

# What is Java

Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language.

Java was developed by *Sun Microsystems* (which is now the subsidiary of Oracle) in the year 1995. *James Gosling* is known as the father of Java. Before Java, its name was *Oak*. Since Oak was already a registered company, so James Gosling and his team changed the Oak name to Java

## 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.

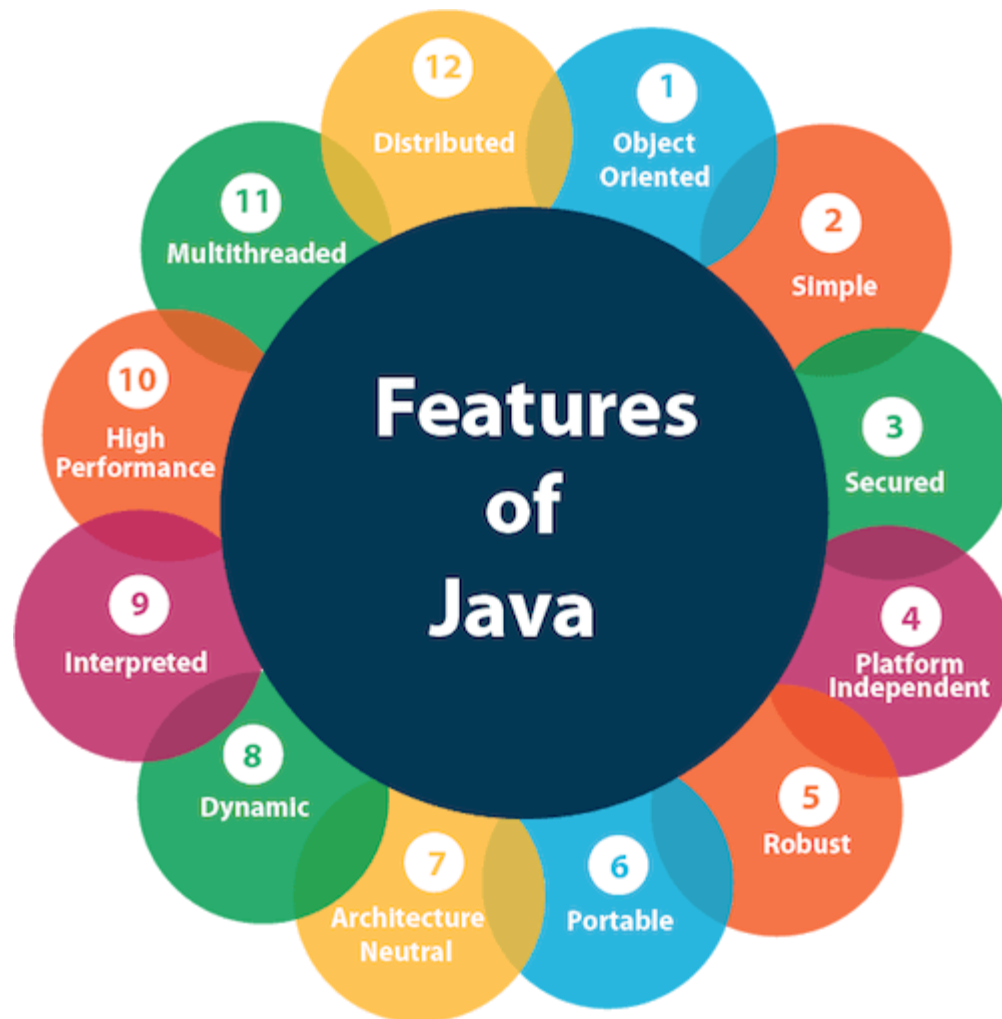
## *4) JavaFX*

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

# 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.

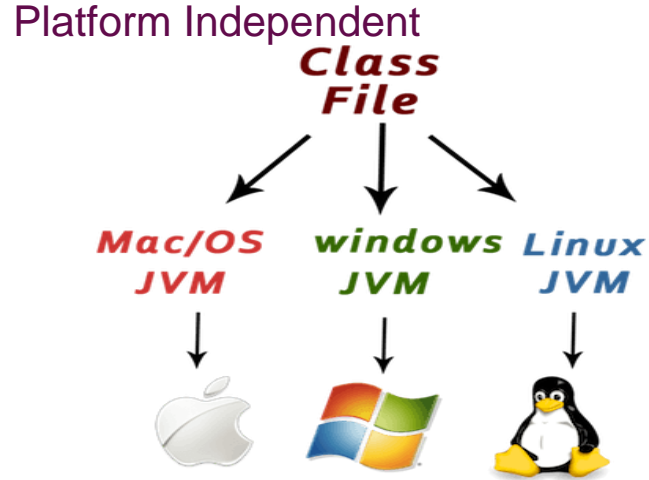
## Object-oriented

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



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:

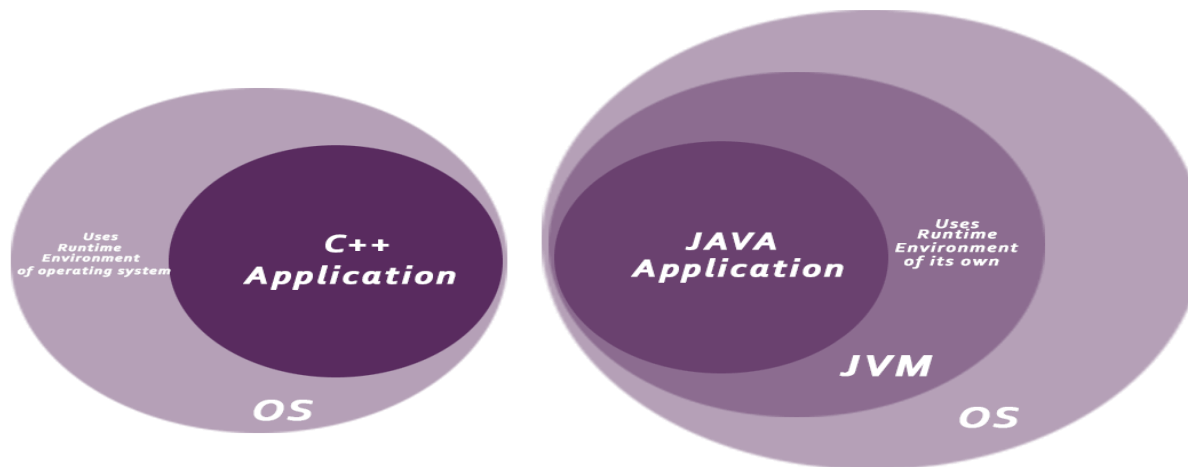
1. Runtime Environment
2. API(Application Programming Interface)

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).

## C++ vs Java

There are many differences and similarities between the [C++ programming](#) language and [Java](#). A list of top differences between C++ and Java are given below:

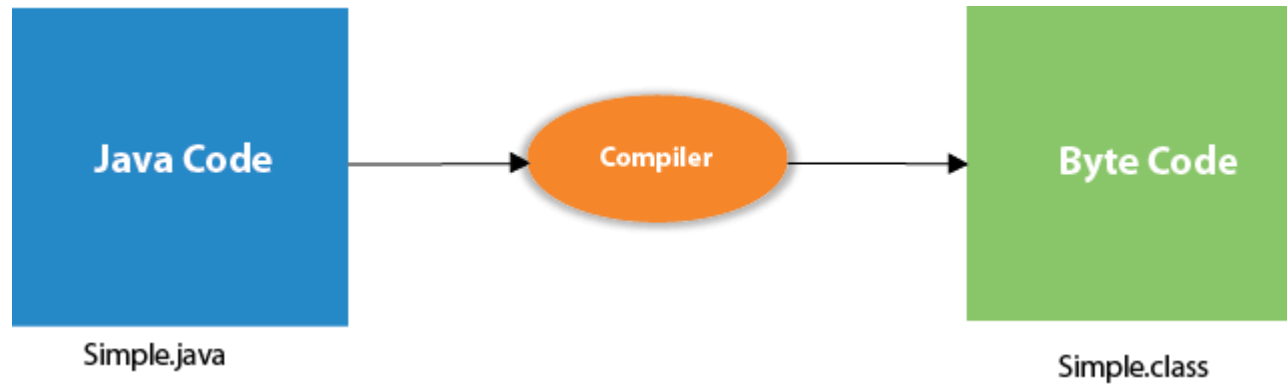
Comparison Index	C++	Java
<b>Platform-independent</b>	C++ is platform-dependent.	Java is platform-independent.
<b>Mainly used for</b>	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications.
<b>Design Goal</b>	C++ was designed for systems and applications programming. It was an extension of <a href="#">C programming language</a> .	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed with a goal of being easy to use and accessible to a broader audience.

