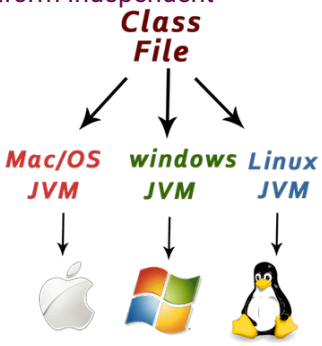
Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

### Platform Independent



Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

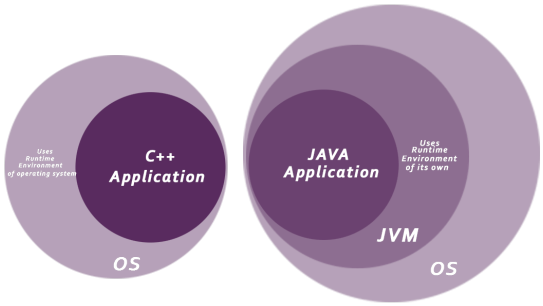
Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere(WORA).

---

## Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- **No explicit pointer**
- **Java Programs run inside a virtual machine sandbox**




- **Classloader:** Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an

application developer explicitly through SSL, JAAS, Cryptography, etc.

---

### Robust

Robust simply means strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

---

### Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

---

### Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

---

### High-performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

---

### Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

---

### Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

### Dynamic

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

### First Java Program | Hello World Example

In this page, we will learn how to write the simple program of java. We can write a simple hello java program easily after

To create a simple java program, you need to create a class that contains the main method. Let's understand the requir

### The requirement for Java Hello World Example

For executing any java program, you need to

- Install the JDK if you don't have installed it, download the JDK and install it.
- Set path of the jdk/bin directory. http://www.javatpoint.com/how-to-set-path-in-java
- Create the java program
- Compile and run the java program

---

### Creating Hello World Example

Let's create the hello java program:

```
1. class Simple{
2.    public static void main(String args[]){
3.     System.out.println("Hello Java");
4.    }
5. }
```

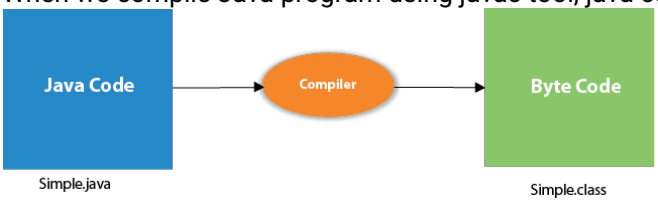
save this file as Simple.java

| | |
|---|---|
| **To compile:** | javac Simple.java |
| **To execute:** | java Simple |

**Output:**Hello Java

**Compilation Flow:**

When we compile Java program using javac tool, java compiler converts the source code into byte code.




### Parameters used in First Java Program

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

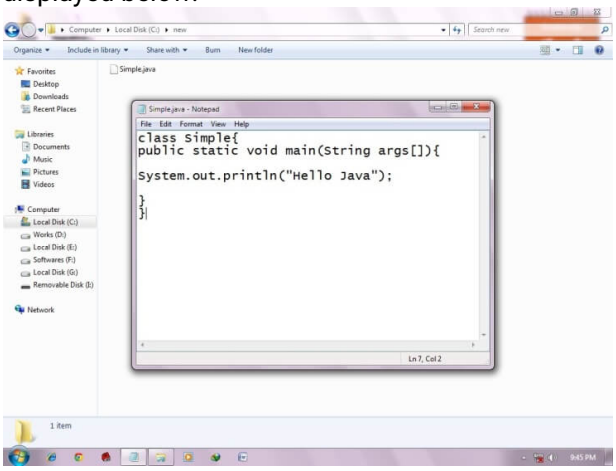
- **class** keyword is used to declare a class in java.
- **public** keyword is an access modifier which represents visibility. It means it is visible to all.
- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of

there is no need to create an object to invoke the static method. The main method is executed by the JVM, so i object to invoke the main method. So it saves memory.

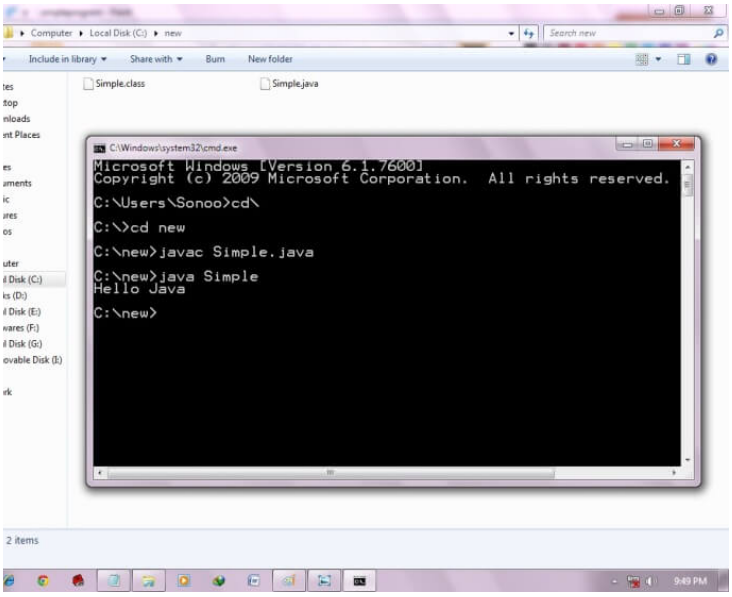
- **void** is the return type of the method. It means it doesn't return any value.
- **main** represents the starting point of the program.
- **String[] args** is used for command line argument. We will learn it later.
- **System.out.println()** is used to print statement. Here, System is a class, out is the object of PrintStream class, p PrintStream class. We will learn about the internal working of System.out.println statement later.

---

To write the simple program, you need to open notepad by **start menu -> All Programs -> Accessories -> notepad** and w displayed below:




As displayed in the above diagram, write the simple program of java in notepad and saved it as Simple.java. To com you need to open the command prompt by **start menu -> All Programs -> Accessories -> command prompt**.



To compile and run the above program, go to your current directory first; my current directory is c:\new. Write here:

| | |
|---|---|
| **To compile:** | javac Simple.java |
| **To execute:** | java Simple |

---

How many ways can we write a Java program

There are many ways to write a Java program. The modifications that can be done in a Java program are given below:

**1) By changing the sequence of the modifiers, method prototype is not changed in Java.**

Let's see the simple code of the main method.

```
1. static public void main(String args[])
```

**2) The subscript notation in Java array can be used after type, before the variable or after the variable.**

Let's see the different codes to write the main method.

```
1. public static void main(String[] args)
2. public static void main(String []args)
3. public static void main(String args[])
```

**3) You can provide var-args support to the main method by passing 3 ellipses (dots)**

Let's see the simple code of using var-args in the main method. We will learn about var-args later in Java New Features

```
1. public static void main(String... args)
```

**4) Having a semicolon at the end of class is optional in Java.**

Let's see the simple code.

```
1. class A{
2. static public void main(String... args){
3. System.out.println("hello java4");
4. }
5. };
```

---

Valid java main method signature

```
1. public static void main(String[] args)
2. public static void main(String []args)
3. public static void main(String args[])
4. public static void main(String... args)
5. static public void main(String[] args)
6. public static final void main(String[] args)
7. final public static void main(String[] args)
8. final strictfp public static void main(String[] args)
```

---

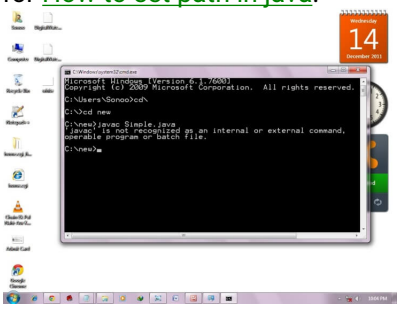
Invalid java main method signature

```
1. public void main(String[] args)
```

2. **static void** main(String[] args)
3. **public void static** main(String[] args)
4. **abstract public static void** main(String[] args)

### Resolving an error "javac is not recognized as an internal or external command"?

If there occurs a problem like displayed in the below figure, you need to set path. Since DOS doesn't know javac or java, is not required in such a case if you save your program inside the JDK/bin directory. However, it is an excellent approac for How to set path in java.

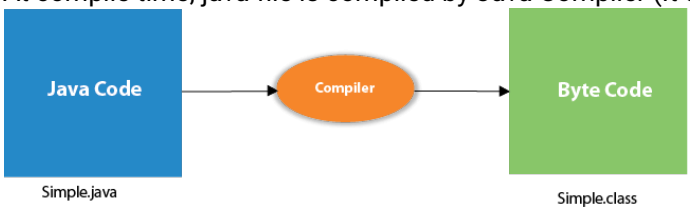



### Internal Details of Hello Java Program

In the previous page, we have learnt about the first program, how to compile and run the first java program. Here, we ar happens while compiling and running the java program. Moreover, we will see some question based on the first progra

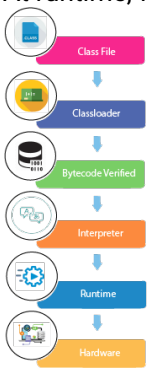
### What happens at compile time?

At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into by




### What happens at runtime?

At runtime, following steps are performed:



**Classloader:** is the subsystem of JVM that is used to load class files.

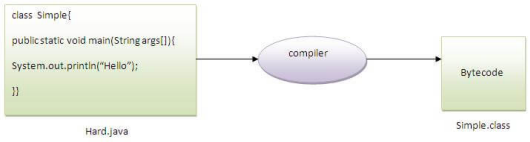
**Bytecode Verifier:** checks the code fragments for illegal code that can violate access right to objects.

**Interpreter:** read bytecode stream then execute the instructions.

---

Q) Can you save a java source file by other name than the class name?

Yes, if the class is not public. It is explained in the figure given below:

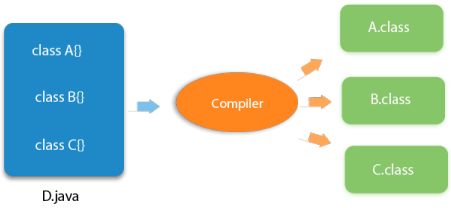



| | |
|---|---|
| **To compile:** | javac Hard.java |
| **To execute:** | java Simple |

---

Q) Can you have multiple classes in a java source file?

Yes, like the figure given below illustrates:




## How to set path in Java

The path is required to be set for using tools such as javac, java, etc.
If you are saving the Java source file inside the JDK/bin directory, the path is not required to be set because all the tools will be available in the current directory.
However, if you have your Java file outside the JDK/bin folder, it is necessary to set the path of JDK.
There are two ways to set the path in Java:

1. Temporary
2. Permanent

### 1) How to set the Temporary Path of JDK in Windows

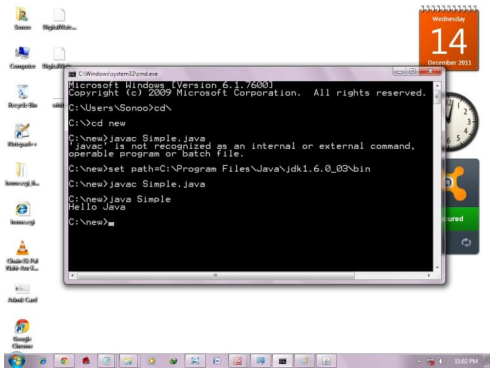
To set the temporary path of JDK, you need to follow the following steps:

- Open the command prompt
- Copy the path of the JDK/bin directory
- Write in command prompt: set path=copied_path

For Example:

set path=C:\Program Files\Java\jdk1.6.0_23\bin

Let's see it in the figure given below:




## 2) How to set Permanent Path of JDK in Windows

For setting the permanent path of JDK, you need to follow these steps:

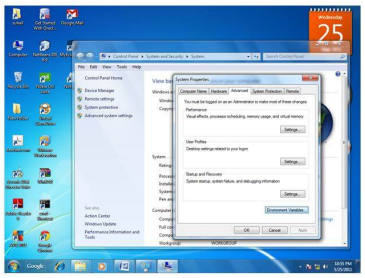
- Go to MyComputer properties -> advanced tab -> environment variables -> new tab of user variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok

For Example:

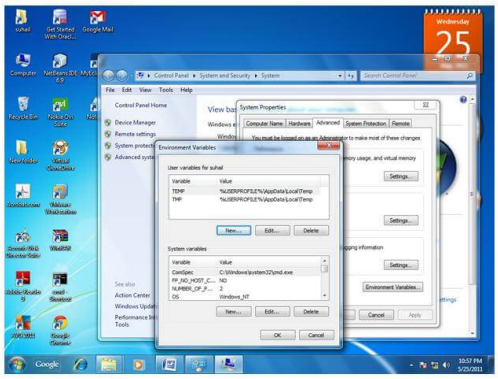
### 1) Go to MyComputer properties

### 2) Click on the advanced tab

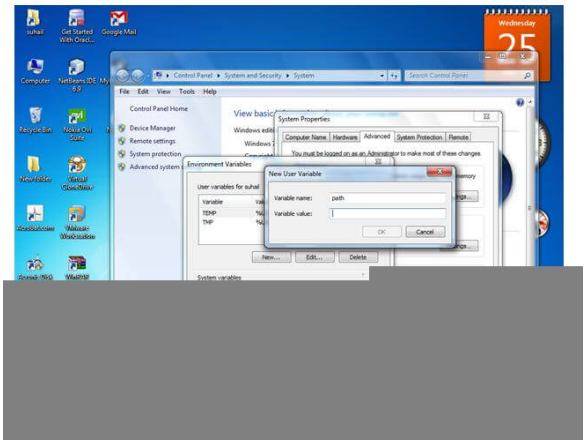
### 3) Click on environment variables



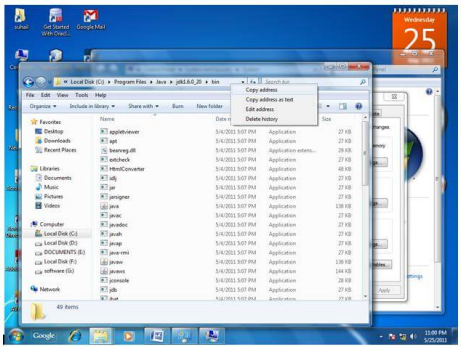
### 4) Click on the new tab of user variables



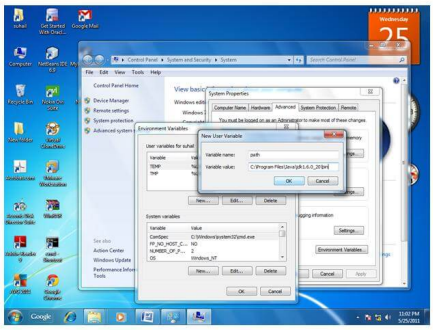
### 5) Write the path in the variable name



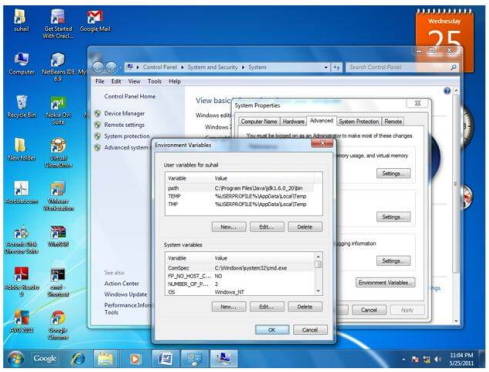
## 6) Copy the path of bin folder



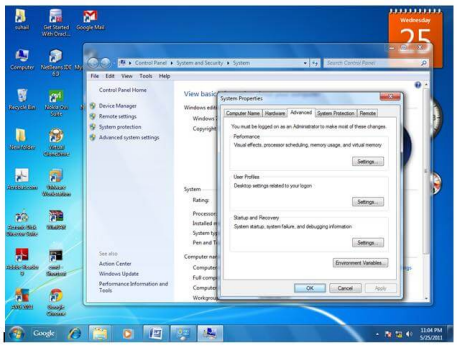
## 7) Paste path of bin folder in the variable value



## 8) Click on ok button



### 9) Click on ok button



Now your permanent path is set. You can now execute any program of java from any drive.

---

## Setting Java Path in Linux OS

Setting path in Linux OS is the same as setting the path in the Windows OS. But, here we use the export tool rather than set. Let's see how to set path in Linux OS:

```
export PATH=$PATH:/home/jdk1.6.01/bin/
```

Here, we have installed the JDK in the home directory under Root (/home).

## Difference between JDK, JRE, and JVM

We must understand the differences between JDK, JRE, and JVM before proceeding further to Java. See the brief overview of JVM here.

If you want to get the detailed knowledge of Java Virtual Machine, move to the next page. Firstly, let's see the differences between the JDK, JRE, and JVM.

---

## JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.

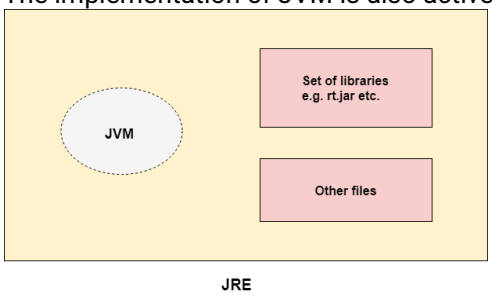
The JVM performs the following main tasks:

- Loads code

- Verifies code
- Executes code
- Provides runtime environment

## JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.
The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



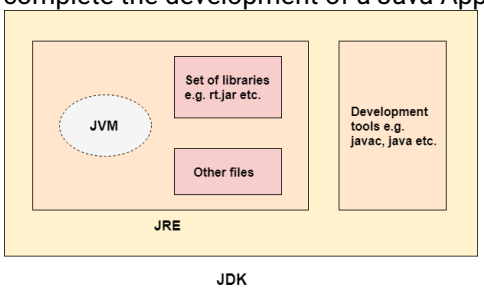

---

## JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.
JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.




## JVM (Java Virtual Machine) Architecture

1. Java Virtual Machine
2. Internal Architecture of JVM

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.
JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

## What is JVM

It is:

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

## What it does

The JVM performs following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

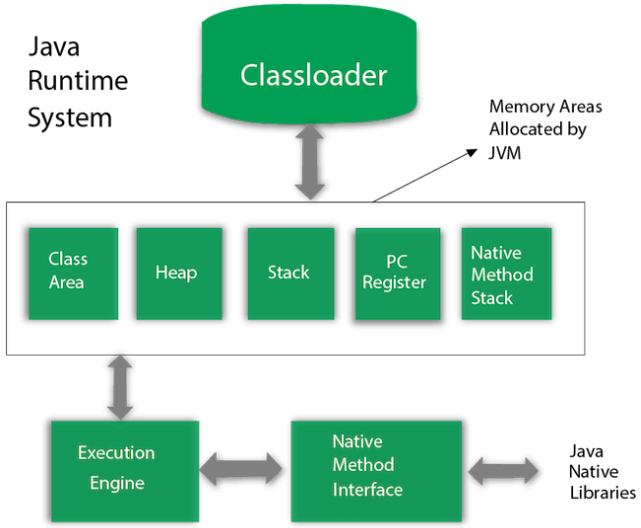
JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

---

## JVM Architecture

Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.




## 1) Classloader

Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

1. **Bootstrap ClassLoader**: This is the first classloader which is the super class of Extension

classloader. It loads the *rt.jar* file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.

2. **Extension ClassLoader**: This is the child classloader of Bootstrap and parent classloader of System classloader. It loades the jar files located inside *$JAVA_HOME/jre/lib/ext* directory.
3. **System/Application ClassLoader**: This is the child classloader of Extension classloader. It loads the classfiles from classpath. By default, classpath is set to current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

```
1.  //Let's see an example to print the classloader name
2.  public class ClassLoaderExample
3.  {
4.     public static void main(String[] args)
5.     {
6.        // Let's print the classloader name of current class.
7.        //Application/System classloader will load this class
8.        Class c=ClassLoaderExample.class;
9.        System.out.println(c.getClassLoader());
10.       //If we print the classloader name of String, it will print null because it is an
11.       //in-built class which is found in rt.jar, so it is loaded by Bootstrap classloader
12.       System.out.println(String.class.getClassLoader());
13.    }
14. }
```

Output:

```
sun.misc.Launcher$AppClassLoader@4e0e2f2a
null
```

These are the internal classloaders provided by Java. If you want to create your own classloader, you need to extend the ClassLoader class.

### 2) Class(Method) Area

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

### 3) Heap

It is the runtime data area in which objects are allocated.

### 4) Stack

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.
Each thread has a private JVM stack, created at the same time as thread.
A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

### 5) Program Counter Register

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

### 6) Native Method Stack

It contains all the native methods used in the application.

### 7) Execution Engine

It contains:

1. **A virtual processor**
2. **Interpreter:** Read bytecode stream then execute the instructions.
3. **Just-In-Time(JIT) compiler:** It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

### 8) Java Native Interface

Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

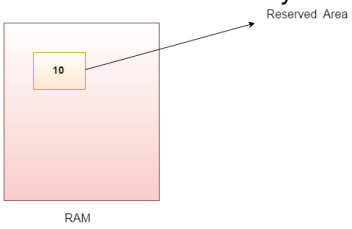
## Java Variables

A variable is a container which holds the value while the java program is executed. A variable is assigned with a datatype.
Variable is a name of memory location. There are three types of variables in java: local, instance and static.
There are two types of data types in java: primitive and non-primitive.

---

### Variable

**Variable** is name of *reserved area allocated in memory*. In other words, it is a *name of memory location*. It is a combination of "vary + able" that means its value can be changed.

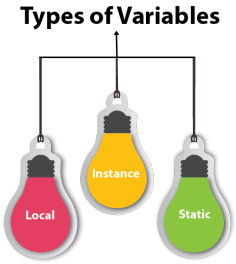



1. `int data=50;//Here data is variable`

### Types of Variables

There are three types of variables in java:

- local variable
- instance variable
- static variable




### 1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.
A local variable cannot be defined with "static" keyword.

### 2) Instance Variable

A variable declared inside the class but outside the body of the method, is called instance variable. It is not declared as static.

It is called instance variable because its value is instance specific and is not shared among instances.

### 3) Static variable

A variable which is declared as static is called static variable. It cannot be local. You can create a single copy of static variable and share among all the instances of the class. Memory allocation for static variable happens only once when the class is loaded in the memory.

### Example to understand the types of variables in java

```
1. class A{
2. int data=50;//instance variable
3. static int m=100;//static variable
4. void method(){
5. int n=90;//local variable
6. }
7. }//end of class
```

### Java Variable Example: Add Two Numbers

```
1. class Simple{
2. public static void main(String[] args){
3. int a=10;
4. int b=10;
5. int c=a+b;
6. System.out.println(c);
7. }}
```

Output:

```
20
```

### Java Variable Example: Widening

```
1. class Simple{
2. public static void main(String[] args){
3. int a=10;
4. float f=a;
5. System.out.println(a);
6. System.out.println(f);
7. }}
```

Output:

```
10
10.0
```

### Java Variable Example: Narrowing (Typecasting)

```
1. class Simple{
2. public static void main(String[] args){
3. float f=10.5f;
4. //int a=f;//Compile time error
5. int a=(int)f;
6. System.out.println(f);
7. System.out.println(a);
8. }}
```

Output:

```
10.5
10
```

Java Variable Example: Overflow

```
1. class Simple{
2. public static void main(String[] args){
3. //Overflow
4. int a=130;
5. byte b=(byte)a;
6. System.out.println(a);
7. System.out.println(b);
8. }}
```

Output:

```
130
-126
```

Java Variable Example: Adding Lower Type

```
1. class Simple{
2. public static void main(String[] args){
3. byte a=10;
4. byte b=10;
5. //byte c=a+b;//Compile Time Error: because a+b=20 will be int
6. byte c=(byte)(a+b);
7. System.out.println(c);
8. }}
```

Output:

```
20
```

## Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

## Java Primitive Data Types

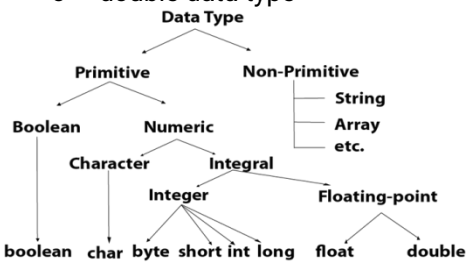
In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

> Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name.

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type

- o int data type
- o long data type
- o float data type
- o double data type




| Data Type | Default Value | Default size |
|---|---|---|
| boolean | False | 1 bit |
| char | '\u0000' | 2 byte |
| byte | 0 | 1 byte |
| short | 0 | 2 byte |
| int | 0 | 4 byte |
| long | 0L | 8 byte |
| float | 0.0f | 4 byte |
| double | 0.0d | 8 byte |

### Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.
The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.
**Example:** Boolean one = false

### Byte Data Type

The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.
The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.
**Example:** byte a = 10, byte b = -20

### Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.
The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.
**Example:** short s = 10000, short r = -5000

### Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.
The int data type is generally used as a default data type for integral values unless if there is no problem about memory.
**Example:** int a = 100000, int b = -200000

### Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between - 9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.
**Example:** long a = 100000L, long b = -200000L

### Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.
**Example:** float f1 = 234.5f

### Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.
**Example:** double d1 = 12.3

### Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.
**Example:** char letterA = 'A'

---

### Why char uses 2 byte in java and what is \u0000 ?

It is because java uses Unicode system not ASCII code system. The \u0000 is the lowest range of Unicode system. To get detail explanation about Unicode visit next page.

### Unicode System

Unicode is a universal international standard character encoding that is capable of representing most of the world's wri

### Why java uses Unicode System?

Before Unicode, there were many language standards:

- **ASCII** (American Standard Code for Information Interchange) for the United States.
- **ISO 8859-1** for Western European Language.
- **KOI-8** for Russian.
- **GB18030 and BIG-5** for chinese, and so on.

### Problem

**This caused two problems:**

1. A particular code value corresponds to different letters in the various language standards.
2. The encodings for languages with large character sets have variable length.Some common characters are enc other require two or more byte.

### Solution

To solve these problems, a new language standard was developed i.e. Unicode System.

In unicode, character holds 2 byte, so java also uses 2 byte for characters.

**lowest value:**\u0000

**highest value:**\uFFFF

### Operators in java

**Operator** in java is a symbol that is used to perform operations. For example: +, -, *, / etc.

There are many types of operators in java which are given below:

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

### Java Operator Precedence

| Operator Type | Category | Precedence |
|---|---|---|
| Unary | postfix | *expr*++ *expr*-- |
| | prefix | ++*expr* --*expr* +*expr* -*expr* ~ ! |
| Arithmetic | multiplicative | * / % |
| | additive | + - |
| Shift | shift | << >> >>> |
| Relational | comparison | < > <= >= instanceof |
| | equality | == != |
| Bitwise | bitwise AND | & |

| | | |
|---|---|---|
| | bitwise exclusive OR | ^ |
| | bitwise inclusive OR | \| |
| Logical | logical AND | && |
| | logical OR | \|\| |
| Ternary | ternary | ? : |
| Assignment | assignment | = += -= *= /= %= &= ^= \|= <<= >>= >>>= |

Java Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

Java Unary Operator Example: ++ and --

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int x=10;
4. System.out.println(x++);//10 (11)
5. System.out.println(++x);//12
6. System.out.println(x--);//12 (11)
7. System.out.println(--x);//10
8. }}
```

Output:

```
10
12
12
10
```

Java Unary Operator Example 2: ++ and --

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=10;
5. System.out.println(a++ + ++a);//10+12=22
```

```
System.out.println(b++ + b++);//10+11=21

}}
```

Output:

```
22
21
```

Java Unary Operator Example: ~ and !

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=-10;
boolean c=true;
boolean d=false;
System.out.println(~a);//-11 (minus of total positive value which starts from 0)
System.out.println(~b);//9 (positive of total minus, positive starts from 0)
System.out.println(!c);//false (opposite of boolean value)
System.out.println(!d);//true
}}
```

Output:

```
-11
9
false
true
```

Java Arithmetic Operators

Java arithmatic operators are used to perform addition, subtraction, multiplication, and division. They act as basic math

Java Arithmetic Operator Example

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

Output:

```
15
5
```

```
50
2
0
```

Java Arithmetic Operator Example: Expression

```
class OperatorExample{
public static void main(String args[]){
System.out.println(10*10/5+3-1*4/2);
}}
```

Output:

```
21
```

Java Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Java Left Shift Operator Example

```
class OperatorExample{
public static void main(String args[]){
System.out.println(10<<2);//10*2^2=10*4=40
System.out.println(10<<3);//10*2^3=10*8=80
System.out.println(20<<2);//20*2^2=20*4=80
System.out.println(15<<4);//15*2^4=15*16=240
}}
```

Output:

```
40
80
80
240
```

Java Right Shift Operator

The Java right shift operator >> is used to move left operands value to right by the number of bits specified by the right

Java Right Shift Operator Example

```
class OperatorExample{
public static void main(String args[]){
System.out.println(10>>2);//10/2^2=10/4=2
System.out.println(20>>2);//20/2^2=20/4=5
System.out.println(20>>3);//20/2^3=20/8=2
}}
```

Output:

```
2
5
2
```

Java Shift Operator Example: >> vs >>>

```
class OperatorExample{
public static void main(String args[]){
   //For positive number, >> and >>> works same
   System.out.println(20>>2);
   System.out.println(20>>>2);
   //For negative number, >>> changes parity bit (MSB) to 0
   System.out.println(-20>>2);
   System.out.println(-20>>>2);
}}
```

Output:

```
5
5
-5
1073741819
```

Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check second condition if first condition is false. It checks second condition only if first

The bitwise & operator always checks both conditions whether first condition is true or false.

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a<c);//false && true = false
System.out.println(a<b&a<c);//false & true = false
}}
```

Output:

```
false
false
```

Java AND Operator Example: Logical && vs Bitwise &

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a++<c);//false && true = false
System.out.println(a);//10 because second condition is not checked
System.out.println(a<b&a++<c);//false && true = false
System.out.println(a);//11 because second condition is checked
}}
```

Output:

```
false
10
false
11
```

### Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check second condition if first condition is true. It checks second condition only if first or

The bitwise | operator always checks both conditions whether first condition is true or false.