<b>Goto</b>	C++ supports the <a href="#">goto</a> statement.	Java doesn't support the goto statement.
<b>Multiple inheritance</b>	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by <a href="#">interfaces in java</a> .
<b>Operator Overloading</b>	C++ supports <a href="#">operator overloading</a> .	Java doesn't support operator overloading.
<b>Pointers</b>	C++ supports <a href="#">pointers</a> . You can write pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.
<b>Compiler and Interpreter</b>	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.	Java uses compiler and interpreter both. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform independent.
<b>Call by Value and Call by reference</b>	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
<b>Structure and Union</b>	C++ supports structures and unions.	Java doesn't support structures and unions.

<b>Thread Support</b>	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in <a href="#">thread</a> support.
<b>Documentation comment</b>	C++ doesn't support documentation comment.	Java supports documentation comment ( <code>/** ... */</code> ) to create documentation for java source code.
<b>Virtual Keyword</b>	C++ supports virtual keyword so that we can decide whether or not override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
<b>unsigned right shift &gt;&gt;&gt;</b>	C++ doesn't support >>> operator.	Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator.
<b>Inheritance Tree</b>	C++ creates a new inheritance tree always.	Java uses a single inheritance tree always because all classes are the child of Object class in java. The object class is the root of the <a href="#">inheritance</a> tree in java.
<b>Hardware</b>	C++ is nearer to hardware.	Java is not so interactive with hardware.

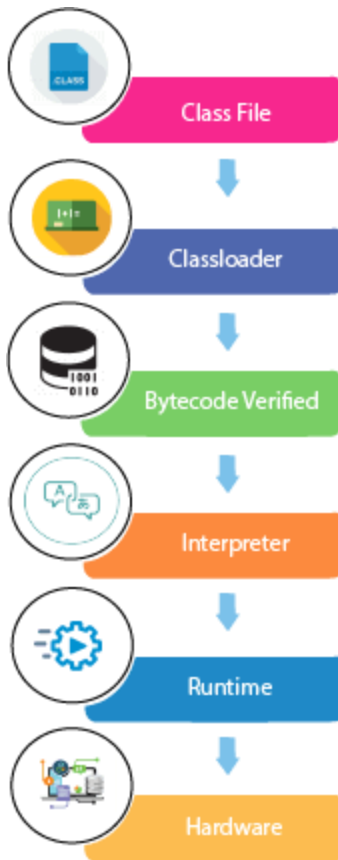
## What happens at compile time?

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



## 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.

# Difference between JDK, JRE, and JVM

1. [A summary of JVM](#)
2. [Java Runtime Environment \(JRE\)](#)
3. [Java Development Kit \(JDK\)](#)

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*.

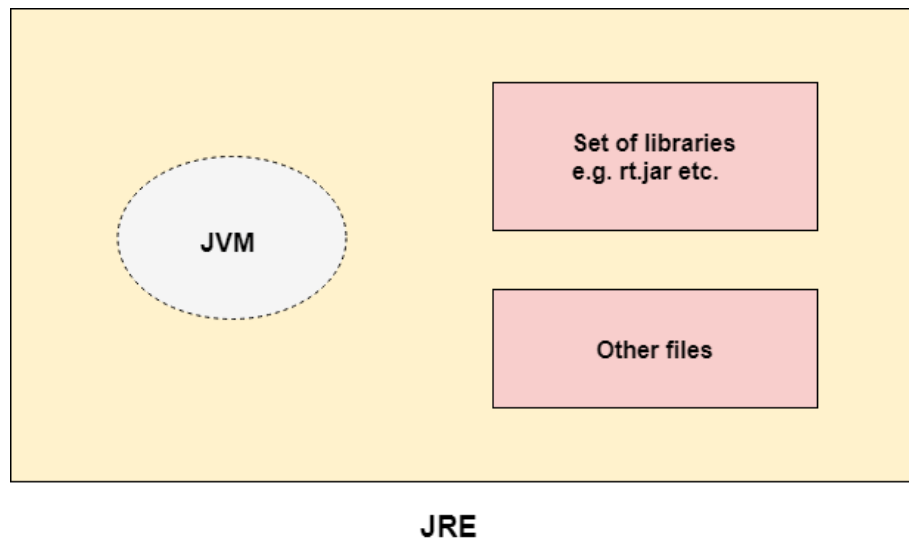
The JVM performs the following main tasks:

- 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

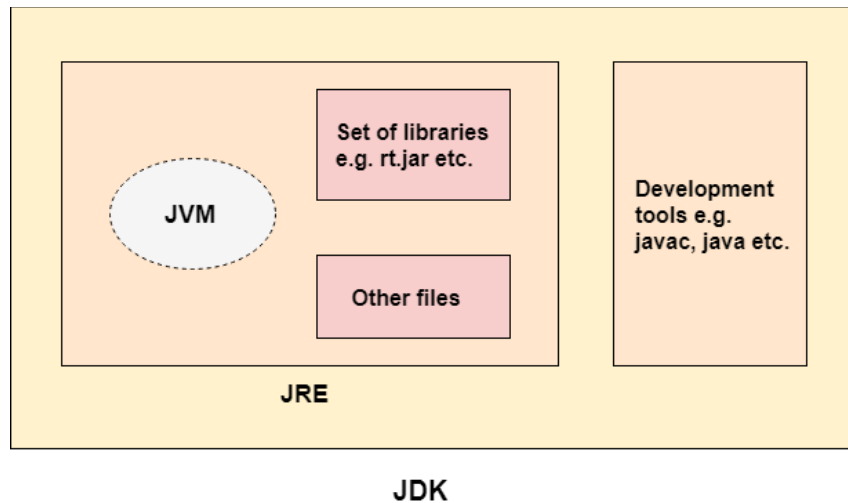
JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and [applets](#). It physically exists. It contains JRE + development tools.

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.



# 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:

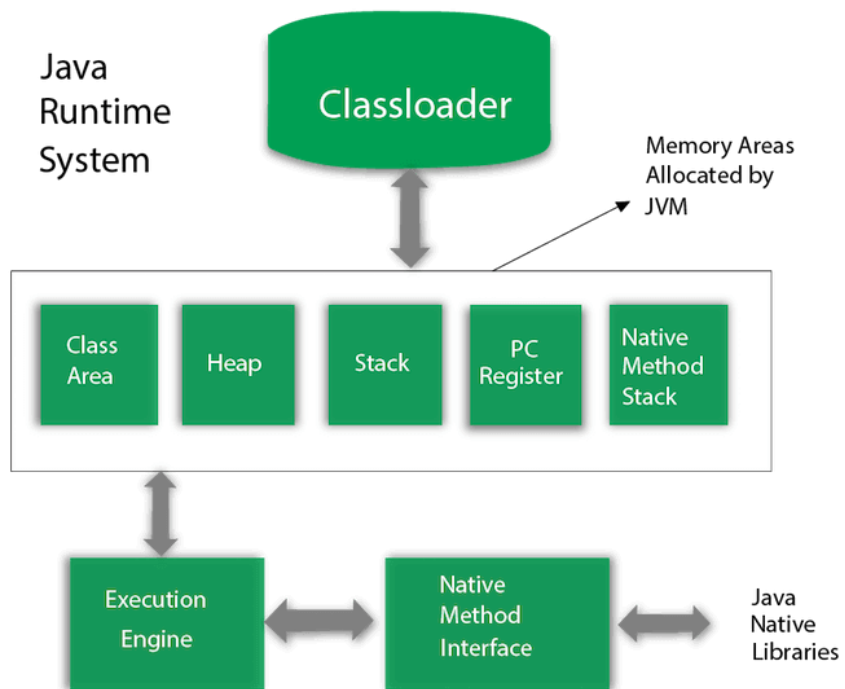
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

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 loads 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.

## 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.

# Operators in Java

**Operator** in [Java](#) is a symbol which 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	<code>expr++ expr--</code>
	prefix	<code>++expr --expr +expr -expr ~ !</code>
Arithmetic	multiplicative	<code>* / %</code>

	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
6.      System.out.println(b++ + b++);//10+11=21
7.
8.      }}
```

Output:

```
22
21
```

## Java Unary Operator Example: ~ and !

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

Output:

```
-11
9
false
true
```

## Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

### Java Arithmetic Operator Example

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

Output:

```
15
5
50
2
0
```

### Java Arithmetic Operator Example: Expression

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

Output:

## 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

```
1.  class OperatorExample{
2.  public static void main(String args[]){
3.      System.out.println(10<<2);//10*2^2=10*4=40
4.      System.out.println(10<<3);//10*2^3=10*8=80
5.      System.out.println(20<<2);//20*2^2=20*4=80
6.      System.out.println(15<<4);//15*2^4=15*16=240
7.  }}
```

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 operand.

### Java Right Shift Operator Example

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

```
5.     System.out.println(20>>3);//20/2^3=20/8=2
6.     }}
```

Output:

```
2
5
2
```

## Java Shift Operator Example: >> vs >>>

```
1.     class OperatorExample{
2.     public static void main(String args[]){
3.         //For positive number, >> and >>> works same
4.         System.out.println(20>>2);
5.         System.out.println(20>>>2);
6.         //For negative number, >>> changes parity bit (MSB) to 0
7.         System.out.println(-20>>2);
8.         System.out.println(-20>>>2);
9.     }}
```

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 one is true.

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);//false && true = false
7.  System.out.println(a<b&a<c);//false & true = false
8.  }}
```

Output:

```
false
false
```

## Java AND Operator Example: Logical && vs Bitwise &

```
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);//false && true = false
7.  System.out.println(a);//10 because second condition is not checked
8.  System.out.println(a<b&a++<c);//false && true = false
9.  System.out.println(a);//11 because second condition is checked
10. }
```

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 one is false.