```
1.  class OperatorExample{
2.  public static void main(String args[]){
3.  int a=10;
4.  int b=5;
5.  int c=20;
6.  System.out.println(a>b||a<c);//true || true = true
7.  System.out.println(a>b|a<c);//true | true = true
8.  //|| vs |
9.  System.out.println(a>b||a++<c);//true || true = true
10. System.out.println(a);//10 because second condition is not checked
11. System.out.println(a>b|a++<c);//true | true = true
12. System.out.println(a);//11 because second condition is checked
13. }}
```

Output:

```
true
true
true
10
true
11
```

### Java Ternary Operator

Java Ternary operator is used as one liner replacement for if-then-else statement and used a lot in java programming. i

operator which takes three operands.

### Java Ternary Operator Example

```
1.  class OperatorExample{
2.  public static void main(String args[]){
3.  int a=2;
4.  int b=5;
5.  int min=(a<b)?a:b;
6.  System.out.println(min);
7.  }}
```

Output:

```
2
```

Another Example:

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

```
5
```

## Java Assignment Operator

Java assignment operator is one of the most common operator. It is used to assign the value on its right to the operan

### Java Assignment Operator Example

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4;//a=a+4 (a=10+4)
b-=4;//b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}}
```

Output:

```
14
16
```

### Java Assignment Operator Example

```
class OperatorExample{
public static void main(String[] args){
int a=10;
a+=3;//10+3
System.out.println(a);
a-=4;//13-4
System.out.println(a);
a*=2;//9*2
System.out.println(a);
a/=2;//18/2
System.out.println(a);
}}
```

Output:

```
13
9
18
9
```

Java Assignment Operator Example: Adding short

```
class OperatorExample{
public static void main(String args[]){
short a=10;
short b=10;
//a+=b;//a=a+b internally so fine
a=a+b;//Compile time error because 10+10=20 now int
System.out.println(a);
}}
```

Output:

```
Compile time error
```

After type cast:

```
class OperatorExample{
public static void main(String args[]){
short a=10;
short b=10;
a=(short)(a+b);//20 which is int now converted to short
System.out.println(a);
}}
```

Output:

```
20
```

## Java Keywords

1. **abstract:** Java abstract keyword is used to declare abstract class. Abstract class can provide the implementation of interface. It can have abstract and non-abstract methods.
2. **boolean:** Java boolean keyword is used to declare a variable as a boolean type. It can hold True and False values only.
3. **break:** Java break keyword is used to break loop or switch statement. It breaks the current flow of the program at specified condition.
4. **byte:** Java byte keyword is used to declare a variable that can hold an 8-bit data values.
5. **case:** Java case keyword is used to with the switch statements to mark blocks of text.
6. **catch:** Java catch keyword is used to catch the exceptions generated by try statements. It must be used after the try block only.
7. **char:** Java char keyword is used to declare a variable that can hold unsigned 16-bit Unicode

characters

8. **class:** Java class keyword is used to declare a class.
9. **continue:** Java continue keyword is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition.
10. **default:** Java default keyword is used to specify the default block of code in a switch statement.
11. **do:** Java do keyword is used in control statement to declare a loop. It can iterate a part of the program several times.
12. **double:** Java double keyword is used to declare a variable that can hold a 64-bit floating-point numbers.
13. **else:** Java else keyword is used to indicate the alternative branches in an if statement.
14. **enum:** Java enum keyword is used to define a fixed set of constants. Enum constructors are always private or default.
15. **extends:** Java extends keyword is used to indicate that a class is derived from another class or interface.
16. **final:** Java final keyword is used to indicate that a variable holds a constant value. It is applied with a variable. It is used to restrict the user.
17. **finally:** Java finally keyword indicates a block of code in a try-catch structure. This block is always executed whether exception is handled or not.
18. **float:** Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.
19. **for:** Java for keyword is used to start a for loop. It is used to execute a set of instructions/functions repeatedly when some conditions become true. If the number of iteration is fixed, it is recommended to use for loop.
20. **if:** Java if keyword tests the condition. It executes the if block if condition is true.
21. **implements:** Java implements keyword is used to implement an interface.
22. **import:** Java import keyword makes classes and interfaces available and accessible to the current source code.
23. **instanceof:** Java instanceof keyword is used to test whether the object is an instance of the specified class or implements an interface.
24. **int:** Java int keyword is used to declare a variable that can hold a 32-bit signed integer.
25. **interface:** Java interface keyword is used to declare an interface. It can have only abstract methods.
26. **long:** Java long keyword is used to declare a variable that can hold a 64-bit integer.
27. **native:** Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface).
28. **new:** Java new keyword is used to create new objects.
29. **null:** Java null keyword is used to indicate that a reference does not refer to anything. It removes the garbage value.
30. **package:** Java package keyword is used to declare a Java package that includes the classes.
31. **private:** Java private keyword is an access modifier. It is used to indicate that a method or variable may be accessed only in the class in which it is declared.
32. **protected:** Java protected keyword is an access modifier. It can be accessible within package and

outside the package but through inheritance only. It can't be applied on the class.

33. **public:** Java public keyword is an access modifier. It is used to indicate that an item is accessible anywhere. It has the widest scope among all other modifiers.
34. **return:** Java return keyword is used to return from a method when its execution is complete.
35. **short:** Java short keyword is used to declare a variable that can hold a 16-bit integer.
36. **static:** Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is used for memory management mainly.
37. **strictfp:** Java strictfp is used to restrict the floating-point calculations to ensure portability.
38. **super:** Java super keyword is a reference variable that is used to refer parent class object. It can be used to invoke immediate parent class method.
39. **switch:** The Java switch keyword contains a switch statement that executes code based on test value. The switch statement tests the equality of a variable against multiple values.
40. **synchronized:** Java synchronized keyword is used to specify the critical sections or methods in multithreaded code.
41. **this:** Java this keyword can be used to refer the current object in a method or constructor.
42. **throw:** The Java throw keyword is used to explicitly throw an exception. The throw keyword is mainly used to throw custom exception. It is followed by an instance.
43. **throws:** The Java throws keyword is used to declare an exception. Checked exception can be propagated with throws.
44. **transient:** Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.
45. **try:** Java try keyword is used to start a block of code that will be tested for exceptions. The try block must be followed by either catch or finally block.
46. **void:** Java void keyword is used to specify that a method does not have a return value.
47. **volatile:** Java volatile keyword is used to indicate that a variable may change asynchronously.
48. **while:** Java while keyword is used to start a while loop. This loop iterates a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

### Java If-else Statement

The Java *if statement* is used to test the condition. It checks boolean condition: *true* or *false*. There are various types of if statement in java.

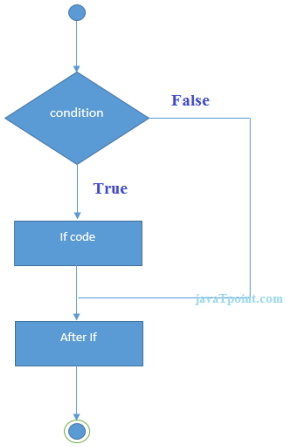
- if statement
- if-else statement
- if-else-if ladder
- nested if statement

### Java if Statement

The Java if statement tests the condition. It executes the *if block* if condition is true.

**Syntax:**

```
1. if(condition){
2. //code to be executed
3. }
```




**Example:**

```
1.  //Java Program to demonstate the use of if statement.
2.  public class IfExample {
3.  public static void main(String[] args) {
4.     //defining an 'age' variable
5.     int age=20;
6.     //checking the age
7.     if(age>18){
8.        System.out.print("Age is greater than 18");
9.     }
10. }
11. }
```

Output:

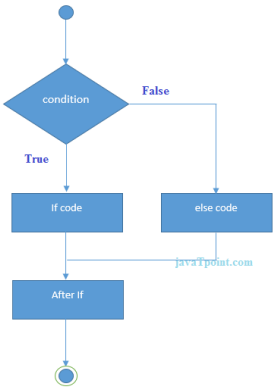
```
Age is greater than 18
```

## Java if-else Statement

The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

**Syntax:**

```
1.  if(condition){
2.  //code if condition is true
3.  }else{
4.  //code if condition is false
5.  }
```




**Example:**

```
1. //A Java Program to demonstrate the use of if-else statement.
2. //It is a program of odd and even number.
3. public class IfElseExample {
4. public static void main(String[] args) {
5.    //defining a variable
6.    int number=13;
7.    //Check if the number is divisible by 2 or not
8.    if(number%2==0){
9.       System.out.println("even number");
10.   }else{
11.      System.out.println("odd number");
12.   }
13. }
14. }
```

Output:

```
odd number
```

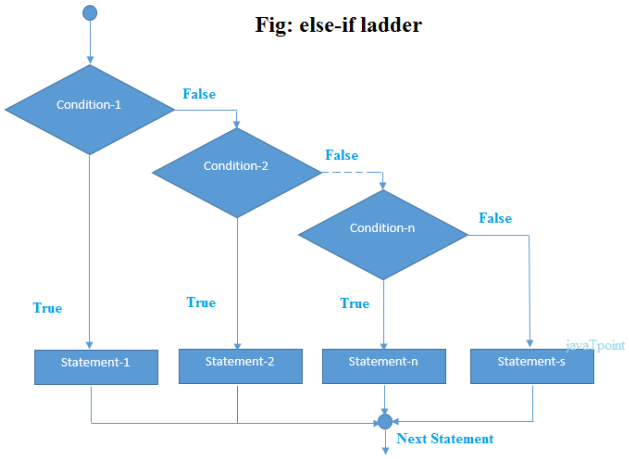
## Java if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

**Syntax:**

```
1. if(condition1){
2. //code to be executed if condition1 is true
3. }else if(condition2){
4. //code to be executed if condition2 is true
5. }
6. else if(condition3){
7. //code to be executed if condition3 is true
8. }
9. ...
10. else{
11. //code to be executed if all the conditions are false
12. }
```

Fig: else-if ladder




**Example:**

```
1.  //Java Program to demonstrate the use of If else-if ladder.
2.  //It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.
3.  public class IfElseIfExample {
4.  public static void main(String[] args) {
5.     int marks=65;
6.
7.     if(marks<50){
8.        System.out.println("fail");
9.     }
10.    else if(marks>=50 && marks<60){
11.       System.out.println("D grade");
12.    }
13.    else if(marks>=60 && marks<70){
14.       System.out.println("C grade");
15.    }
16.    else if(marks>=70 && marks<80){
17.       System.out.println("B grade");
18.    }
19.    else if(marks>=80 && marks<90){
20.       System.out.println("A grade");
21.    }else if(marks>=90 && marks<100){
22.       System.out.println("A+ grade");
23.    }else{
24.       System.out.println("Invalid!");
25.    }
26. }
27. }
```

Output:

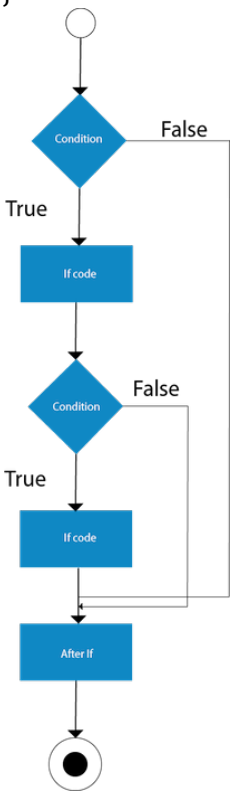
```
C grade
```

## Java Nested if statement

The nested if statement represents the *if block within another if block*. Here, the inner if block condition executes only when outer if block condition is true.

**Syntax:**

```
if(condition){
    //code to be executed
        if(condition){
          //code to be executed
    }
}
```




**Example:**

```
//Java Program to demonstrate the use of Nested If Statement.
public class JavaNestedIfExample {
public static void main(String[] args) {
    //Creating two variables for age and weight
    int age=20;
    int weight=80;
    //applying condition on age and weight
    if(age>=18){
       if(weight>50){
          System.out.println("You are eligible to donate blood");
       }
    }
```

```
}}
```

Output:

```
You are eligible to donate blood
```

**Example 2:**

```
//Java Program to demonstrate the use of Nested If Statement.
public class JavaNestedIfExample2 {
public static void main(String[] args) {
    //Creating two variables for age and weight
    int age=25;
    int weight=48;
    //applying condition on age and weight
    if(age>=18){
        if(weight>50){
            System.out.println("You are eligible to donate blood");
        } else{
            System.out.println("You are not eligible to donate blood");
        }
    } else{
      System.out.println("Age must be greater than 18");
    }
} }
```

Output:

```
You are not eligible to donate blood
```

## Java Switch Statement

The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use strings in the switch statement.
In other words, the switch statement tests the equality of a variable against multiple values.

### Points to Remember

- There can be *one or N number of case values* for a switch expression.
- The case value must be of switch expression type only. The case value must be *literal or constant*. It doesn't allow variables.
- The case values must be *unique*. In case of duplicate value, it renders compile-time error.
- The Java switch expression must be of *byte, short, int, long (with its Wrapper type), enums and string*.
- Each case statement can have a *break statement* which is optional. When control reaches to the break statement, it jumps the control after the switch expression. If a break statement is not found, it executes the next case.
- The case value can have a *default label* which is optional.

**Syntax:**

```
switch(expression){
case value1:
 //code to be executed;
```

```
    break;  //optional
  case value2:
   //code to be executed;
   break;  //optional
   ......

  default:
   code to be executed if all cases are not matched;
 }
```

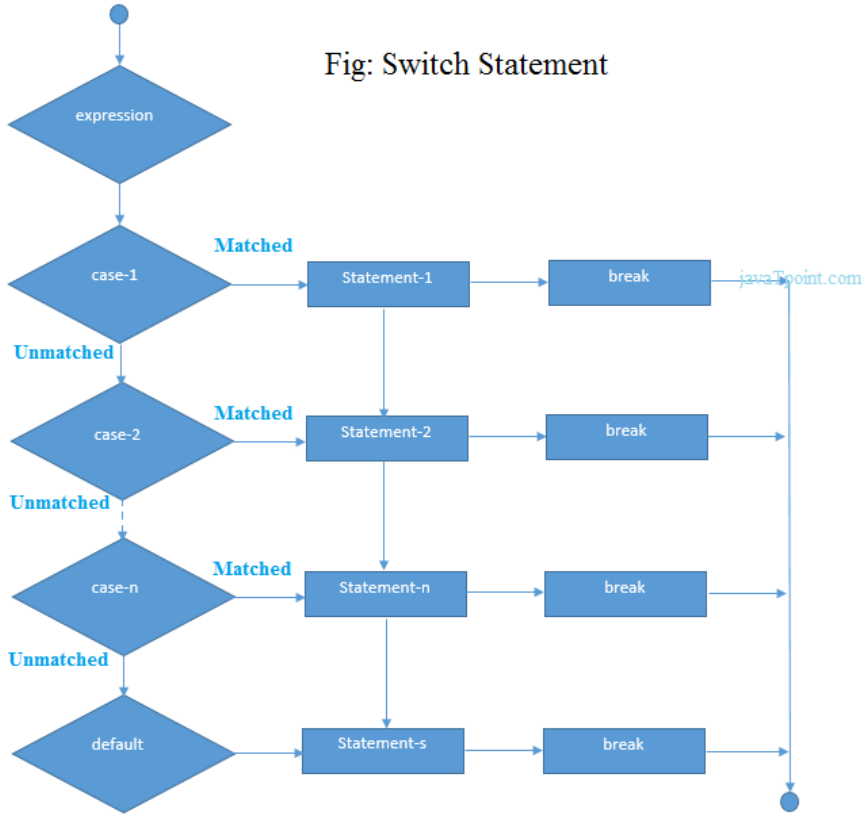



Fig: Switch Statement

**Example:**

```
public class SwitchExample {
public static void main(String[] args) {
    //Declaring a variable for switch expression
    int number=20;
    //Switch expression
    switch(number){
    //Case statements
    case 10: System.out.println("10");
    break;
    case 20: System.out.println("20");
```

```
    break;
    case 30: System.out.println("30");
    break;
    //Default case statement
    default:System.out.println("Not in 10, 20 or 30");
    }
}
}
```

Output:

```
20
```

**Finding Month Example:**

```
//Java Program to demonstrate the example of Switch statement
//where we are printing month name for the given number
public class SwitchMonthExample {
public static void main(String[] args) {
    //Specifying month number
    int month=7;
    String monthString="";
    //Switch statement
    switch(month){
    //case statements within the switch block
    case 1: monthString="1 - January";
    break;
    case 2: monthString="2 - February";
    break;
    case 3: monthString="3 - March";
    break;
    case 4: monthString="4 - April";
    break;
    case 5: monthString="5 - May";
    break;
    case 6: monthString="6 - June";
    break;
    case 7: monthString="7 - July";
    break;
    case 8: monthString="8 - August";
    break;
    case 9: monthString="9 - September";
    break;
    case 10: monthString="10 - October";
    break;
```

```
31.     case 11: monthString="11 - November";
32.     break;
33.     case 12: monthString="12 - December";
34.     break;
35.     default:System.out.println("Invalid Month!");
36.     }
37.     //Printing month of the given number
38.     System.out.println(monthString);
39. }
40. }
```

Output:

```
7 - July
```

### Java Switch Statement is fall-through

The Java switch statement is fall-through. It means it executes all statements after the first match if a break statement is not present.

**Example:**

```
1.  //Java Switch Example where we are omitting the
2.  //break statement
3.  public class SwitchExample2 {
4.  public static void main(String[] args) {
5.     int number=20;
6.     //switch expression with int value
7.     switch(number){
8.     //switch cases without break statements
9.     case 10: System.out.println("10");
10.    case 20: System.out.println("20");
11.    case 30: System.out.println("30");
12.    default:System.out.println("Not in 10, 20 or 30");
13.    }
14. }
15. }
```

Output:

```
20
30
Not in 10, 20 or 30
```

### Java Switch Statement with String

Java allows us to use strings in switch expression since Java SE 7. The case statement should be string literal.

**Example:**

```
1.  //Java Program to demonstrate the use of Java Switch
2.  //statement with String
3.  public class SwitchStringExample {
4.  public static void main(String[] args) {
5.     //Declaring String variable
```

```
6.      String levelString="Expert";
7.      int level=0;
8.      //Using String in Switch expression
9.      switch(levelString){
10.     //Using String Literal in Switch case
11.     case "Beginner": level=1;
12.     break;
13.     case "Intermediate": level=2;
14.     break;
15.     case "Expert": level=3;
16.     break;
17.     default: level=0;
18.     break;
19.     }
20.     System.out.println("Your Level is: "+level);
21. }
22. }
```

Output:

```
Your Level is: 3
```

### Java Nested Switch Statement

We can use switch statement inside other switch statement in Java. It is known as nested switch statement.

**Example:**

```
1.  //Java Program to demonstrate the use of Java Nested Switch
2.  public class NestedSwitchExample {
3.     public static void main(String args[])
4.      {
5.      //C - CSE, E - ECE, M - Mechanical
6.        char branch = 'C';
7.        int collegeYear = 4;
8.        switch( collegeYear )
9.        {
10.          case 1:
11.             System.out.println("English, Maths, Science");
12.             break;
13.          case 2:
14.             switch( branch )
15.             {
16.                case 'C':
17.                   System.out.println("Operating System, Java, Data Structure");
18.                   break;
19.                case 'E':
```

```
                System.out.println("Micro processors, Logic switching theory");
                break;
            case 'M':
                System.out.println("Drawing, Manufacturing Machines");
                break;
        }
        break;
    case 3:
        switch( branch )
        {
            case 'C':
                System.out.println("Computer Organization, MultiMedia");
                break;
            case 'E':
                System.out.println("Fundamentals of Logic Design, Microelectronics");
                break;
            case 'M':
                System.out.println("Internal Combustion Engines, Mechanical Vibration");
                break;
        }
        break;
    case 4:
        switch( branch )
        {
            case 'C':
                System.out.println("Data Communication and Networks, MultiMedia");
                break;
            case 'E':
                System.out.println("Embedded System, Image Processing");
                break;
            case 'M':
                System.out.println("Production Technology, Thermal Engineering");
                break;
        }
        break;
    }
  }
}
```

Output:

```
Data Communication and Networks, MultiMedia
```

## Java Enum in Switch Statement

Java allows us to use enum in switch statement.

**Example:**

```
1.  //Java Program to demonstrate the use of Enum
2.  //in switch statement
3.  public class JavaSwitchEnumExample {
4.       public enum Day {  Sun, Mon, Tue, Wed, Thu, Fri, Sat  }
5.       public static void main(String args[])
6.       {
7.         Day[] DayNow = Day.values();
8.           for (Day Now : DayNow)
9.           {
10.              switch (Now)
11.              {
12.                 case Sun:
13.                    System.out.println("Sunday");
14.                    break;
15.                 case Mon:
16.                    System.out.println("Monday");
17.                    break;
18.                 case Tue:
19.                    System.out.println("Tuesday");
20.                    break;
21.                 case Wed:
22.                    System.out.println("Wednesday");
23.                    break;
24.                 case Thu:
25.                    System.out.println("Thursday");
26.                    break;
27.                 case Fri:
28.                    System.out.println("Friday");
29.                    break;
30.                 case Sat:
31.                    System.out.println("Saturday");
32.                    break;
33.              }
34.           }
35.       }
36. }
```

Output:

```
Sunday
Monday
Twesday
Wednesday
```

```
Thursday
Friday
Saturday
```

### Java Wrapper in Switch Statement

Java allows us to use four wrapper classes: Byte, Short, Integer and Long in switch statement.

**Example:**

```
1.  //Java Program to demonstrate the use of Wrapper class
2.  //in switch statement
3.  public class WrapperInSwitchCaseExample {
4.      public static void main(String args[])
5.      {
6.          Integer age = 18;
7.          switch (age)
8.          {
9.              case (16):
10.                 System.out.println("You are under 18.");
11.                 break;
12.             case (18):
13.                 System.out.println("You are eligible for vote.");
14.                 break;
15.             case (65):
16.                 System.out.println("You are senior citizen.");
17.                 break;
18.             default:
19.                 System.out.println("Please give the valid age.");
20.                 break;
21.         }
22.     }
23. }
```

Output:

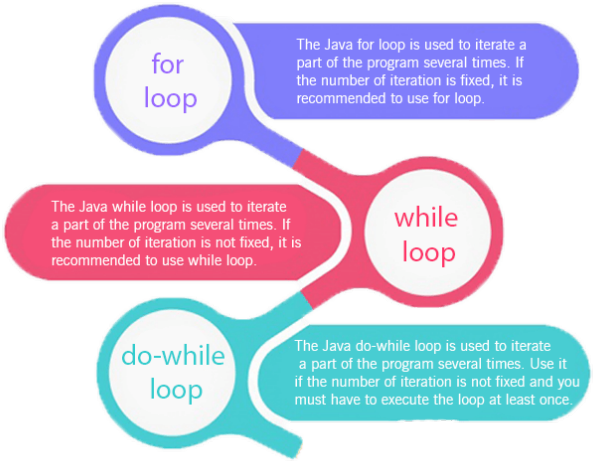
```
You are eligible for vote.
```

### Loops in Java

In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions becom

three types of loops in java.

- for loop
- while loop
- do-while loop




Java For Loop vs While Loop vs Do While Loop

| Comparison | for loop | while loop | do while loop |
|---|---|---|---|
| Introduction | The Java for loop is a control flow statement that iterates a part of the programs multiple times. | The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition. | The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition. |
| When to use | If the number of iteration is fixed, it is recommended to use for loop. | If the number of iteration is not fixed, it is recommended to use while loop. | If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop. |
| Syntax | for(init;condition;incr/decr){<br>// code to be executed<br>} | while(condition){<br>//code to be executed<br>} | do{<br>//code to be executed<br>}while(condition); |

| | | | |
|---|---|---|---|
| Example | //for loop<br>for(int i=1;i<=10;i++){<br>System.out.println(i);<br>} | //while loop<br>int i=1;<br>while(i<=10){<br>System.out.println(i);<br>i++;<br>} | //do-while loop<br>int i=1;<br>do{<br>System.out.println(i);<br>i++;<br>}while(i<=10); |
| Syntax for infinitive loop | for(;;){<br>//code to be executed<br>} | while(true){<br>//code to be executed<br>} | do{<br>//code to be executed<br>}while(true); |

## Java For Loop

e Java *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use f

There are three types of for loops in java.

- Simple For Loop
- For-each or Enhanced For Loop
- Labeled For Loop

## Java Simple For Loop

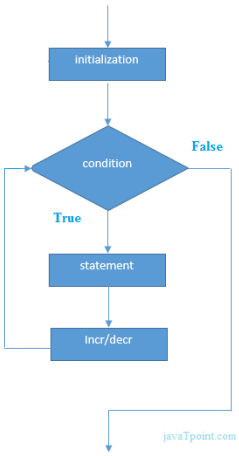
A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It cons

1. **Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, o already initialized variable. It is an optional condition.
2. **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues executio is false. It must return boolean value either true or false. It is an optional condition.
3. **Statement**: The statement of the loop is executed each time until the second condition is false.
4. **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.

**Syntax:**

```
for(initialization;condition;incr/decr){
//statement or code to be executed
}
```

**Flowchart:**




**Example:**

```
1. //Java Program to demonstrate the example of for loop
2. //which prints table of 1
3. public class ForExample {
4. public static void main(String[] args) {
5.    //Code of Java for loop
6.    for(int i=1;i<=10;i++){
7.       System.out.println(i);
8.    }
9. }
10. }
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Java for-each Loop

The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to use subscript notation.

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

**Syntax:**

```
for(Type var:array){
//code to be executed
}
```

**Example:**

```
//Java For-each loop example which prints the
//elements of the array
public class ForEachExample {
public static void main(String[] args) {
    //Declaring an array
    int arr[]={12,23,44,56,78};
    //Printing array using for-each loop
    for(int i:arr){
        System.out.println(i);
    }
}
}
```

Output:

```
12
23
44
56
78
```

Java Labeled For Loop

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

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

**Syntax:**

```
labelname:
for(initialization;condition;incr/decr){
//code to be executed
}
```

**Example:**

```
//A Java program to demonstrate the use of labeled for loop
public class LabeledForExample {
public static void main(String[] args) {
    //Using Label for outer and for loop
    aa:
        for(int i=1;i<=3;i++){
            bb:
```

```
            for(int j=1;j<=3;j++){
                if(i==2&&j==2){
                    break aa;
                }
                System.out.println(i+" "+j);
            }
        }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
```

If you use **break bb;**, it will break inner loop only which is the default behavior of any loop.

```
public class LabeledForExample2 {
public static void main(String[] args) {
    aa:
        for(int i=1;i<=3;i++){
            bb:
                for(int j=1;j<=3;j++){
                    if(i==2&&j==2){
                        break bb;
                    }
                    System.out.println(i+" "+j);
                }
        }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

### Java Infinitive For Loop

If you use two semicolons ;; in the for loop, it will be infinitive for loop.

**Syntax:**

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

**Example:**

```
//Java program to demonstrate the use of infinite for loop
//which prints an statement
public class ForExample {
public static void main(String[] args) {
   //Using no condition in for loop
   for(;;){
      System.out.println("infinitive loop");
   }
}
}
```

Output:

```
infinitive loop
infinitive loop
infinitive loop
infinitive loop
infinitive loop
ctrl+c
```

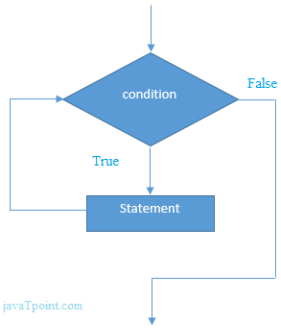
Now, you need to press ctrl+c to exit from the program.

## Java While Loop

The Java *while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

**Syntax:**

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




**Example:**

```
1. public class WhileExample {
2. public static void main(String[] args) {
3.    int i=1;
4.    while(i<=10){
5.       System.out.println(i);
6.    i++;
7.    }
8. }
9. }
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

### Java Infinitive While Loop

If you pass **true** in the while loop, it will be infinitive while loop.

**Syntax:**

```
1. while(true){
2. //code to be executed
3. }
```

**Example:**

```
1. public class WhileExample2 {
2. public static void main(String[] args) {
3.    while(true){
4.       System.out.println("infinitive while loop");
5.    }
6. }
7. }
```

Output:

```
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
ctrl+c
```

Now, you need to press ctrl+c to exit from the program.

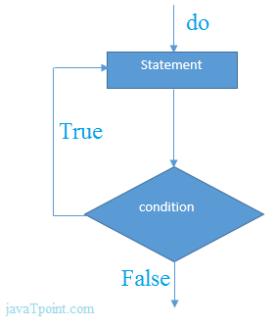
### Java do-while Loop

The Java *do-while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The Java *do-while loop* is executed at least once because condition is checked after loop body.

**Syntax:**

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




**Example:**

```
public class DoWhileExample {
public static void main(String[] args) {
    int i=1;
    do{
        System.out.println(i);
    i++;
    }while(i<=10);
}
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Java Infinitive do-while Loop

If you pass **true** in the do-while loop, it will be infinitive do-while loop.

**Syntax:**

```
do{
//code to be executed
}while(true);
```

**Example:**

```
public class DoWhileExample2 {
public static void main(String[] args) {
    do{
        System.out.println("infinitive do while loop");
    }while(true);
```

```
6. }
7. }
```

Output:

```
infinitive do while loop
infinitive do while loop
infinitive do while loop
ctrl+c
```

Now, you need to press ctrl+c to exit from the program.

## Java Break Statement

When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

The Java *break* is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

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

**Syntax:**

```
1. jump-statement;
2. break;
```

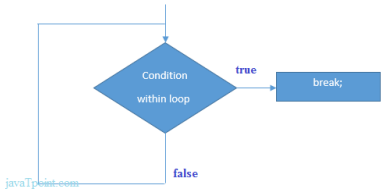



Figure: Flowchart of break statement

## Java Break Statement with Loop

**Example:**

```
1. //Java Program to demonstrate the use of break statement
2. //inside the for loop.
3. public class BreakExample {
4. public static void main(String[] args) {
5.    //using for loop
6.    for(int i=1;i<=10;i++){
7.       if(i==5){
8.          //breaking the loop
9.          break;
10.      }
11.      System.out.println(i);
12.   }
13. }
14. }
```

Output:

```
1
2
3
4
```

Java Break Statement with Inner Loop

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

**Example:**

```
1. //Java Program to illustrate the use of break statement
2. //inside an inner loop
3. public class BreakExample2 {
4. public static void main(String[] args) {
5.         //outer loop
6.         for(int i=1;i<=3;i++){
7.             //inner loop
8.             for(int j=1;j<=3;j++){
9.               if(i==2&&j==2){
10.                 //using break statement inside the inner loop
11.                 break;
12.               }
13.               System.out.println(i+" "+j);
14.             }
15.         }
16. }
17. }
```

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

Java Break Statement with Labeled For Loop

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

**Example:**

```
1. //Java Program to illustrate the use of continue statement
2. //with label inside an inner loop to break outer loop
3. public class BreakExample3 {
```

```
4.  public static void main(String[] args) {
5.          aa:
6.          for(int i=1;i<=3;i++){
7.              bb:
8.              for(int j=1;j<=3;j++){
9.                  if(i==2&&j==2){
10.                     //using break statement with label
11.                     break aa;
12.                 }
13.                 System.out.println(i+" "+j);
14.             }
15.         }
16. }
17. }
```

Output:

```
1 1
1 2
1 3
2 1
```

Java Break Statement in while loop

**Example:**

```
1.  //Java Program to demonstrate the use of break statement
2.  //inside the while loop.
3.  public class BreakWhileExample {
4.  public static void main(String[] args) {
5.     //while loop
6.     int i=1;
7.     while(i<=10){
8.        if(i==5){
9.           //using break statement
10.          i++;
11.          break;//it will break the loop
12.       }
13.       System.out.println(i);
14.       i++;
15.    }
16. }
17. }
```

Output:

```
1
```

```
2
3
4
```

Java Break Statement in do-while loop

**Example:**

```
1. //Java Program to demonstrate the use of break statement
2. //inside the Java do-while loop.
3. public class BreakDoWhileExample {
4. public static void main(String[] args) {
5.    //declaring variable
6.    int i=1;
7.    //do-while loop
8.    do{
9.       if(i==5){
10.         //using break statement
11.         i++;
12.         break;//it will break the loop
13.      }
14.      System.out.println(i);
15.      i++;
16.   }while(i<=10);
17. }
18. }
```

Output:

```
1
2
3
4
```

java Continue Statement

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

The Java *continue statement* is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only.