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. it is the only conditional 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:

```
1.    class OperatorExample{  
2.    public static void main(String args[]){  
3.    int a=10;  
4.    int b=5;  
5.    int min=(a<b)?a:b;  
6.    System.out.println(min);  
7.    }}
```

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 operand on its left.

### Java Assignment Operator Example

```
1.  class OperatorExample{
2.  public static void main(String args[]){
3.  int a=10;
4.  int b=20;
5.  a+=4;//a=a+4 (a=10+4)
6.  b-=4;//b=b-4 (b=20-4)
7.  System.out.println(a);
8.  System.out.println(b);
9.  }}
```

Output:

```
14
16
```

### Java Assignment Operator Example

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



```
9.      System.out.println(a);
10.     a/=2;//18/2
11.     System.out.println(a);
12.     }}
```

Output:

```
13
9
18
9
```

## Java Assignment Operator Example: Adding short

```
1.      class OperatorExample{
2.      public static void main(String args[]){
3.      short a=10;
4.      short b=10;
5.      //a+=b;//a=a+b internally so fine
6.      a=a+b;//Compile time error because 10+10=20 now int
7.      System.out.println(a);
8.      }}
```

Output:

```
Compile time error
```

After type cast:

```
1.      class OperatorExample{
2.      public static void main(String args[]){
3.      short a=10;
4.      short b=10;
```

```
5.      a=(short)(a+b);//20 which is int now converted to short
6.      System.out.println(a);
7.      }}
```

Output:

```
20
```

## Java Keywords

**Java keywords** are also known as **reserved words**. Keywords are particular words which acts as a key to a code. These are predefined words by Java so it cannot be used as a variable or object name.

### List of Java Keywords

A list of Java keywords or reserved words are given below:

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.

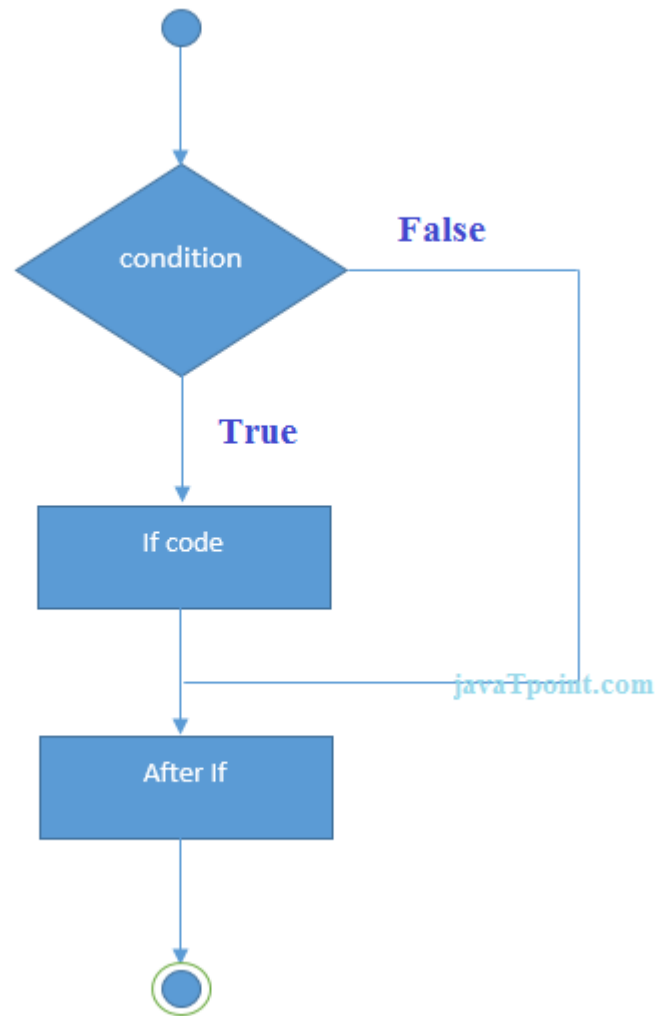
- if statement
- if-else statement
- if-else-if ladder
- nested if statement

## 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:

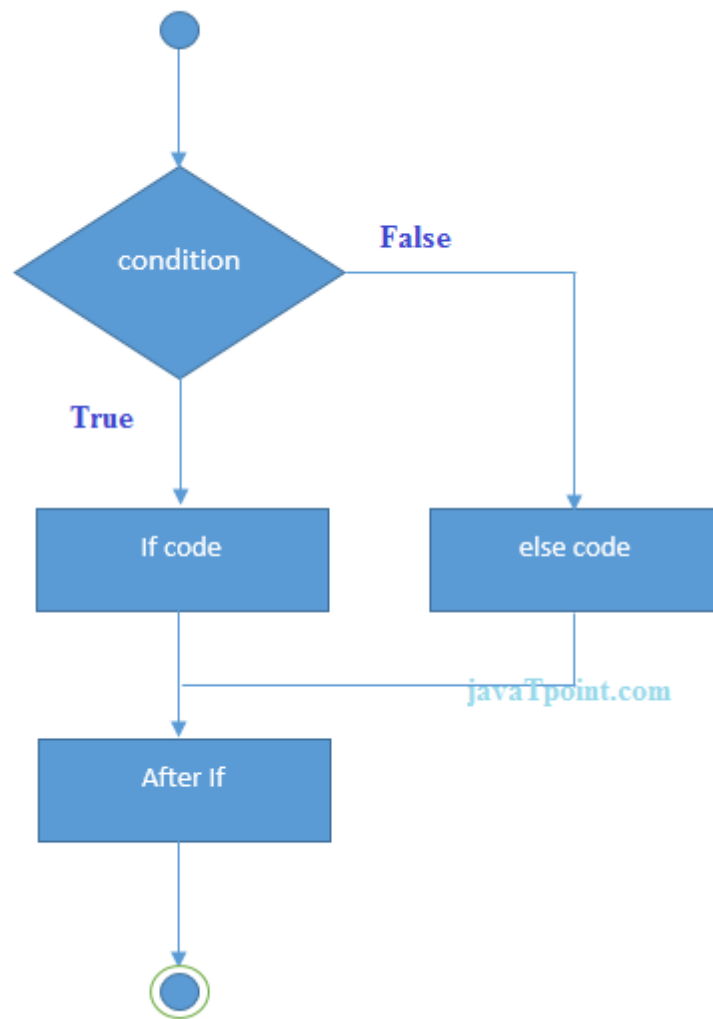
```
Age is greater than 18
```

## 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
```

### Leap Year Example:

A year is leap, if it is divisible by 4 and 400. But, not by 100.

```
1.      public class LeapYearExample {
2.      public static void main(String[] args) {
3.          int year=2020;
4.          if(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0)){
5.              System.out.println("LEAP YEAR");
6.          }
7.          else{
```

```
8.         System.out.println("COMMON YEAR");
9.     }
10. }
11. }
```

Output:

```
LEAP YEAR
```

## Using Ternary Operator

We can also use ternary operator (? :) to perform the task of if...else statement. It is a shorthand way to check the condition. If the condition is true, the result of ? is returned. But, if the condition is false, the result of : is returned.

**Example:**

```
1.     public class IfElseTernaryExample {
2.     public static void main(String[] args) {
3.         int number=13;
4.         //Using ternary operator
5.         String output=(number%2==0)?"even number":"odd number";
6.         System.out.println(output);
7.     }
8. }
```

Output:

```
odd number
```

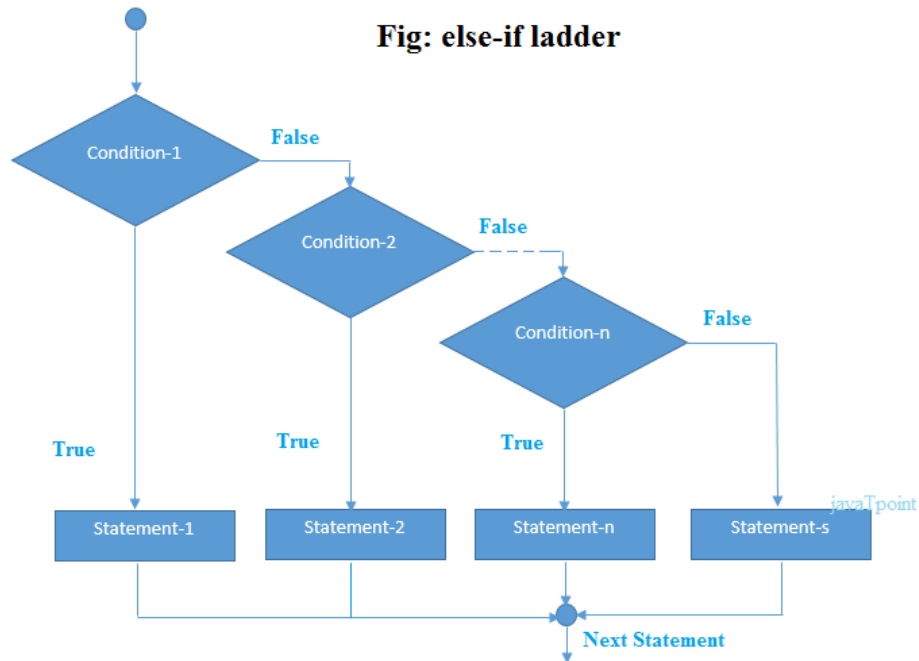
# Java if-else-if ladder Statement

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:

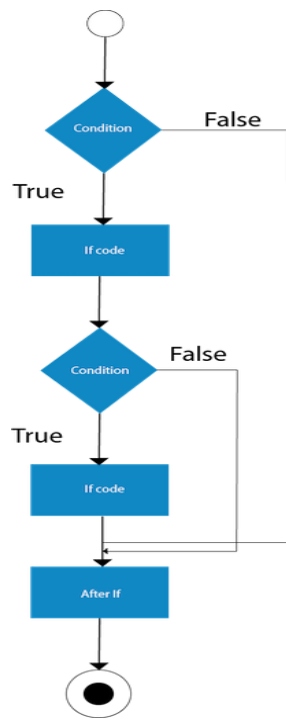
```
C grade
```

# 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:

```
1.      if(condition){
2.          //code to be executed
3.          if(condition){
4.              //code to be executed
5.          }
6.      }
```



### Example:

1. `//Java Program to demonstrate the use of Nested If Statement.`
2. `public class JavaNestedIfExample {`
3. `public static void main(String[] args) {`
4. `//Creating two variables for age and weight`
5. `int age=20;`
6. `int weight=80;`
7. `//applying condition on age and weight`
8. `if(age>=18){`

```
9.         if(weight>50){
10.             System.out.println("You are eligible to donate blood");
11.         }
12.     }
13. }}
```

Output:

```
You are 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*.

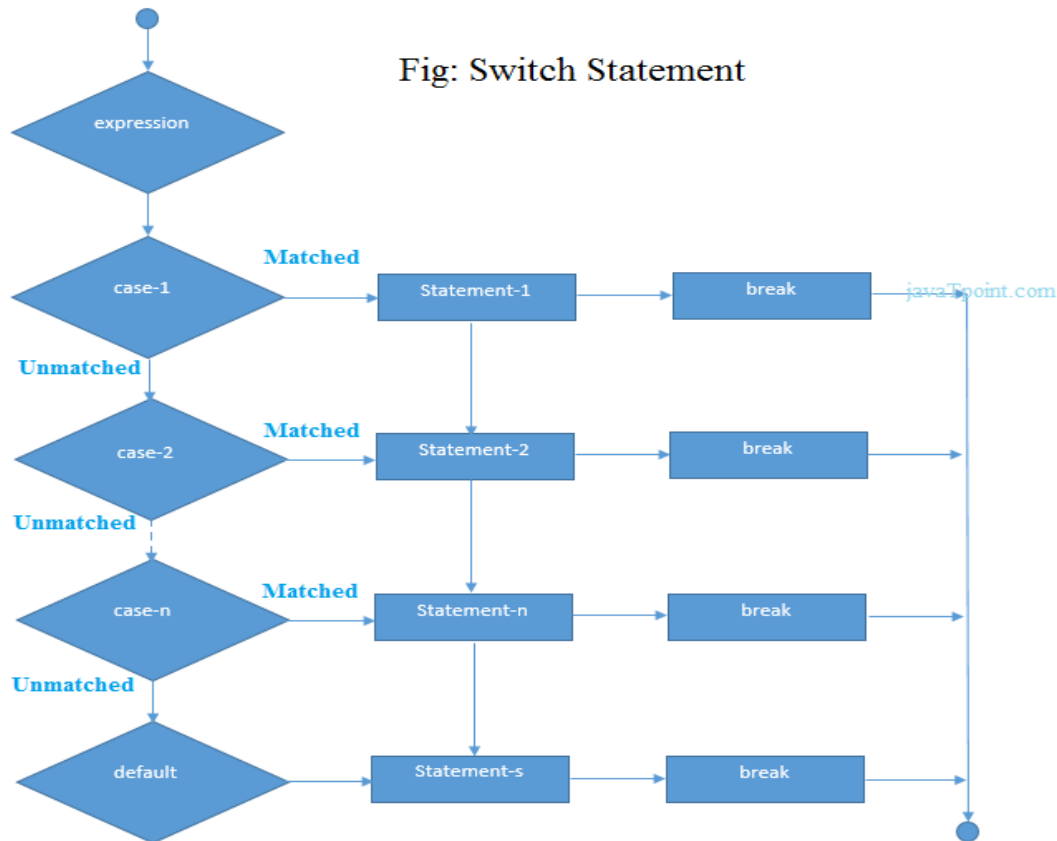


- Each case statement can have a *break statement* which is optional. When control reaches to the break statement, it jumps the control after the switch expression. If a break statement is not found, it executes the next case.
- The case value can have a *default label* which is optional.

**Syntax:**

```
1.  switch(expression){
2.  case value1:
3.    //code to be executed;
4.    break; //optional
5.  case value2:
6.    //code to be executed;
7.    break; //optional
8.  .....
9.
10. default:
11.   code to be executed if all cases are not matched;
12. }
```

Fig: Switch Statement



**Example:**

1. **public class** SwitchExample {
2. **public static void** main(String[] args) {
3.     //Declaring a variable for switch expression
4.     **int** number=20;
5.     //Switch expression

```
6.      switch(number){
7.      //Case statements
8.      case 10: System.out.println("10");
9.      break;
10.     case 20: System.out.println("20");
11.     break;
12.     case 30: System.out.println("30");
13.     break;
14.     //Default case statement
15.     default: System.out.println("Not in 10, 20 or 30");
16.     }
17. }
18. }
```

Output:

```
20
```

## 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':
20.                             System.out.println("Micro processors, Logic switching theory");
21.                             break;
22.                         case 'M':
23.                             System.out.println("Drawing, Manufacturing Machines");
```

```
24.         break;
25.     }
26.     break;
27. case 3:
28.     switch( branch )
29.     {
30.         case 'C':
31.             System.out.println("Computer Organization, MultiMedia");
32.             break;
33.         case 'E':
34.             System.out.println("Fundamentals of Logic Design, Microelectronics");
35.             break;
36.         case 'M':
37.             System.out.println("Internal Combustion Engines, Mechanical Vibration");
38.             break;
39.     }
40.     break;
41. case 4:
42.     switch( branch )
43.     {
44.         case 'C':
45.             System.out.println("Data Communication and Networks, MultiMedia");
46.             break;
47.         case 'E':
48.             System.out.println("Embedded System, Image Processing");
49.             break;
50.         case 'M':
51.             System.out.println("Production Technology, Thermal Engineering");
52.             break;
```

```

53.         }
54.         break;
55.     }
56. }
57. }

```

Output:

Data Communication and Networks, MultiMedia

## 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 <a href="#">programs</a> 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	If the number of iteration is not fixed and you must have to execute the loop at least

		to use while loop.	once, it is recommended to use the do-while loop.
Syntax	<pre>for(init;condition;incr/decr){ // code to be executed }</pre>	<pre>while(condition){ //code to be executed }</pre>	<pre>do{ //code to be executed }while(condition);</pre>
Example	<pre>//for loop for(int i=1;i&lt;=10;i++){ System.out.println(i); }</pre>	<pre>//while loop int i=1; while(i&lt;=10){ System.out.println(i); i++; }</pre>	<pre>//do-while loop int i=1; do{ System.out.println(i); i++; }while(i&lt;=10);</pre>
Syntax for infinitive loop	<pre>for(;;){ //code to be executed }</pre>	<pre>while(true){ //code to be executed }</pre>	<pre>do{ //code to be executed }while(true);</pre>

## Java For Loop

The 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 for loop.

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

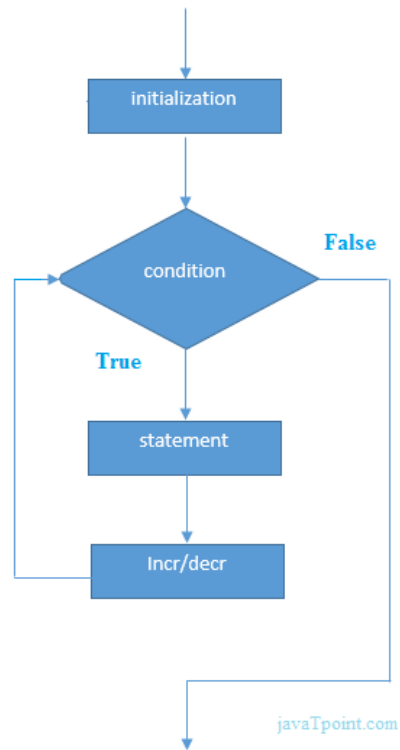
A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

1. **Initialization:** It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2. **Condition:** It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
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:

1. `for(initialization;condition;incr/decr){`
2. `//statement or code to be executed`
3. `}`

### 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.     }
```

**Test it Now**

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 increment value and use subscript notation.