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

**Syntax:**

```
1. jump-statement;
2. continue;
```

Java Continue Statement Example

**Example:**

```
1. //Java Program to demonstrate the use of continue statement
```

```
//inside the for loop.
public class ContinueExample {
public static void main(String[] args) {
    //for loop
    for(int i=1;i<=10;i++){
        if(i==5){
            //using continue statement
            continue;//it will skip the rest statement
        }
        System.out.println(i);
    }
}
}
```

Output:

```
1
2
3
4
6
7
8
9
10
```

As you can see in the above output, 5 is not printed on the console. It is because the loop is continued when it reaches to 5.

### Java Continue Statement with Inner Loop

It continues inner loop only if you use the continue statement inside the inner loop.

**Example:**

```
//Java Program to illustrate the use of continue statement
//inside an inner loop
public class ContinueExample2 {
public static void main(String[] args) {
        //outer loop
        for(int i=1;i<=3;i++){
            //inner loop
            for(int j=1;j<=3;j++){
                if(i==2&&j==2){
                    //using continue statement inside inner loop
                    continue;
                }
                System.out.println(i+" "+j);
            }
        }
}
```

```
}
```

Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

## Java Continue Statement with Labeled For Loop

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

**Example:**

```
//Java Program to illustrate the use of continue statement
//with label inside an inner loop to continue outer loop
public class ContinueExample3 {
public static void main(String[] args) {
        aa:
        for(int i=1;i<=3;i++){
            bb:
            for(int j=1;j<=3;j++){
               if(i==2&&j==2){
                  //using continue statement with label
                  continue aa;
               }
               System.out.println(i+" "+j);
            }
        }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

## Java Continue Statement in while loop

**Example:**

```
//Java Program to demonstrate the use of continue statement
//inside the while loop.
public class ContinueWhileExample {
public static void main(String[] args) {
   //while loop
```

```
6.      int i=1;
7.      while(i<=10){
8.        if(i==5){
9.          //using continue statement
10.         i++;
11.         continue;//it will skip the rest statement
12.       }
13.       System.out.println(i);
14.       i++;
15.     }
16. }
17. }
```

Test it Now

Output:

```
1
2
3
4
6
7
8
9
10
```

Java Continue Statement in do-while loop

**Example:**

```
1.  //Java Program to demonstrate the use of continue statement
2.  //inside the Java do-while loop.
3.  public class ContinueDoWhileExample {
4.  public static void main(String[] args) {
5.     //declaring variable
6.     int i=1;
7.     //do-while loop
8.     do{
9.        if(i==5){
10.            //using continue statement
11.             i++;
12.          continue;//it will skip the rest statement
13.       }
14.       System.out.println(i);
15.       i++;
16.    }while(i<=10);
17. }
18. }
```

Output:

```
1
2
3
4
6
7
8
9
10
```

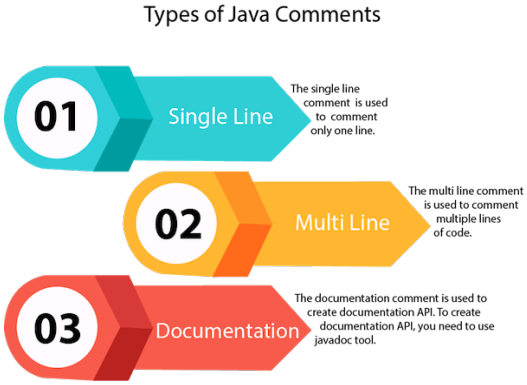
## Java Comments

The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

## Types of Java Comments

There are 3 types of comments in java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment




### 1) Java Single Line Comment

The single line comment is used to comment only one line.

**Syntax:**

```
1. //This is single line comment
```

**Example:**

```
1. public class CommentExample1 {
2. public static void main(String[] args) {
3.    int i=10;//Here, i is a variable
4.    System.out.println(i);
5. }
6. }
```

Output:

```
10
```

### 2) Java Multi Line Comment

The multi line comment is used to comment multiple lines of code.

**Syntax:**

```
1. /*
```

```
2. This
3. is
4. multi line
5. comment
6. */
```

**Example:**

```
1. public class CommentExample2 {
2. public static void main(String[] args) {
3. /* Let's declare and
4.  print variable in java. */
5.    int i=10;
6.    System.out.println(i);
7. }
8. }
```

Output:

```
10
```

### 3) Java Documentation Comment

The documentation comment is used to create documentation API. To create documentation API, you need to use **javadoc tool**.

**Syntax:**

```
1. /**
2. This
3. is
4. documentation
5. comment
6. */
```

**Example:**

```
1. /** The Calculator class provides methods to get addition and subtraction of given 2 numbers.*/
2. public class Calculator {
3. /** The add() method returns addition of given numbers.*/
4. public static int add(int a, int b){return a+b;}
5. /** The sub() method returns subtraction of given numbers.*/
6. public static int sub(int a, int b){return a-b;}
7. }
```

Compile it by javac tool:

```
javac Calculator.java
```

Create Documentation API by javadoc tool:

```
javadoc Calculator.java
```

Now, there will be HTML files created for your Calculator class in the current directory. Open the HTML files and see the explanation of Calculator class provided through documentation comment.

## Java OOPs Concepts

Object-Oriented Programming is a paradigm that provides many concepts such as **inheritance**, **data binding**, **polymorphism**, etc.

**Simula** is considered the first object-oriented programming language. The programming paradigm where

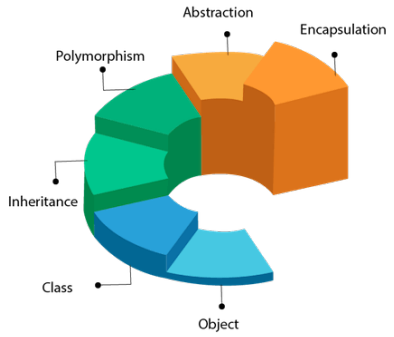
everything is represented as an object is known as a truly object-oriented programming language.
**Smalltalk** is considered the first truly object-oriented programming language.
The popular object-oriented languages are Java, C#, PHP, Python, C++, etc.
The main aim of object-oriented programming is to implement real-world entities for example object, classes, abstraction, inheritance, polymorphism, etc.

## OOPs (Object-Oriented Programming System)

**Object** means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

OOPs (Object-Oriented Programming System)




## Object


Any entity that has state and behavior is known as an object. For example a chair, pen, table, keyboard, bike, etc. It can be physical or logical.
An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.
**Example:** A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

## Class

*Collection of objects* is called class. It is a logical entity.
A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

## Inheritance

*When one object acquires all the properties and behaviors of a parent object*, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.


## Polymorphism

If *one task is performed by different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.
In Java, we use method overloading and method overriding to achieve polymorphism.
Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

## Abstraction

*Hiding internal details and showing functionality* is known as abstraction. For example phone call, we don't know the internal processing.
In Java, we use abstract class and interface to achieve abstraction.


## Encapsulation

*Binding (or wrapping) code and data together into a single unit are known as encapsulation*. For example capsule, it is wrapped with different medicines.
A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

---

## Advantage of OOPs over Procedure-oriented programming language

1) OOPs makes development and maintenance easier whereas in a procedure-oriented programming language it is not easy to manage if code grows as project size increases.
2) OOPs provides data hiding whereas in a procedure-oriented programming language a global data can be accessed from anywhere.

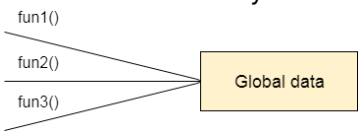



Figure: Data Representation in Procedure-Oriented Programming

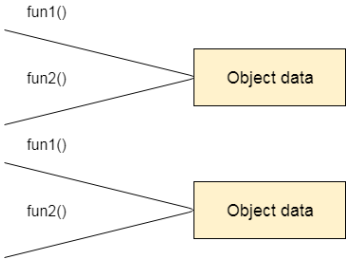



Figure: Data Representation in Object-Oriented Programming
3) OOPs provides the ability to simulate real-world event much more effectively. We can provide the

solution of real word problem if we are using the Object-Oriented Programming language.

### Java Naming conventions

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.
But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.
All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

### Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.
The following are the key rules that must be followed by every identifier:

- The name must not contain any white spaces.
- The name should not start with special characters like & (ampersand), $ (dollar), _ (underscore).

Let's see some other rules that should be followed by identifiers.

### Class

- It should start with the uppercase letter.
- It should be a noun such as Color, Button, System, Thread, etc.
- Use appropriate words, instead of acronyms.
- **Example: -**

```
1. public class Employee
2. {
3. //code snippet
4. }
```

### Interface

- It should start with the uppercase letter.
- It should be an adjective such as Runnable, Remote, ActionListener.
- Use appropriate words, instead of acronyms.
- **Example: -**

```
1. interface Printable
2. {
3. //code snippet
4. }
```

### Method

- It should start with lowercase letter.
- It should be a verb such as main(), print(), println().
- If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().
- **Example:-**

```
1.  class Employee
2. {
```

```
3. //method
4. void draw()
5. {
6. //code snippet
7. }
8. }
```

Variable

- It should start with a lowercase letter such as id, name.
- It should not start with the special characters like & (ampersand), $ (dollar), _ (underscore).
- If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
- Avoid using one-character variables such as x, y, z.
- **Example :-**

```
1.  class Employee
2. {
3. //variable
4. int id;
5. //code snippet
6. }
```

Package

- It should be a lowercase letter such as java, lang.
- If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.
- **Example :-**

```
1. package com.javatpoint; //package
2. class Employee
3. {
4. //code snippet
5. }
```

Constant

- It should be in uppercase letters such as RED, YELLOW.
- If the name contains multiple words, it should be separated by an underscore(_) such as MAX_PRIORITY.
- It may contain digits but not as the first letter.
- **Example :-**

```
1. class Employee
2. {
3. //constant
4.  static final int MIN_AGE = 18;
5. //code snippet
6. }
```

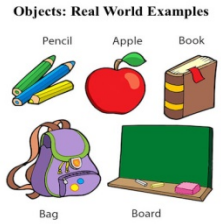
CamelCase in java naming conventions

Java follows camel-case syntax for naming the class, interface, method, and variable.
If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, ActionEvent, ActionListener, etc.

## Objects and Classes in Java

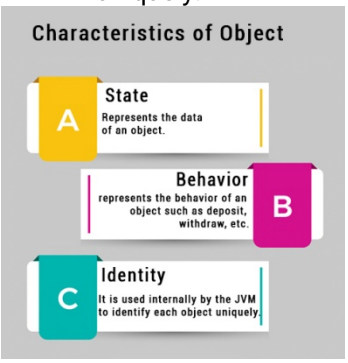
In this page, we will learn about Java objects and classes. In object-oriented programming technique, we design a program using objects and classes.
An object in Java is the physical as well as logical entity whereas a class in Java is a logical entity only.

### What is an object in Java




An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.
An object has three characteristics:

- **State:**represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.




For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.
**An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.
**Object Definitions:**

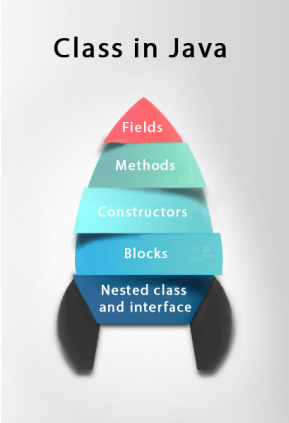
- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

---

### What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
A class in Java can contain:

- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- **Nested class and interface**




Syntax to declare a class:

```
1. class <class_name>{
2.     field;
3.     method;
4. }
```

---

### Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

---

### Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

### Advantage of Method

- Code Reusability
- Code Optimization

---

### new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

---

### Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.
Here, we are creating a main() method inside the class.
*File: Student.java*

```
1. //Java Program to illustrate how to define a class and fields
2. //Defining a Student class.
```

```
3.  class Student{
4.   //defining fields
5.   int id;//field or data member or instance variable
6.   String name;
7.   //creating main method inside the Student class
8.   public static void main(String args[]){
9.    //Creating an object or instance
10.   Student s1=new Student();//creating an object of Student
11.   //Printing values of the object
12.   System.out.println(s1.id);//accessing member through reference variable
13.   System.out.println(s1.name);
14.  }
15. }
```

Output:

```
0
null
```

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different java files or single java file. If you define multiple classes in a single java source file, it is a good idea to save the file name with the class name which has main() method.

*File: TestStudent1.java*

```
1.  //Java Program to demonstrate having the main method in
2.  //another class
3.  //Creating Student class.
4.  class Student{
5.   int id;
6.   String name;
7.  }
8.  //Creating another class TestStudent1 which contains the main method
9.  class TestStudent1{
10.  public static void main(String args[]){
11.   Student s1=new Student();
12.   System.out.println(s1.id);
13.   System.out.println(s1.name);
14.  }
15. }
```

Output:

```
0
null
```

3 Ways to initialize object

There are 3 ways to initialize object in java.

1. By reference variable

2. By method
3. By constructor

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

*File: TestStudent2.java*

```
1. class Student{
2.  int id;
3.  String name;
4. }
5. class TestStudent2{
6.  public static void main(String args[]){
7.   Student s1=new Student();
8.   s1.id=101;
9.   s1.name="Sonoo";
10.  System.out.println(s1.id+" "+s1.name);//printing members with a white space
11.  }
12. }
```

Output:

```
101 Sonoo
```

We can also create multiple objects and store information in it through reference variable.

*File: TestStudent3.java*

```
1. class Student{
2.  int id;
3.  String name;
4. }
5. class TestStudent3{
6.  public static void main(String args[]){
7.   //Creating objects
8.   Student s1=new Student();
9.   Student s2=new Student();
10.  //Initializing objects
11.  s1.id=101;
12.  s1.name="Sonoo";
13.  s2.id=102;
14.  s2.name="Amit";
15.  //Printing data
16.  System.out.println(s1.id+" "+s1.name);
17.  System.out.println(s2.id+" "+s2.name);
18.  }
19. }
```

Output:

```
101 Sonoo
102 Amit
```

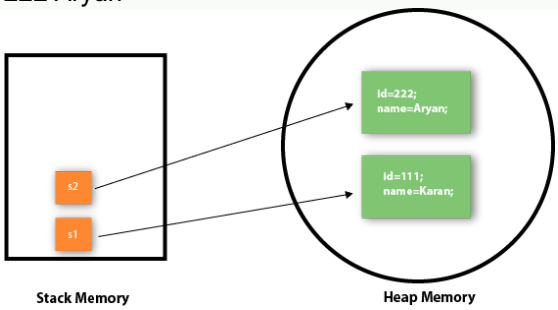
.

*File: TestStudent4.java*

```
1.  class Student{
2.   int rollno;
3.   String name;
4.   void insertRecord(int r, String n){
5.    rollno=r;
6.    name=n;
7.   }
8.   void displayInformation(){System.out.println(rollno+" "+name);}
9.  }
10. class TestStudent4{
11.  public static void main(String args[]){
12.   Student s1=new Student();
13.   Student s2=new Student();
14.   s1.insertRecord(111,"Karan");
15.   s2.insertRecord(222,"Aryan");
16.   s1.displayInformation();
17.   s2.displayInformation();
18.  }
19. }
```

Output:

```
111 Karan
222 Aryan
```




As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

*File: TestEmployee.java*

```
1.  class Employee{
2.     int id;
3.     String name;
4.     float salary;
5.     void insert(int i, String n, float s) {
6.         id=i;
7.         name=n;
8.         salary=s;
```

```
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}
public class TestEmployee {
public static void main(String[] args) {
    Employee e1=new Employee();
    Employee e2=new Employee();
    Employee e3=new Employee();
    e1.insert(101,"ajeet",45000);
    e2.insert(102,"irfan",25000);
    e3.insert(103,"nakul",55000);
    e1.display();
    e2.display();
    e3.display();
}
}
```

Output:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
```

### Object and Class Example: Rectangle

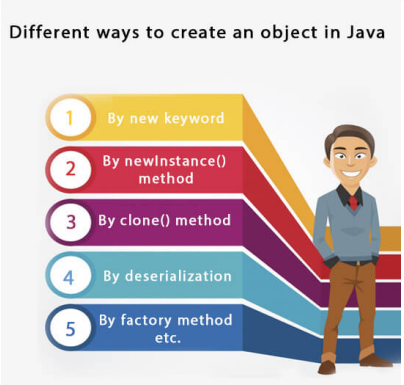
There is given another example that maintains the records of Rectangle class.

*File: TestRectangle1.java*

```
class Rectangle{
 int length;
 int width;
 void insert(int l, int w){
  length=l;
  width=w;
 }
 void calculateArea(){System.out.println(length*width);}
}
class TestRectangle1{
 public static void main(String args[]){
  Rectangle r1=new Rectangle();
  Rectangle r2=new Rectangle();
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
}
}
```

Output:

55
45




Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

```
1. new Calculation();//anonymous object
```

Calling method through a reference:

```
1. Calculation c=new Calculation();
2. c.fact(5);
```

Calling method through an anonymous object

```
1. new Calculation().fact(5);
```

Let's see the full example of an anonymous object in Java.

```
1. class Calculation{
2.  void fact(int  n){
3.   int fact=1;
4.   for(int i=1;i<=n;i++){
5.    fact=fact*i;
6.   }
7.  System.out.println("factorial is "+fact);
8. }
9. public static void main(String args[]){
10.  new Calculation().fact(5);//calling method with anonymous object
11. }
12. }
```

Output:

```
Factorial is 120
```

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

```
1. int a=10, b=20;
```

Initialization of refernce variables:

```
1. Rectangle r1=new Rectangle(), r2=new Rectangle();//creating two objects
```

Let's see the example:

```
1. //Java Program to illustrate the use of Rectangle class which
```

```
2.  //has length and width data members
3.  class Rectangle{
4.   int length;
5.   int width;
6.   void insert(int l,int w){
7.    length=l;
8.    width=w;
9.   }
10.  void calculateArea(){System.out.println(length*width);}
11. }
12. class TestRectangle2{
13.  public static void main(String args[]){
14.   Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
15.   r1.insert(11,5);
16.   r2.insert(3,15);
17.   r1.calculateArea();
18.   r2.calculateArea();
19. }
20. }
```

Output:

```
55
45
```

Real World Example: Account

*File: TestAccount.java*

```
1.  //Java Program to demonstrate the working of a banking-system
2.  //where we deposit and withdraw amount from our account.
3.  //Creating an Account class which has deposit() and withdraw() methods
4.  class Account{
5.  int acc_no;
6.  String name;
7.  float amount;
8.  //Method to initialize object
9.  void insert(int a,String n,float amt){
10. acc_no=a;
11. name=n;
12. amount=amt;
13. }
14. //deposit method
15. void deposit(float amt){
16. amount=amount+amt;
17. System.out.println(amt+" deposited");
18. }
```

```
19. //withdraw method
20. void withdraw(float amt){
21. if(amount<amt){
22. System.out.println("Insufficient Balance");
23. }else{
24. amount=amount-amt;
25. System.out.println(amt+" withdrawn");
26. }
27. }
28. //method to check the balance of the account
29. void checkBalance(){System.out.println("Balance is: "+amount);}
30. //method to display the values of an object
31. void display(){System.out.println(acc_no+" "+name+" "+amount);}
32. }
33. //Creating a test class to deposit and withdraw amount
34. class TestAccount{
35. public static void main(String[] args){
36. Account a1=new Account();
37. a1.insert(832345,"Ankit",1000);
38. a1.display();
39. a1.checkBalance();
40. a1.deposit(40000);
41. a1.checkBalance();
42. a1.withdraw(15000);
43. a1.checkBalance();
44. }}
```

Output:

```
832345 Ankit 1000.0
Balance is: 1000.0
40000.0 deposited
Balance is: 41000.0
15000.0 withdrawn
Balance is: 26000.0
```

### Constructors in Java

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the object is created, and memory is allocated for the object.

It is a special type of method which is used to initialize the object.

### When is a constructor called

Every time an object is created using new() keyword, at least one constructor is called. It calls a default constructor.

**Note:** It is called constructor because it constructs the values at the time of object creation. It is not

necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

### Rules for creating Java constructor

There are two rules defined for the constructor.

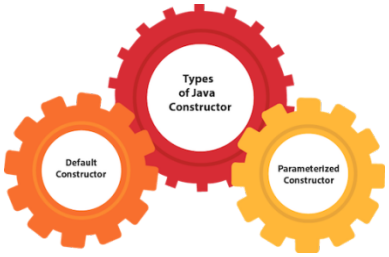
1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

> Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

### Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor




---

### Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

### Syntax of default constructor:

1. <class_name>(){}

### Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation

```
1. //Java Program to create and call a default constructor
2. class Bike1{
3. //creating a default constructor
4. Bike1(){System.out.println("Bike is created");}
5. //main method
6. public static void main(String args[]){
7. //calling a default constructor
8. Bike1 b=new Bike1();
9. }
10. }
```

Output:

```
Bike is created
```

> Rule: If there is no constructor in a class, compiler automatically creates a default constructor.


Q) What is the purpose of a default constructor?
The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.
Example of default constructor that displays the default values

```
1. //Let us see another example of default constructor
2. //which displays the default values
3. class Student3{
4. int id;
5. String name;
6. //method to display the value of id and name
7. void display(){System.out.println(id+" "+name);}
8.
9. public static void main(String args[]){
10. //creating objects
11. Student3 s1=new Student3();
12. Student3 s2=new Student3();
13. //displaying values of the object
14. s1.display();
15. s2.display();
16. }
17. }
```

Output:
0 null
0 null
**Explanation:**In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

---

Java Parameterized Constructor
A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?
The parameterized constructor is used to provide different values to the distinct objects. However, you can provide the same values also.
Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
1. //Java Program to demonstrate the use of parameterized constructor
2. class Student4{
3.    int id;
4.    String name;
```

```
5.      //creating a parameterized constructor
6.      Student4(int i,String n){
7.      id = i;
8.      name = n;
9.      }
10.     //method to display the values
11.     void display(){System.out.println(id+" "+name);}
12.
13.     public static void main(String args[]){
14.     //creating objects and passing values
15.     Student4 s1 = new Student4(111,"Karan");
16.     Student4 s2 = new Student4(222,"Aryan");
17.     //calling method to display the values of object
18.     s1.display();
19.     s2.display();
20.    }
21. }
```

Output:

```
111 Karan
222 Aryan
```

---

## Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

### Example of Constructor Overloading

```
1.  //Java program to overload constructors in java
2.  class Student5{
3.     int id;
4.     String name;
5.     int age;
6.     //creating two arg constructor
7.     Student5(int i,String n){
8.     id = i;
9.     name = n;
10.    }
11.    //creating three arg constructor
12.    Student5(int i,String n,int a){
13.    id = i;
14.    name = n;
15.    age=a;
```

```
16.    }
17.    void display(){System.out.println(id+" "+name+" "+age);}
18.
19.    public static void main(String args[]){
20.    Student5 s1 = new Student5(111,"Karan");
21.    Student5 s2 = new Student5(222,"Aryan",25);
22.    s1.display();
23.    s2.display();
24.   }
25. }
```

Output:

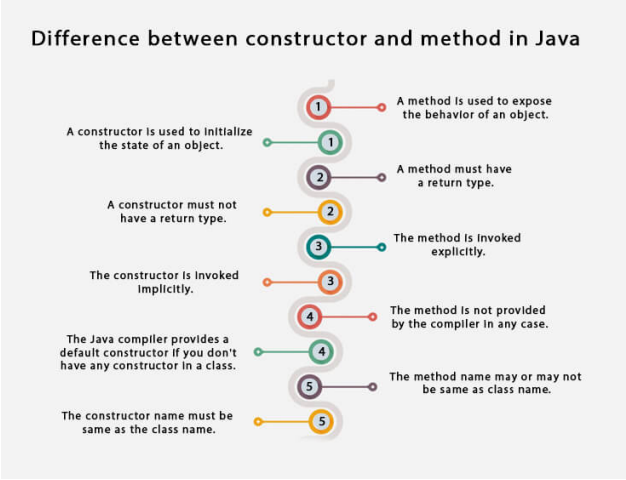
```
111 Karan 0
222 Aryan 25
```

---

## Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

| Java Constructor | Java Method |
|---|---|
| A constructor is used to initialize the state of an object. | A method is used to expose the behavior of an object. |
| A constructor must not have a return type. | A method must have a return type. |
| The constructor is invoked implicitly. | The method is invoked explicitly. |
| The Java compiler provides a default constructor if you don't have any constructor in a class. | The method is not provided by the compiler in any case. |
| The constructor name must be same as the class name. | The method name may or may not be same as class name. |




## Java Copy Constructor

There is no copy constructor in java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using java constructor.

```
1.  //Java program to initialize the values from one object to another
2.  class Student6{
3.     int id;
4.     String name;
5.     //constructor to initialize integer and string
6.     Student6(int i,String n){
7.     id = i;
8.     name = n;
9.     }
10.    //constructor to initialize another object
11.    Student6(Student6 s){
12.    id = s.id;
13.    name =s.name;
14.    }
15.    void display(){System.out.println(id+" "+name);}
16.
17.    public static void main(String args[]){
18.    Student6 s1 = new Student6(111,"Karan");
19.    Student6 s2 = new Student6(s1);
20.    s1.display();
21.    s2.display();
```

```
22.    }
23. }
```

Output:

```
111 Karan
111 Karan
```

---

### Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
1.  class Student7{
2.     int id;
3.     String name;
4.     Student7(int i,String n){
5.     id = i;
6.     name = n;
7.     }
8.     Student7(){}
9.     void display(){System.out.println(id+" "+name);}
10.
11.    public static void main(String args[]){
12.    Student7 s1 = new Student7(111,"Karan");
13.    Student7 s2 = new Student7();
14.    s2.id=s1.id;
15.    s2.name=s1.name;
16.    s1.display();
17.    s2.display();
18.   }
19. }
```

Output:

```
111 Karan
111 Karan
```

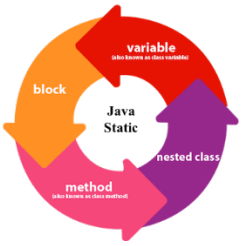
---

### Java static keyword

The **static keyword** in Java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class




1) Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

Understanding the problem without static variable

```
1. class Student{
2.     int rollno;
3.     String name;
4.     String college="ITS";
5. }
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

> Java static property is shared to all objects.

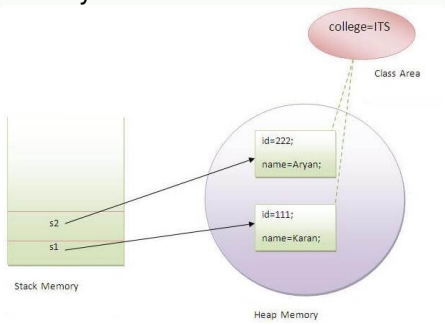
Example of static variable

```
1. //Java Program to demonstrate the use of static variable
2. class Student{
3.   int rollno;//instance variable
4.   String name;
5.   static String college ="ITS";//static variable
6.   //constructor
7.   Student(int r, String n){
8.   rollno = r;
9.   name = n;
10.  }
11.  //method to display the values
12.  void display (){System.out.println(rollno+" "+name+" "+college);}
13. }
14. //Test class to show the values of objects
15. public class TestStaticVariable1{
```

```
16.  public static void main(String args[]){
17.  Student s1 = new Student(111,"Karan");
18.  Student s2 = new Student(222,"Aryan");
19.  //we can change the college of all objects by the single line of code
20.  //Student.college="BBDIT";
21.  s1.display();
22.  s2.display();
23.  }
24. }
```

Output:

```
111 Karan ITS
222 Aryan ITS
```




---

Program of the counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

```
1.  //Java Program to demonstrate the use of an instance variable
2.  //which get memory each time when we create an object of the class.
3.  class Counter{
4.  int count=0;//will get memory each time when the instance is created
5.
6.  Counter(){
7.  count++;//incrementing value
8.  System.out.println(count);
9.  }
10.
11. public static void main(String args[]){
12. //Creating objects
13. Counter c1=new Counter();
14. Counter c2=new Counter();
15. Counter c3=new Counter();
16. }
17. }
```

Output:

```
1
1
1
```

---

Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
1. //Java Program to illustrate the use of static variable which
2. //is shared with all objects.
3. class Counter2{
4. static int count=0;//will get memory only once and retain its value
5.
6. Counter2(){
7. count++;//incrementing the value of static variable
8. System.out.println(count);
9. }
10.
11. public static void main(String args[]){
12. //creating objects
13. Counter2 c1=new Counter2();
14. Counter2 c2=new Counter2();
15. Counter2 c3=new Counter2();
16. }
17. }
```

Output:

```
1
2
3
```

---

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static method

```
1. //Java Program to demonstrate the use of a static method.
2. class Student{
3.     int rollno;
4.     String name;
5.     static String college = "ITS";
6.     //static method to change the value of static variable
7.     static void change(){
8.     college = "BBDIT";
```

```
    }
    //constructor to initialize the variable
    Student(int r, String n){
    rollno = r;
    name = n;
    }
    //method to display values
    void display(){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to create and display the values of object
public class TestStaticMethod{
  public static void main(String args[]){
  Student.change();//calling change method
  //creating objects
  Student s1 = new Student(111,"Karan");
  Student s2 = new Student(222,"Aryan");
  Student s3 = new Student(333,"Sonoo");
  //calling display method
  s1.display();
  s2.display();
  s3.display();
  }
}
```

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
       333 Sonoo BBDIT
```

---

Another example of a static method that performs a normal calculation

```
//Java Program to get the cube of a given number using the static method

class Calculate{
  static int cube(int x){
  return x*x*x;
  }

  public static void main(String args[]){
  int result=Calculate.cube(5);
  System.out.println(result);
  }
}
```

```
Output:125
```

Restrictions for the static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
class A{
 int a=40;//non static

 public static void main(String args[]){
  System.out.println(a);
 }
}
```

Output:Compile Time Error

3) Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

Example of static block

```
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[]){
    System.out.println("Hello main");
  }
}
```

Output:static block is invoked
Hello main

---

Q) Can we execute a program without main() method?