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

**Syntax:**

```
1.     for(Type var:array){
2.         //code to be executed
3.     }
```

**Example:**

```
1.     //Java For-each loop example which prints the
```

```
2.      //elements of the array
3.      public class ForEachExample {
4.      public static void main(String[] args) {
5.          //Declaring an array
6.          int arr[]={12,23,44,56,78};
7.          //Printing array using for-each loop
8.          for(int i:arr){
9.              System.out.println(i);
10.         }
11.     }
12. }
```

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 that we can break/continue specific for loop.

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

### Syntax:

```
1.      labelname:
2.      for(initialization;condition;incr/decr){
```

```
3.      //code to be executed
4.      }
```

### Example:

```
1.      //A Java program to demonstrate the use of labeled for loop
2.      public class LabeledForExample {
3.      public static void main(String[] args) {
4.          //Using Label for outer and for loop
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.                             break aa;
11.                         }
12.                         System.out.println(i+ " "+j);
13.                     }
14.                 }
15.             }
16.         }
```

### 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.

```
1.      public class LabeledForExample2 {
2.      public static void main(String[] args) {
3.          aa:
4.          for(int i=1;i<=3;i++){
5.              bb:
6.              for(int j=1;j<=3;j++){
7.                  if(i==2&&j==2){
8.                      break bb;
9.                  }
10.                 System.out.println(i+ " "+j);
11.             }
12.         }
13.     }
14. }
```

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:**

```
1.     for(;;){
2.         //code to be executed
3.     }
```

#### Example:

```
1.         //Java program to demonstrate the use of infinite for loop
2.         //which prints an statement
3.     public class ForExample {
4.     public static void main(String[] args) {
5.         //Using no condition in for loop
6.         for(;;){
7.             System.out.println("infinitive loop");
8.         }
9.     }
10. }
```

#### Output:

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

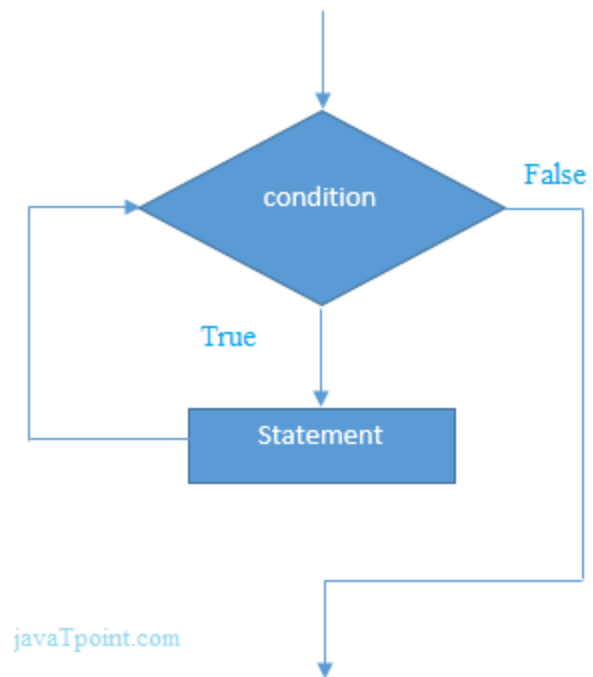
Now, you need to press ctrl+c to exit from the program.

# 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:

1. **while**(condition){
2. *//code to be executed*
3. }





### 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.    }
```

## 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.

Output:

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

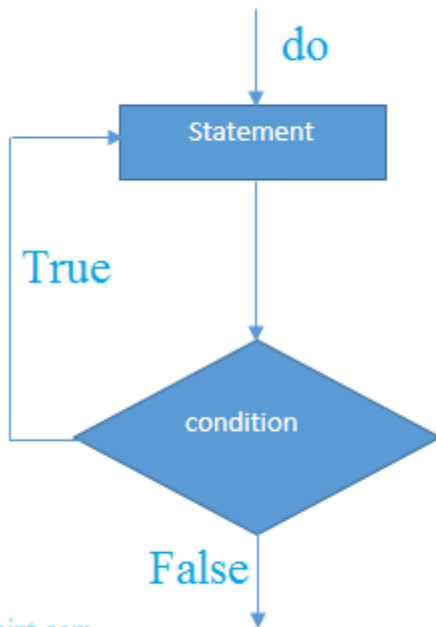
## Java do-while Loop

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:**

1. **do**{
2.   //code to be executed
3. }**while**(condition);



javaTpoint.com

**Example:**

1. **public class** DoWhileExample {
2.   **public static void** main(String[] args) {
3.     **int** i=1;

```
4.      do{
5.          System.out.println(i);
6.          i++;
7.      }while(i<=10);
8.  }
9.  }
```

**Test it Now**

Output:

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

## Java do-while Loop

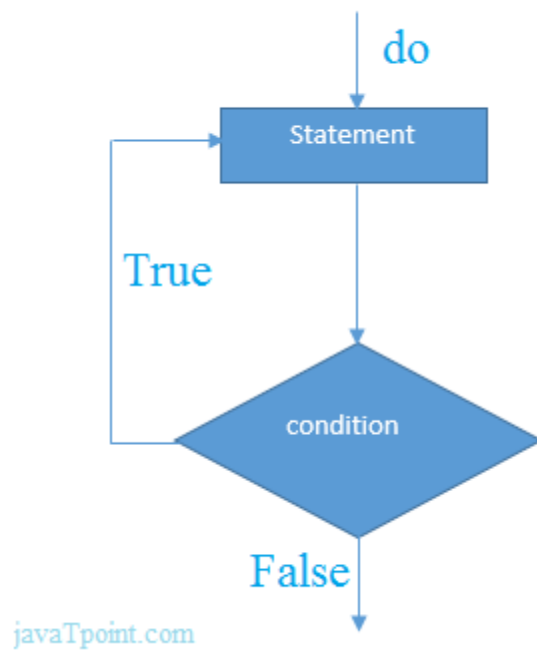
The Java *do-while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

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

**Syntax:**

```
1.      do{
2.          //code to be executed
```

3.        } **while**(condition);



#### Example:

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

### Test it Now

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:

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

### Example:

```
1.      public class DoWhileExample2 {
2.      public static void main(String[] args) {
3.      do{
4.      System.out.println("infinitive do while loop");
5.      }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

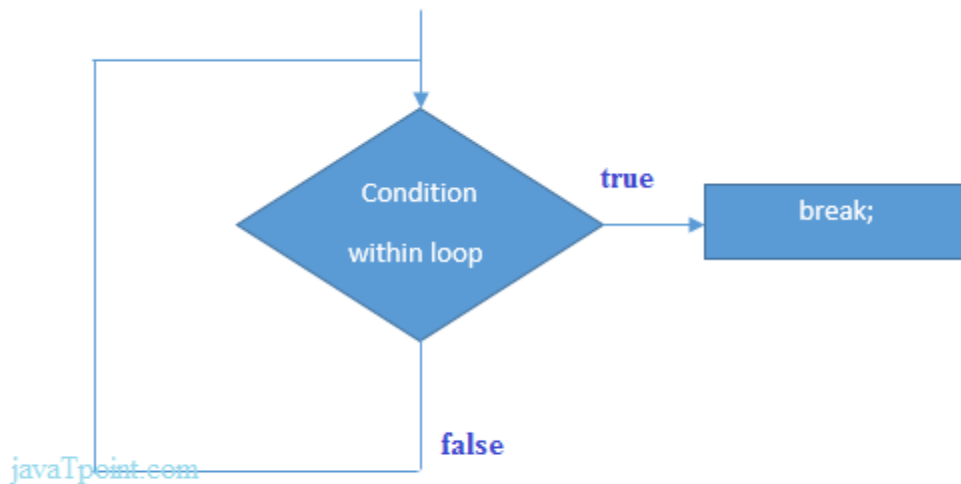
When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

The Java *break* statement 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 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
2.    //inside the for loop.
3.    public class ContinueExample {
4.    public static void main(String[] args) {
5.        //for loop
6.        for(int i=1;i<=10;i++){
7.            if(i==5){
8.                //using continue statement
9.                continue;//it will skip the rest statement
10.        }
11.        System.out.println(i);
12.    }
13.    }
14.    }
```

## Test it Now

## 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:

```
1.      //Java Program to illustrate the use of continue statement
2.      //inside an inner loop
3.      public class ContinueExample2 {
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 continue statement inside inner loop
11.                     continue;
12.                 }
13.                 System.out.println(i+ " "+j);
14.             }
15.         }
16.     }
17. }
```

Output:

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

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

# Java Comments

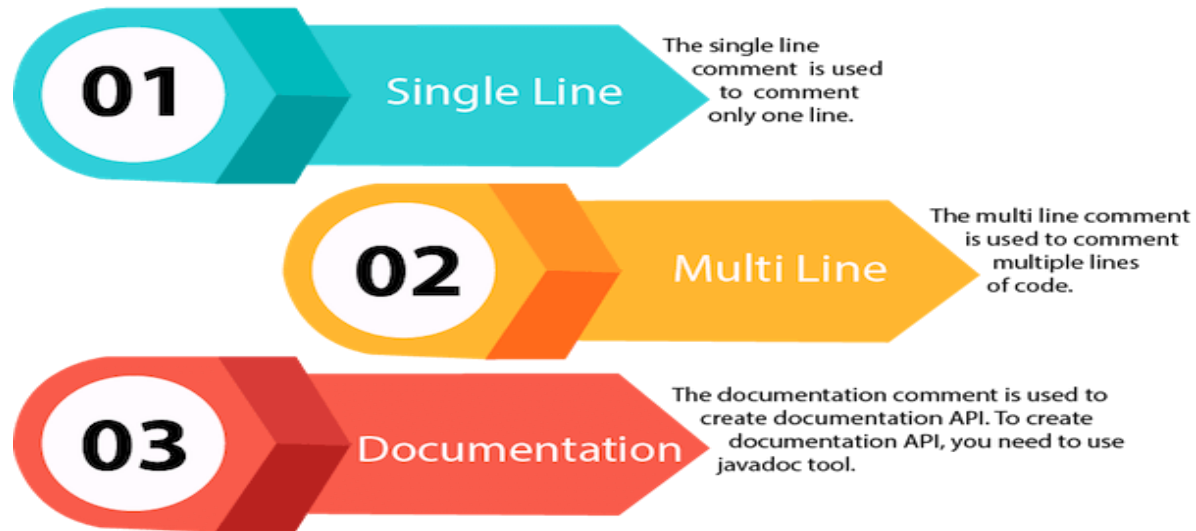
The [Java](#) comments are the statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the [variable](#), method, [class](#) or any statement. It can also be used to hide program code.

## Types of Java Comments

There are three types of comments in Kava.

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

## Types of Java Comments



### 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
```

# Java Arrays

Normally, an array is a collection of similar type of elements which has contiguous memory location.

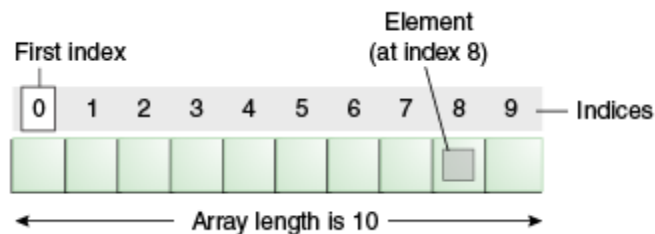
**Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimensional or multidimensional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



## Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

## Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

# Types of Array in java

There are two types of array.

- Single Dimensional Array
  - Multidimensional Array
- 

## Single Dimensional Array in Java

### Syntax to Declare an Array in Java

1.        dataType[] arr; (or)
2.        dataType []arr; (or)
3.        dataType arr[];

### Instantiation of an Array in Java

1.        arrayRefVar=**new** datatype[size];

## Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

1.        **//Java Program to illustrate how to declare, instantiate, initialize**
2.        **//and traverse the Java array.**
3.        **class** Testarray{

```
4.    public static void main(String args[]){
5.    int a[]=new int[5];//declaration and instantiation
6.    a[0]=10;//initialization
7.    a[1]=20;
8.    a[2]=70;
9.    a[3]=40;
10.   a[4]=50;
11.   //traversing array
12.   for(int i=0;i<a.length;i++)//length is the property of array
13.   System.out.println(a[i]);
14.   }
```

#### Test it Now

Output:

```
10
20
70
40
50
```

## Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
1.    int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Let's see the simple example to print this array.

```
1.    //Java Program to illustrate the use of declaration, instantiation
2.    //and initialization of Java array in a single line
3.    class Testarray1{
4.    public static void main(String args[]){
5.    int a[]={33,3,4,5}; //declaration, instantiation and initialization
6.    //printing array
7.    for(int i=0;i<a.length;i++) //length is the property of array
8.    System.out.println(a[i]);
9.    }}
```

Output:

```
33
3
4
5
```

## For-each Loop for Java Array

We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

```
1.    for(data_type variable:array){
2.    //body of the loop
3.    }
```

Let us see the example of print the elements of Java array using the for-each loop.

```
1. //Java Program to print the array elements using for-each loop
2. class Testarray1{
3. public static void main(String args[]){
4. int arr[]={33,3,4,5};
5. //printing array using for-each loop
6. for(int i:arr)
7. System.out.println(i);
8. }}
```

Output:

```
33
3
4
5
```

## Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get the minimum number of an array using a method.

```
1. //Java Program to demonstrate the way of passing an array
2. //to method.
3. class Testarray2{
4. //creating a method which receives an array as a parameter
5. static void min(int arr[]){
```

```

6.      int min=arr[0];
7.      for(int i=1;i<arr.length;i++)
8.          if(min>arr[i])
9.              min=arr[i];
10.
11.      System.out.println(min);
12.  }
13.
14.      public static void main(String args[]){
15.          int a[]={33,3,4,5}; //declaring and initializing an array
16.          min(a); //passing array to method
17.      }

```

Output:

```
3
```

## Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

```

1.      //Java Program to demonstrate the way of passing an anonymous array
2.      //to method.
3.      public class TestAnonymousArray{
4.          //creating a method which receives an array as a parameter
5.          static void printArray(int arr[]){
6.              for(int i=0;i<arr.length;i++)

```



```
7.      System.out.println(arr[i]);
8.      }
9.
10.     public static void main(String args[]){
11.         printArray(new int[]{10,22,44,66});//passing anonymous array to method
12.     }}
```

Output:

```
10
22
44
66
```

## Returning Array from the Method

We can also return an array from the method in Java.

```
1.      //Java Program to return an array from the method
2.      class TestReturnArray{
3.          //creating method which returns an array
4.          static int[] get(){
5.              return new int[]{10,30,50,90,60};
6.          }
7.
8.          public static void main(String args[]){
9.              //calling method which returns an array
10.         int arr[]=get();
```

```
11.    //printing the values of an array
12.    for(int i=0;i<arr.length;i++)
13.        System.out.println(arr[i]);
14.    }}
```

Output:

```
10
30
50
90
60
```

## Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

### Syntax to Declare Multidimensional Array in Java

```
1.    dataType[][] arrayRefVar; (or)
2.    dataType [][]arrayRefVar; (or)
3.    dataType arrayRefVar[][]; (or)
4.    dataType []arrayRefVar[];
```

### Example to instantiate Multidimensional Array in Java

```
1.    int[][] arr=new int[3][3]; //3 row and 3 column
```

### Example to initialize Multidimensional Array in Java

```
1.    arr[0][0]=1;
2.    arr[0][1]=2;
3.    arr[0][2]=3;
4.    arr[1][0]=4;
5.    arr[1][1]=5;
6.    arr[1][2]=6;
7.    arr[2][0]=7;
8.    arr[2][1]=8;
9.    arr[2][2]=9;
```

## Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
1.    //Java Program to illustrate the use of multidimensional array
2.    class Testarray3{
3.    public static void main(String args[]){
4.    //declaring and initializing 2D array
5.    int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
6.    //printing 2D array
7.    for(int i=0;i<3;i++){
8.    for(int j=0;j<3;j++){
9.        System.out.print(arr[i][j]+" ");
10.    }
11.    System.out.println();
12.    }
13.    }}
```

### Test it Now

Output:

```
1 2 3
2 4 5
4 4 5
```

## Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

```
1.      //Java Program to illustrate the jagged array
2.      class TestJaggedArray{
3.          public static void main(String[] args){
4.              //declaring a 2D array with odd columns
5.              int arr[][] = new int[3][];
6.              arr[0] = new int[3];
7.              arr[1] = new int[4];
8.              arr[2] = new int[2];
9.              //initializing a jagged array
10.             int count = 0;
11.             for (int i=0; i<arr.length; i++)
12.                 for(int j=0; j<arr[i].length; j++)
13.                     arr[i][j] = count++;
14.
```

```
15.         //printing the data of a jagged array
16.         for (int i=0; i<arr.length; i++){
17.             for (int j=0; j<arr[i].length; j++){
18.                 System.out.print(arr[i][j]+ " ");
19.             }
20.             System.out.println();//new line
21.         }
22.     }
23. }
```

Output:

```
0 1 2
3 4 5 6
7 8
```

# 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 software development and maintenance by providing some concepts:

- [Object](#)
- Class
- [Inheritance](#)
- [Polymorphism](#)
- [Abstraction](#)
- [Encapsulation](#)

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

- Coupling
- Cohesion
- Association
- Aggregation
- Composition

## 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 in 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, a 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.

## Coupling

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

## Cohesion

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

## Association

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- One to One
- One to Many
- Many to One, and
- Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be unidirectional or bidirectional.

## Aggregation

Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a *has-a* relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

## Composition

The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

---

## 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, global data can be accessed from anywhere.

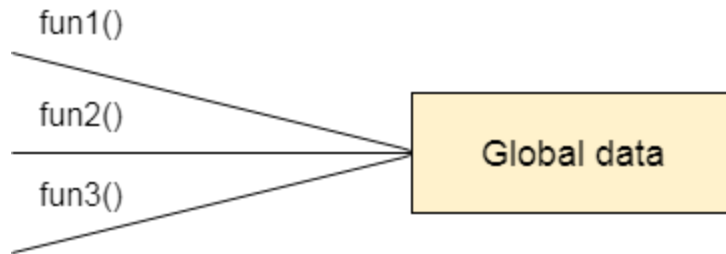


Figure: Data Representation in Procedure-Oriented Programming

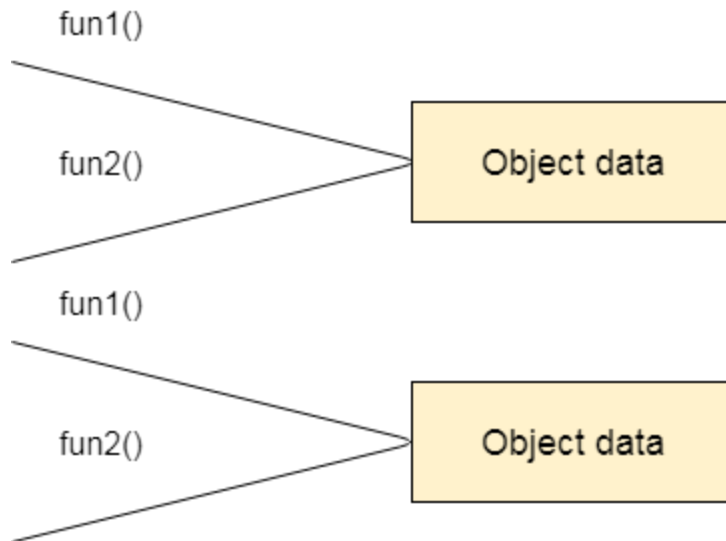


Figure: Data Representation in Object-Oriented Programming

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.

**Object** – Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.

**Class** – A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

## Difference between object and class

There are many differences between object and class. A list of differences between object and class are given below:

No.	Object	Class
1)	Object is an <b>instance</b> of a class.	Class is a <b>blueprint or template</b> from which objects are created.
2)	Object is a <b>real world entity</b> such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a <b>group of similar objects</b> .
3)	Object is a <b>physical</b> entity.	Class is a <b>logical</b> entity.
4)	Object is created through <b>new keyword</b> mainly e.g. Student s1=new Student();	Class is declared using <b>class keyword</b> e.g. class Student{}

5)	Object is created <b>many times</b> as per requirement.	Class is declared <b>once</b> .
6)	Object <b>allocates memory when it is created</b> .	Class <b>doesn't allocated memory when it is created</b> .
7)	There are <b>many ways to create object</b> in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only <b>one way to define class</b> in java using class keyword.

Let's see some real life example of class and object in java to understand the difference well:

**Class:** Human **Object:** Man, Woman

## Objects in Java

Let us now look deep into what are objects. If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

If you compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

# Classes in Java

A class is a blueprint from which individual objects are created.

Following is a sample of a class.

## Example

```
public class Dog {  
    String breed;  
    int age;  
    String color;  
  
    void barking() {  
    }  
  
    void hungry() {  
    }  
  
    void sleeping() {  
    }  
}
```

A class can contain any of the following variable types.

- **Local variables** – Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables** – Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables** – Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods

## Creating an Object

As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.

There are three steps when creating an object from a class –

- **Declaration** – A variable declaration with a variable name with an object type.
- **Instantiation** – The 'new' keyword is used to create the object.
- **Initialization** – The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other start-up procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

## Syntax

Following is the syntax of a constructor –

```
class ClassName {  
    ClassName() {  
    }  
}
```

Java allows two types of constructors namely –

- No argument Constructors
- Parameterized Constructors

## No argument Constructors

As the name specifies the no argument constructors of Java does not accept any parameters instead, using these constructors the instance variables of a method will be initialized with fixed values for all objects.

### Example

```
Public class MyClass {  
    Int num;  
    MyClass() {  
        num = 100;  
    }  
}
```

You would call constructor to initialize objects as follows

```
public class ConsDemo {  
    public static void main(String args[]) {  
        MyClass t1 = new MyClass();  
        MyClass t2 = new MyClass();  
        System.out.println(t1.num + " " + t2.num);  
    }  
}
```

This would produce the following result

100 100

## Parameterized Constructors



Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

## Example

Here is a simple example that uses a constructor –

```
// A simple constructor.
class MyClass {
    int x;

    // Following is the constructor
    MyClass(int i ) {
        x = i;
    }
}
```

You would call constructor to initialize objects as follows –

```
public class ConsDemo {
    public static void main(String args[]) {
        MyClass t1 = new MyClass( 10 );
        MyClass t2 = new MyClass( 20 );
        System.out.println(t1.x + " " + t2.x);
    }
}
```

This would produce the following result –

```
10 20
```

## 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 object.
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
```

## Difference between constructor and method in Java

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

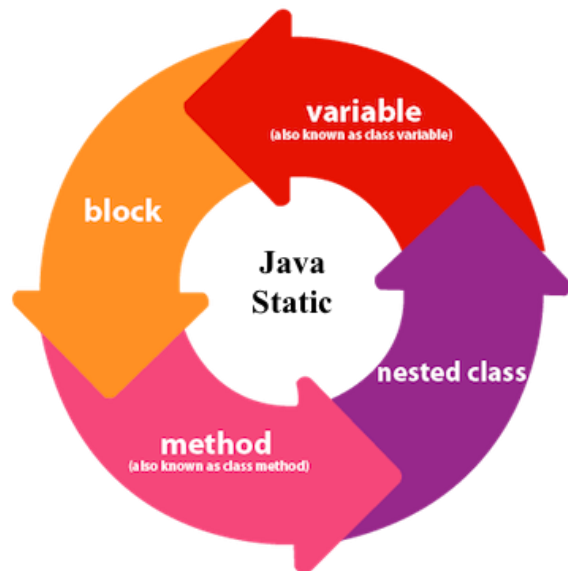
Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java 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 the class name.

# Java static keyword

The **static keyword** in [Java](#) is used for memory management mainly. We can apply static keyword with [variables](#), methods, blocks and [nested classes](#). 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

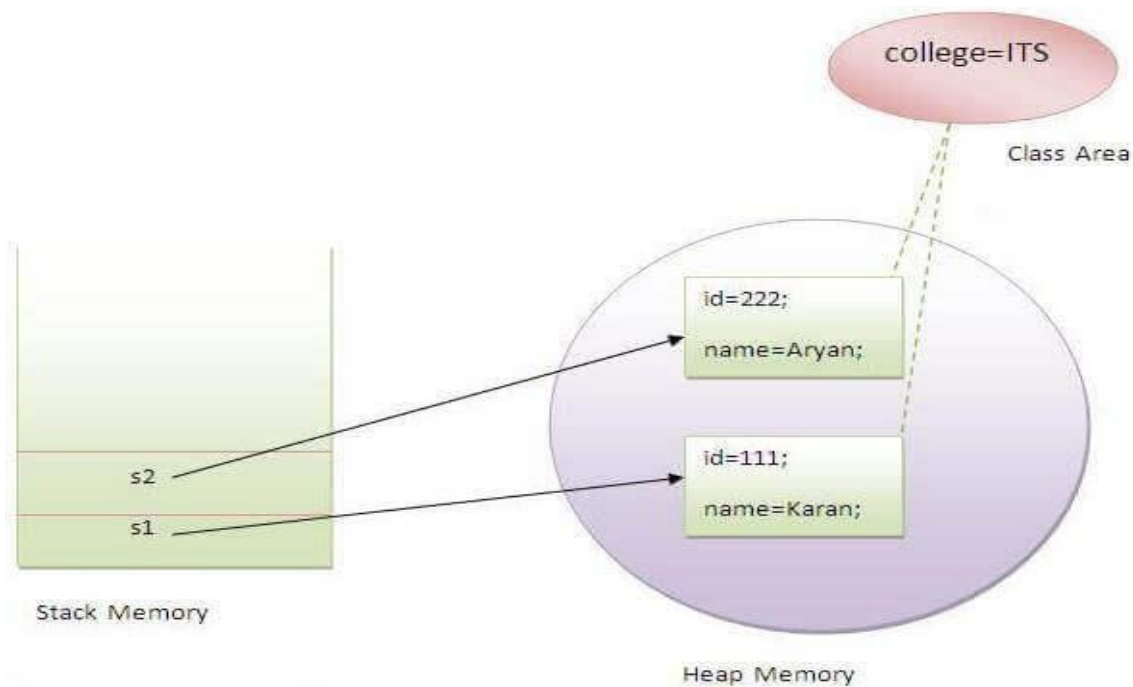
```
1.      //Java Program to demonstrate the use of static variable
2.      class Student{
3.          int rollno;//instance variable
4.          String name;
```

```
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. }
```