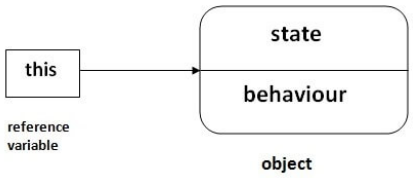
Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a java class without the main method.

```
class A3{
  static{
  System.out.println("static block is invoked");
  System.exit(0);
  }
}
```

Output:
static block is invoked
Since JDK 1.7 and above, output would be:
Error: Main method not found in class A3, please define the main method as:
  public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application

## this keyword in java

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

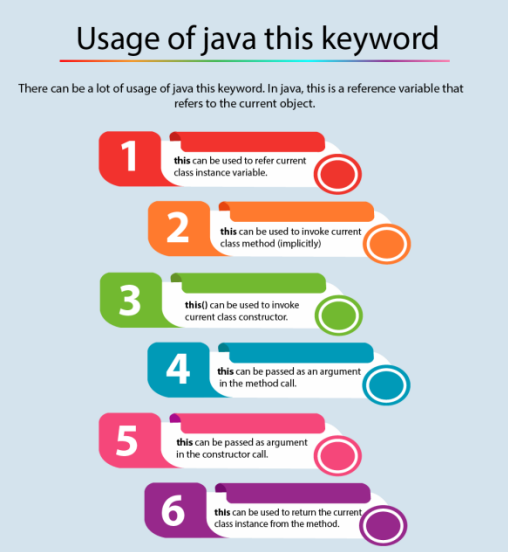



## Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

**Suggestion:** If you are beginner to java, lookup only three usage of this keyword.




---

## 1) this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and p

keyword resolves the problem of ambiguity.

### Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

```
1. class Student{
2. int rollno;
3. String name;
4. float fee;
5. Student(int rollno,String name,float fee){
```

```
rollno=rollno;
name=name;
fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
0 null 0.0
0 null 0.0
```

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distin
and instance variable.

Solution of the above problem by this keyword

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
111 ankit 5000
```

```
112 sumit 6000
```

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the followi

Program where this keyword is not required

```
1.  class Student{
2.  int rollno;
3.  String name;
4.  float fee;
5.  Student(int r,String n,float f){
6.  rollno=r;
7.  name=n;
8.  fee=f;
9.  }
10. void display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12.
13. class TestThis3{
14. public static void main(String args[]){
15. Student s1=new Student(111,"ankit",5000f);
16. Student s2=new Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}
```

Output:

```
111 ankit 5000
112 sumit 6000
```

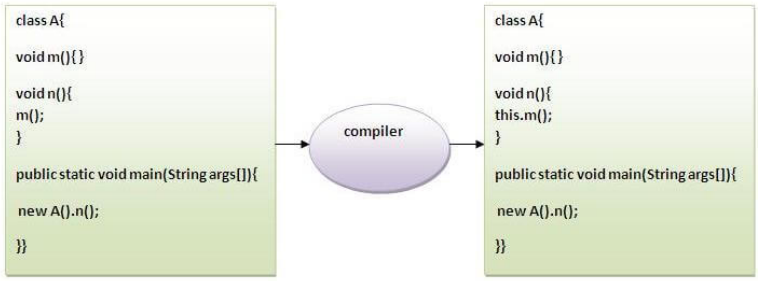
---

It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in re
use this keyword.

---

2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automati
keyword while invoking the method. Let's see the example




```
1.  class A{
```

```
2.  void m(){System.out.println("hello m");}
3.  void n(){
4.  System.out.println("hello n");
5.  //m();//same as this.m()
6.  this.m();
7.  }
8.  }
9.  class TestThis4{
10. public static void main(String args[]){
11. A a=new A();
12. a.n();
13. }}
```

Output:

```
hello n
hello m
```

3) this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other word constructor chaining.

**Calling default constructor from parameterized constructor:**

```
1.  class A{
2.  A(){System.out.println("hello a");}
3.  A(int x){
4.  this();
5.  System.out.println(x);
6.  }
7.  }
8.  class TestThis5{
9.  public static void main(String args[]){
10. A a=new A(10);
11. }}
```

Output:

```
hello a
10
```

**Calling parameterized constructor from default constructor:**

```
1.  class A{
2.  A(){
3.  this(5);
4.  System.out.println("hello a");
5.  }
6.  A(int x){
```

```
System.out.println(x);
}
}
class TestThis6{
public static void main(String args[]){
A a=new A();
}}
```

Output:

```
5
hello a
```

Real usage of this() constructor call

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the co
used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this(rollno,name,course);//reusing constructor
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis7{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

Output:

```
111 ankit java null
112 sumit java 6000
```

Rule: Call to this() must be the first statement in constructor.

```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this.fee=fee;
this(rollno,name,course);//C.T.Error
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis8{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

Compile Time Error: Call to this must be first statement in constructor

4) this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the exam

```
class S2{
  void m(S2 obj){
  System.out.println("method is invoked");
  }
  void p(){
  m(this);
  }
  public static void main(String args[]){
  S2 s1 = new S2();
  s1.p();
  }
}
```

Output:

```
method is invoked
```

Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object

---

5) this: to pass as argument in the constructor call

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the

```
1.  class B{
2.    A4 obj;
3.    B(A4 obj){
4.      this.obj=obj;
5.    }
6.    void display(){
7.      System.out.println(obj.data);//using data member of A4 class
8.    }
9.  }
10.
11. class A4{
12.   int data=10;
13.   A4(){
14.    B b=new B(this);
15.    b.display();
16.   }
17.   public static void main(String args[]){
18.    A4 a=new A4();
19.   }
20. }
```

Test it Now

Output:10

---

6) this keyword can be used to return current class instance

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (

see the example:

Syntax of this that can be returned as a statement

```
1.  return_type method_name(){
2.  return this;
3.  }
```

Example of this keyword that you return as a statement from the method

```
1.  class A{
2.  A getA(){
3.  return this;
4.  }
```

```
void msg(){System.out.println("Hello java");}
}
class Test1{
public static void main(String args[]){
new A().getA().msg();
}
}
```

Output:

```
Hello java
```

Proving this keyword

Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference vari
of both variables are same.

```
class A5{
void m(){
System.out.println(this);//prints same reference ID
}
public static void main(String args[]){
A5 obj=new A5();
System.out.println(obj);//prints the reference ID
obj.m();
}
}
```

Output:

```
A5@22b3ea59
A5@22b3ea59
```

Inheritance in Java

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

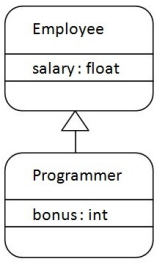
The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example




As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
```

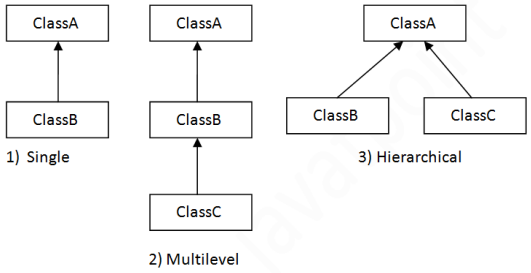
11. }

```
Programmer salary is:40000.0
 Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

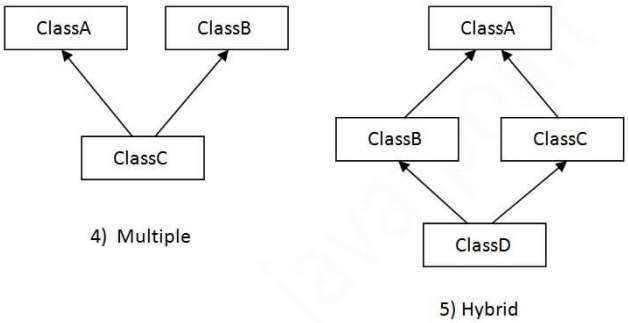
### Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical. In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.




> Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:




.

---

### Single Inheritance Example

*File: TestInheritance.java*

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
```

```
7. class TestInheritance{
8. public static void main(String args[]){
9. Dog d=new Dog();
10. d.bark();
11. d.eat();
12. }}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance Example

*File: TestInheritance2.java*

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class BabyDog extends Dog{
8. void weep(){System.out.println("weeping...");}
9. }
10. class TestInheritance2{
11. public static void main(String args[]){
12. BabyDog d=new BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example

*File: TestInheritance3.java*

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class Cat extends Animal{
```

```
8.  void meow(){System.out.println("meowing...");}
9.  }
10. class TestInheritance3{
11. public static void main(String args[]){
12. Cat c=new Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}
```

Output:

```
meowing...
eating...
```

## Method Overloading in Java

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.


## Advantage of method overloading

Method overloading *increases the readability of the program*.

## Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

> In java, Method Overloading is not possible by changing the return type of the method only.

---

## 1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
1. class Adder{
2. static int add(int a,int b){return a+b;}
3. static int add(int a,int b,int c){return a+b+c;}
4. }
5. class TestOverloading1{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(11,11,11));
9. }}
```

Output:

```
22
33
```

---

### 2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
1. class Adder{
2. static int add(int a, int b){return a+b;}
3. static double add(double a, double b){return a+b;}
4. }
5. class TestOverloading2{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(12.3,12.6));
9. }}
```

Output:

```
22
24.9
```

---

### Q) Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
1. class Adder{
2. static int add(int a,int b){return a+b;}
3. static double add(int a,int b){return a+b;}
4. }
5. class TestOverloading3{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));//ambiguity
8. }}
```

Output:

```
Compile Time Error: method add(int,int) is already defined in class Adder
```

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

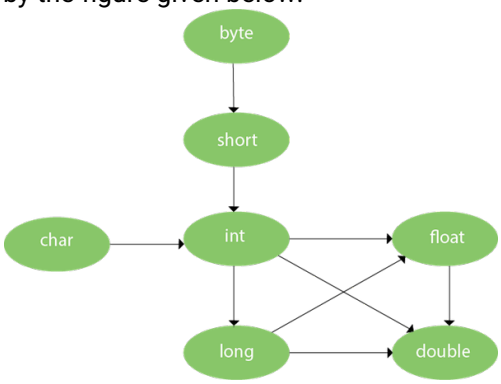
```
1. class TestOverloading4{
2. public static void main(String[] args){System.out.println("main with String[]");}
3. public static void main(String args){System.out.println("main with String");}
4. public static void main(){System.out.println("main without args");}
5. }
```

Output:

```
main with String[]
```

Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:




As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int,long,float or double. The char datatype can be promoted to int,long,float or double and so on.

Example of Method Overloading with TypePromotion

```
1. class OverloadingCalculation1{
2.   void sum(int a,long b){System.out.println(a+b);}
3.   void sum(int a,int b,int c){System.out.println(a+b+c);}
4.
5.   public static void main(String args[]){
6.   OverloadingCalculation1 obj=new OverloadingCalculation1();
7.   obj.sum(20,20);//now second int literal will be promoted to long
8.   obj.sum(20,20,20);
9.
10.  }
11. }
```

Output:40

```
     60
```

### Example of Method Overloading with Type Promotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

```
1.  class OverloadingCalculation2{
2.    void sum(int a,int b){System.out.println("int arg method invoked");}
3.    void sum(long a,long b){System.out.println("long arg method invoked");}
4.
5.    public static void main(String args[]){
6.    OverloadingCalculation2 obj=new OverloadingCalculation2();
7.    obj.sum(20,20);//now int arg sum() method gets invoked
8.    }
9.  }
```

Output:int arg method invoked

### Example of Method Overloading with Type Promotion in case of ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
1.  class OverloadingCalculation3{
2.    void sum(int a,long b){System.out.println("a method invoked");}
3.    void sum(long a,int b){System.out.println("b method invoked");}
4.
5.    public static void main(String args[]){
6.    OverloadingCalculation3 obj=new OverloadingCalculation3();
7.    obj.sum(20,20);//now ambiguity
8.    }
9.  }
```

Output:Compile Time Error

### Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

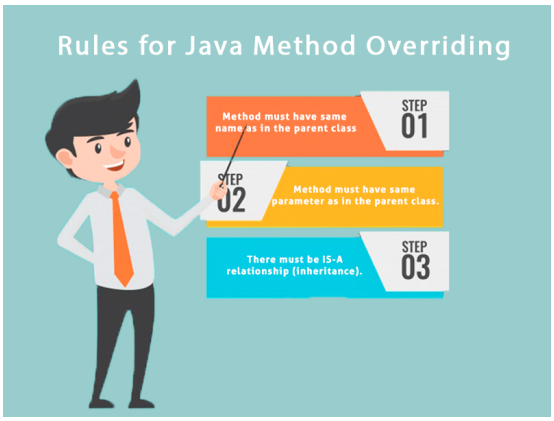
In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

### Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

### Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).




Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
1. //Java Program to demonstrate why we need method overriding
2. //Here, we are calling the method of parent class with child
3. //class object.
4. //Creating a parent class
5. class Vehicle{
6.   void run(){System.out.println("Vehicle is running");}
7. }
8. //Creating a child class
9. class Bike extends Vehicle{
10.  public static void main(String args[]){
11.  //creating an instance of child class
12.  Bike obj = new Bike();
13.  //calling the method with child class instance
14.  obj.run();
15.  }
16. }
```

Output:

```
Vehicle is running
```

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

---

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
1. //Java Program to illustrate the use of Java Method Overriding
2. //Creating a parent class.
3. class Vehicle{
4.   //defining a method
5.   void run(){System.out.println("Vehicle is running");}
6. }
```

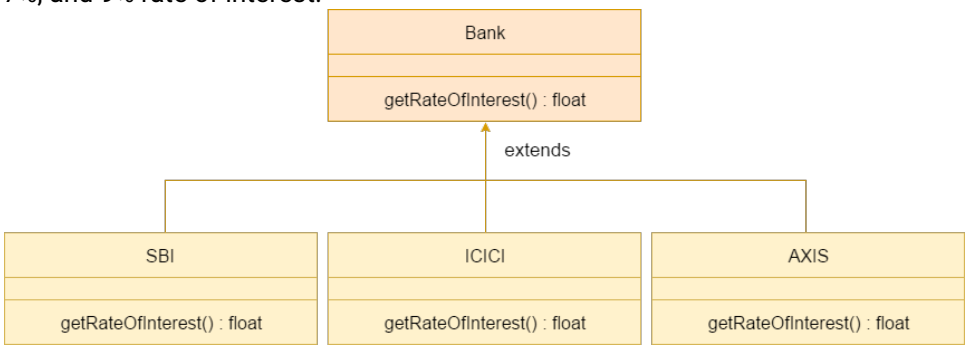
```
7.  //Creating a child class
8.  class Bike2 extends Vehicle{
9.    //defining the same method as in the parent class
10.   void run(){System.out.println("Bike is running safely");}
11.
12.   public static void main(String args[]){
13.   Bike2 obj = new Bike2();//creating object
14.   obj.run();//calling method
15.   }
16. }
```

Output:
Bike is running safely

---

A real example of Java Method Overriding

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.




Java method overriding is mostly used in Runtime Polymorphism which we will learn in next pages.

```
1.  //Java Program to demonstrate the real scenario of Java Method Overriding
2.  //where three classes are overriding the method of a parent class.
3.  //Creating a parent class.
4.  class Bank{
5.  int getRateOfInterest(){return 0;}
6.  }
7.  //Creating child classes.
8.  class SBI extends Bank{
9.  int getRateOfInterest(){return 8;}
10. }
11.
12. class ICICI extends Bank{
13. int getRateOfInterest(){return 7;}
14. }
15. class AXIS extends Bank{
```

```
16. int getRateOfInterest(){return 9;}
17. }
18. //Test class to create objects and call the methods
19. class Test2{
20. public static void main(String args[]){
21. SBI s=new SBI();
22. ICICI i=new ICICI();
23. AXIS a=new AXIS();
24. System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
25. System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
26. System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
27. }
28. }
```

```
Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

### Interface in Java

An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

### Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

### How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

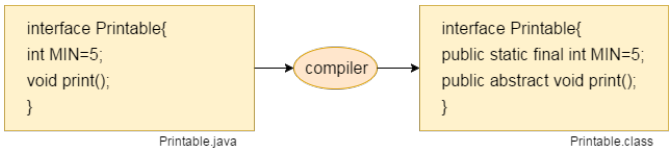
```
interface <interface_name>{

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

Java 8 Interface Improvement

Since Java 8, interface can have default and static methods which is discussed later.

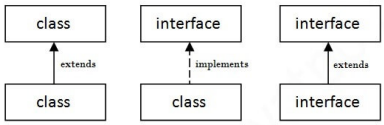
Internal addition by the compiler

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.




The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.




---

Java Interface Example

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

Output:

```
Hello
```

### Java Interface Example: Drawable

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

*File: TestInterface1.java*

```
1. //Interface declaration: by first user
2. interface Drawable{
3. void draw();
4. }
5. //Implementation: by second user
6. class Rectangle implements Drawable{
7. public void draw(){System.out.println("drawing rectangle");}
8. }
9. class Circle implements Drawable{
10. public void draw(){System.out.println("drawing circle");}
11. }
12. //Using interface: by third user
13. class TestInterface1{
14. public static void main(String args[]){
15. Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
16. d.draw();
17. }}
```

Output:

```
drawing circle
```

### Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

*File: TestInterface2.java*

```
1. interface Bank{
2. float rateOfInterest();
3. }
4. class SBI implements Bank{
5. public float rateOfInterest(){return 9.15f;}
6. }
7. class PNB implements Bank{
8. public float rateOfInterest(){return 9.7f;}
9. }
```

```
10. class TestInterface2{
11. public static void main(String[] args){
12. Bank b=new SBI();
13. System.out.println("ROI: "+b.rateOfInterest());
14. }}
```

Output:

```
ROI: 9.15
```

---

### Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.


**Multiple Inheritance in Java**

```
1. interface Printable{
2. void print();
3. }
4. interface Showable{
5. void show();
6. }
7. class A7 implements Printable,Showable{
8. public void print(){System.out.println("Hello");}
9. public void show(){System.out.println("Welcome");}
10.
11. public static void main(String args[]){
12. A7 obj = new A7();
13. obj.print();
14. obj.show();
15. }
16. }
```

Output:Hello

Welcome

### Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

```
interface Printable{
void print();
}
interface Showable{
void print();
}

class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
 }
}
```

Output:

```
Hello
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

---

Interface inheritance

A class implements an interface, but one interface extends another interface.

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
 }
}
```

Output:

```
Hello
Welcome
```

### Java 8 Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

*File: TestInterfaceDefault.java*

```
1. interface Drawable{
2. void draw();
3. default void msg(){System.out.println("default method");}
4. }
5. class Rectangle implements Drawable{
6. public void draw(){System.out.println("drawing rectangle");}
7. }
8. class TestInterfaceDefault{
9. public static void main(String args[]){
10. Drawable d=new Rectangle();
11. d.draw();
12. d.msg();
13. }}
```

Output:

```
drawing rectangle
default method
```

### Java 8 Static Method in Interface

Since Java 8, we can have static method in interface. Let's see an example:

*File: TestInterfaceStatic.java*

```
1. interface Drawable{
2. void draw();
3. static int cube(int x){return x*x*x;}
4. }
5. class Rectangle implements Drawable{
6. public void draw(){System.out.println("drawing rectangle");}
7. }
8.
9. class TestInterfaceStatic{
10. public static void main(String args[]){
11. Drawable d=new Rectangle();
12. d.draw();
13. System.out.println(Drawable.cube(3));
14. }}
```

Output:

```
drawing rectangle
```

## Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.
But there are many differences between abstract class and interface that are given below.

| Abstract class | Interface |
|---|---|
| 1) Abstract class can **have abstract and non-abstract**methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class**can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class**can be extended using keyword "extends". | An **interface class**can be implemented using keyword "implements". |
| 8) A Java**abstract class**can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:**<br>public abstract class Shape{<br>public abstract void draw();<br>} | **Example:**<br>public interface Drawable{<br>void draw();<br>} |

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

### Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

```
1. //Creating interface that has 4 methods
2. interface A{
3. void a();//bydefault, public and abstract
4. void b();
5. void c();
6. void d();
7. }
8.
9. //Creating abstract class that provides the implementation of one method of A interface
10. abstract class B implements A{
11. public void c(){System.out.println("I am C");}
12. }
13.
14. //Creating subclass of abstract class, now we need to provide the implementation of rest of the methods

15. class M extends B{
16. public void a(){System.out.println("I am a");}
17. public void b(){System.out.println("I am b");}
18. public void d(){System.out.println("I am d");}
19. }
20.
21. //Creating a test class that calls the methods of A interface
22. class Test5{
23. public static void main(String args[]){
24. A a=new M();
25. a.a();
26. a.b();
27. a.c();
28. a.d();
29. }}
```

Output:

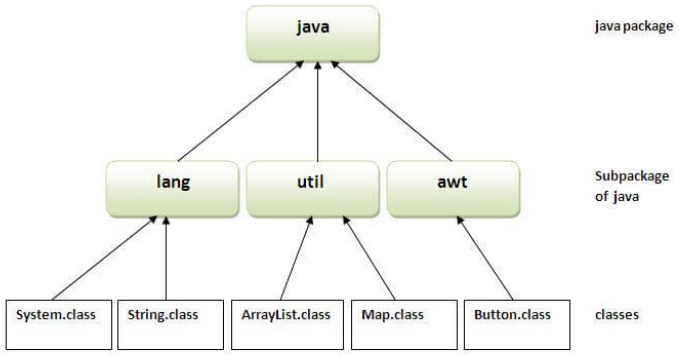
```
I am a
I am b
I am c
I am d
```

### Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.
Package in java can be categorized in two form, built-in package and user-defined package.
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
Here, we will have the detailed learning of creating and using user-defined packages.

## Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.




---

## Simple example of java package

The **package keyword** is used to create a package in java.

```
1. //save as Simple.java
2. package mypack;
3. public class Simple{
4.  public static void main(String args[]){
5.    System.out.println("Welcome to package");
6.   }
7. }
```

## How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

---

## How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

---

**To Compile:** javac -d . Simple.java

**To Run:** java mypack.Simple

```
Output:Welcome to package
```

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the cu

---

## How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
1. //save by A.java
2. package pack;
3. public class A{
4.   public void msg(){System.out.println("Hello");}
5. }
```

```
1. //save by B.java
2. package mypack;
3. import pack.*;
4.
5. class B{
6.   public static void main(String args[]){
7.    A obj = new A();
8.    obj.msg();
9.   }
10. }
```

Output:Hello

---

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
1. //save by A.java
2.
3. package pack;
4. public class A{
5.   public void msg(){System.out.println("Hello");}
6. }
```

```
1. //save by B.java
2. package mypack;
3. import pack.A;
4.
5. class B{
6.   public static void main(String args[]){
7.    A obj = new A();
8.    obj.msg();
9.   }
```

```
10. }
```
Output:Hello

---

### 3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
1. //save by A.java
2. package pack;
3. public class A{
4.   public void msg(){System.out.println("Hello");}
5. }
```

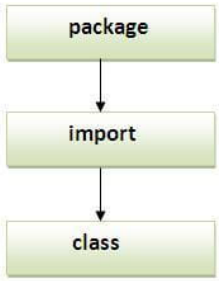
```
1. //save by B.java
2. package mypack;
3. class B{
4.   public static void main(String args[]){
5.    pack.A obj = new pack.A();//using fully qualified name
6.    obj.msg();
7.   }
8. }
```

Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.




### Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.
Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and

put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

Example of Subpackage

```
1. package com.javatpoint.core;
2. class Simple{
3.   public static void main(String args[]){
4.    System.out.println("Hello subpackage");
5.   }
6. }
```

**To Compile:** javac -d . Simple.java

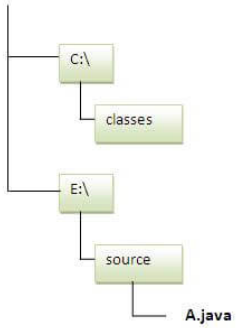
**To Run:** java com.javatpoint.core.Simple

Output:Hello subpackage

---

How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:




```
1. //save as Simple.java
2. package mypack;
3. public class Simple{
4.  public static void main(String args[]){
5.    System.out.println("Welcome to package");
6.   }
7. }
```

To Compile:

**e:\sources> javac -d c:\classes Simple.java**

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

**e:\sources> set classpath=c:\classes;.;**

**e:\sources> java mypack.Simple**

### Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

```
e:\sources> java -classpath c:\classes mypack.Simple
```

Output:Welcome to package

---

### Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

- Temporary
  - By setting the classpath in the command prompt
  - By -classpath switch
- Permanent
  - By setting the classpath in the environment variables
  - By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

### Access Modifiers in java

There are two types of modifiers in java: **access modifiers** and **non-access modifiers**.

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc. Here, we will

---

#### 1) private access modifier

The private access modifier is accessible only within class.

#### Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private meth

private members from outside the class, so there is compile time error.

```
1.  class A{
2.  private int data=40;
3.  private void msg(){System.out.println("Hello java");}
4.  }
5.
6.  public class Simple{
7.   public static void main(String args[]){
8.     A obj=new A();
```

```
9.    System.out.println(obj.data);//Compile Time Error
10.   obj.msg();//Compile Time Error
11.   }
12. }
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For e

```
1.  class A{
2.  private A(){}//private constructor
3.  void msg(){System.out.println("Hello java");}
4.  }
5.  public class Simple{
6.   public static void main(String args[]){
7.    A obj=new A();//Compile Time Error
8.   }
9.  }
```

Note: A class cannot be private or protected except nested class.

---

2) default access modifier

If you don't use any modifier, it is treated as **default** bydefault. The default modifier is accessible only within packag

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its pa

public, so it cannot be accessed from outside the package.

```
1.  //save by A.java
2.  package pack;
3.  class A{
4.   void msg(){System.out.println("Hello");}
5.  }
1.  //save by B.java
2.  package mypack;
3.  import pack.*;
4.  class B{
5.   public static void main(String args[]){
6.    A obj = new A();//Compile Time Error
7.    obj.msg();//Compile Time Error
8.   }
9.  }
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the p

---

### 3) protected access modifier

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the c

#### Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be

package. But msg method of this package is declared as protected, so it can be accessed from outside the class only t

```
1.  //save by A.java
2.  package pack;
3.  public class A{
4.  protected void msg(){System.out.println("Hello");}
5.  }
1.  //save by B.java
2.  package mypack;
3.  import pack.*;
4.
5.  class B extends A{
6.    public static void main(String args[]){
7.     B obj = new B();
8.     obj.msg();
9.    }
10. }
```

Output:Hello

---

### 4) public access modifier

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

#### Example of public access modifier

```
1.  //save by A.java
2.
3.  package pack;
4.  public class A{
5.  public void msg(){System.out.println("Hello");}
6.  }
1.  //save by B.java
2.
3.  package mypack;
4.  import pack.*;
5.
6.  class B{
7.    public static void main(String args[]){
8.     A obj = new A();
```

```
9.    obj.msg();
10.  }
11. }
```

Output:Hello

---

## Understanding all java access modifiers

Let's understand the access modifiers by a simple table.

| Access Modifier | within class | within package | outside package by subclass only | outside package |
|---|---|---|---|---|
| Private | Y | N | N | N |
| Default | Y | Y | N | N |
| Protected | Y | Y | Y | N |
| Public | Y | Y | Y | Y |

---

## Java access modifiers with method overriding

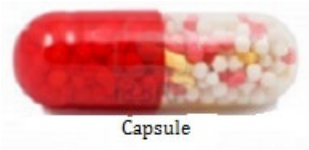
If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
1.  class A{
2.  protected void msg(){System.out.println("Hello java");}
3.  }
4.
5.  public class Simple extends A{
6.  void msg(){System.out.println("Hello java");}//C.T.Error
7.   public static void main(String args[]){
8.    Simple obj=new Simple();
9.    obj.msg();
10.  }
11. }
```

The default modifier is more restrictive than protected. That is why there is compile time error.

## Encapsulation in Java

**Encapsulation in Java** is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.



Capsule

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.
The **Java Bean** class is the example of a fully encapsulated class.

### Advantage of Encapsulation in Java

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.
It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.
It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.
The encapsulate class is **easy to test**. So, it is better for unit testing.
The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

### Simple Example of Encapsulation in Java

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.
*File: Student.java*

```
1.  //A Java class which is a fully encapsulated class.
2.  //It has a private data member and getter and setter methods.
3.  package com.javatpoint;
4.  public class Student{
5.  //private data member
6.  private String name;
7.  //getter method for name
8.  public String getName(){
9.  return name;
10. }
11. //setter method for name
12. public void setName(String name){
13. this.name=name
14. }
15. }
```

*File: Test.java*

```
1.  //A Java class to test the encapsulated class.
2.  package com.javatpoint;
3.  class Test{
4.  public static void main(String[] args){
5.  //creating instance of the encapsulated class
6.  Student s=new Student();
7.  //setting value in the name member
8.  s.setName("vijay");
```

```
9.  //getting value of the name member
10. System.out.println(s.getName());
11. }
12. }
```

Compile By: javac -d . Test.java
Run By: java com.javatpoint.Test
Output:
vijay

Read-Only class

```
1.  //A Java class which has only getter methods.
2.  public class Student{
3.  //private data member
4.  private String college="AKG";
5.  //getter method for college
6.  public String getCollege(){
7.  return college;
8.  }
9.  }
```

Now, you can't change the value of the college data member which is "AKG".

```
1.  s.setCollege("KITE");//will render compile time error
```

Write-Only class

```
1.  //A Java class which has only setter methods.
2.  public class Student{
3.  //private data member
4.  private String college;
5.  //getter method for college
6.  public void setCollege(String college){
7.  this.college=college;
8.  }
9.  }
```

Now, you can't get the value of the college, you can only change the value of college data member.

```
1.  System.out.println(s.getCollege());//Compile Time Error, because there is no such method
2.  System.out.println(s.college);//Compile Time Error, because the college data member is private.
3.  //So, it can't be accessed from outside the class
```

Another Example of Encapsulation in Java

Let's see another example of encapsulation that has only four fields with its setter and getter methods.

*File: Account.java*

```
1.  //A Account class which is a fully encapsulated class.
2.  //It has a private data member and getter and setter methods.
3.  class Account {
4.  //private data members
5.  private long acc_no;
6.  private String name,email;
```

```
7.  private float amount;
8.  //public getter and setter methods
9.  public long getAcc_no() {
10.     return acc_no;
11. }
12. public void setAcc_no(long acc_no) {
13.     this.acc_no = acc_no;
14. }
15. public String getName() {
16.     return name;
17. }
18. public void setName(String name) {
19.     this.name = name;
20. }
21. public String getEmail() {
22.     return email;
23. }
24. public void setEmail(String email) {
25.     this.email = email;
26. }
27. public float getAmount() {
28.     return amount;
29. }
30. public void setAmount(float amount) {
31.     this.amount = amount;
32. }
33.
34. }
```

*File: TestAccount.java*

```
1.  //A Java class to test the encapsulated class Account.
2.  public class TestEncapsulation {
3.  public static void main(String[] args) {
4.      //creating instance of Account class
5.      Account acc=new Account();
6.      //setting values through setter methods
7.      acc.setAcc_no(7560504000L);
8.      acc.setName("Sonoo Jaiswal");
9.      acc.setEmail("sonoojaiswal@javatpoint.com");
10.     acc.setAmount(500000f);
11.     //getting values through getter methods
12.     System.out.println(acc.getAcc_no()+" "+acc.getName()+" "+acc.getEmail()+" "+acc.getAmount());
```

13. }
14. }

Output:

```
7560504000 Sonoo Jaiswal sonoojaiswal@javatpoint.com 500000.0
```