### **Test it Now**

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. }
```

### **Test it Now**

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. }
```

### **Test it Now**

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";
9.        }
10.       //constructor to initialize the variable
11.       Student(int r, String n){
12.           rollNo = r;
13.           name = n;
14.       }
15.       //method to display values
16.       void display(){System.out.println(rollNo+" "+name+" "+college);}
17.   }
18.   //Test class to create and display the values of object
19.   public class TestStaticMethod{
20.       public static void main(String args[]){
21.           Student.change();//calling change method
22.           //creating objects
23.           Student s1 = new Student(111,"Karan");
24.           Student s2 = new Student(222,"Aryan");
25.           Student s3 = new Student(333,"Sonoo");
26.           //calling display method
27.           s1.display();
```

```
28.         s2.display();
29.         s3.display();
30.     }
31. }
```

#### Test it Now

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

### 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.

### Q) Why is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, [JVM](#) creates an object first then call main() method that will lead the problem of extra memory allocation.

## 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

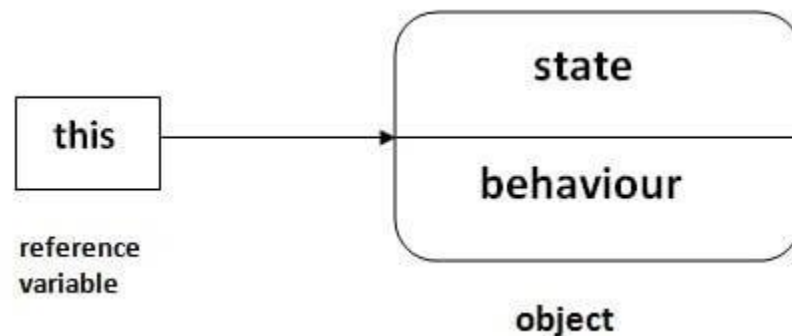
```
1.  class A2{
2.      static{System.out.println("static block is invoked");}
3.      public static void main(String args[]){
4.          System.out.println("Hello main");
5.      }
6.  }
```

### Test it Now

```
Output:static block is invoked
       Hello main
```

# 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.

### 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 parameters, this 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){
6.    rollno=rollno;
7.    name=name;
8.    fee=fee;
9.    }
10.   void display(){System.out.println(rollno+" "+name+" "+fee);}
11.   }
12.   class TestThis1{
13.   public static void main(String args[]){
14.   Student s1=new Student(111,"ankit",5000f);
15.   Student s2=new Student(112,"sumit",6000f);
16.   s1.display();
17.   s2.display();
18.   }}
```

#### **Test it Now**

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 distinguish local variable and instance variable.

*Solution of the above problem by this keyword*

```
1.      class Student{
2.      int rollno;
3.      String name;
4.      float fee;
5.      Student(int rollno,String name,float fee){
6.      this.rollno=rollno;
7.      this.name=name;
8.      this.fee=fee;
9.      }
10.     void display(){System.out.println(rollno+" "+name+" "+fee);}
11.     }
12.
13.     class TestThis2{
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.     }}
```

**Test it Now**

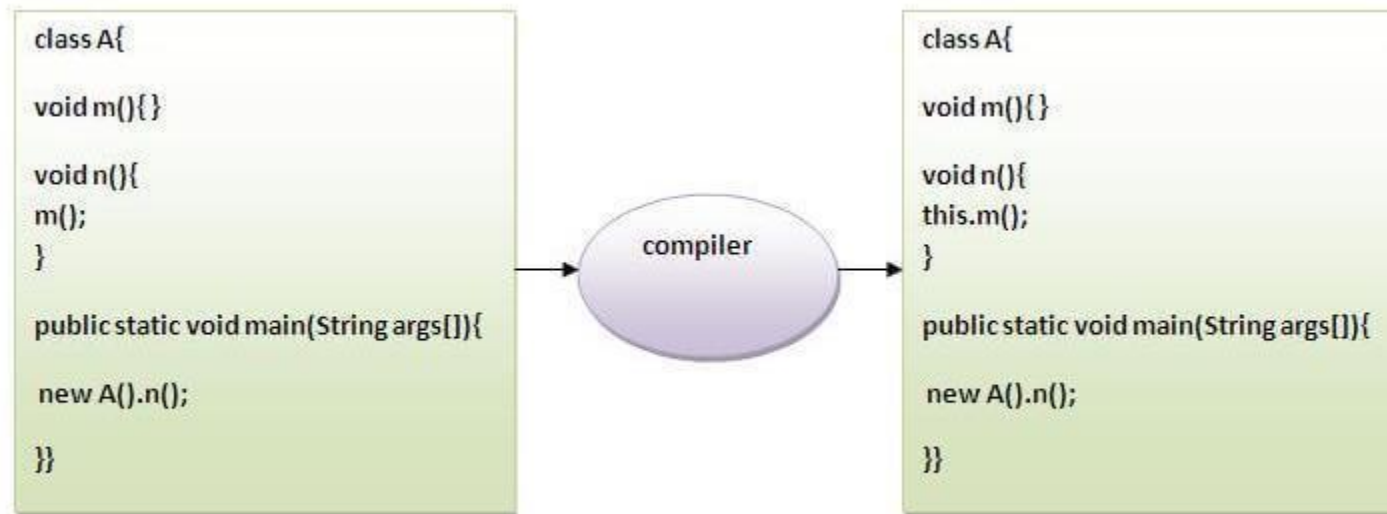
Output:

```
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 following program:

## 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 automatically adds `this` 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.      }}
```

**Test it Now**

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 words, it is used for constructor chaining.

**Calling default constructor from parameterized constructor:**

```
1.      class A{
2.      A()
3.      {
4.          System.out.println("hello a");
5.      }
6.      A(int x){
7.      this();
8.      System.out.println(x);
9.      }
10.     }
11.     class TestThis5{
12.     public static void main(String args[]){
13.     A a=new A(10);
14.     }}
```

12.       }}

Output:

```
hello a
10
```

he this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

**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){
7.      System.out.println(x);
8.      }
9.      }
10.     class TestThis6{
11.     public static void main(String args[]){
12.     A a=new A();
13.     }}
```

**Test it Now**

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 constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

```
1.    class Student{  
2.    int rollno;  
3.    String name,course;  
4.    float fee;  
5.    Student(int rollno,String name,String course){  
6.    this.rollno=rollno;  
7.    this.name=name;  
8.    this.course=course;  
9.    }  
10.   Student(int rollno,String name,String course,float fee){  
11.   this(rollno,name,course);  
12.   this.fee=fee;  
13.   }  
14.   void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
```

```
15.     }
16.     class TestThis7{
17.     public static void main(String args[]){
18.         Student s1=new Student(111,"ankit","java");
19.         Student s2=new Student(112,"sumit","java",6000f);
20.         s1.display();
21.         s2.display();
22.     }}
```

**Test it Now**

Output:

```
111 ankit java 0.0
112 sumit java 6000
```

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

```
1.     class Student{
2.     int rollno;
3.     String name,course;
4.     float fee;
5.     Student(int rollno,String name,String course){
6.         this.rollno=rollno;
7.         this.name=name;
8.         this.course=course;
9.     }
```

```

10. Student(int rollNo,String name,String course,float fee){
11.     this.fee=fee;
12.     this(rollNo,name,course);//C.T.Error
13. }
14. void display(){System.out.println(rollNo+" "+name+" "+course+" "+fee);}
15. }
16. class TestThis8{
17.     public static void main(String args[]){
18.         Student s1=new Student(111,"ankit","java");
19.         Student s2=new Student(112,"sumit","java",6000f);
20.         s1.display();
21.         s2.display();
22.     }}

```

#### Test it Now

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 example:

```

1. class S2{
2.     void m(S2 obj){
3.         System.out.println("method is invoked");

```

```
4.     }
5.     void p(){
6.     m(this);
7.     }
8.     public static void main(String args[]){
9.     S2 s1 = new S2();
10.    s1.p();
11.    }
12. }
```

**Test it Now**

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 in many methods.

## 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 example:

```
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 (non-primitive). Let's 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.      }  
5.      void msg()  
6.      {  
7.      System.out.println("Hello java");  
8.      }  
9.      }  
10.     class Test1{  
11.     public static void main(String args[]){  
12.     new A().getA().msg();  
13.     }  
14.     }
```

**Test it Now**

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 variable and this, output of both variables are same.

```
1.    class A5{
2.    void m(){
3.    System.out.println(this);//prints same reference ID
4.    }
5.    public static void main(String args[]){
6.    A5 obj=new A5();
7.    System.out.println(obj);//prints the reference ID
8.    obj.m();
9.    }
10. }
```

**Test it Now**

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.

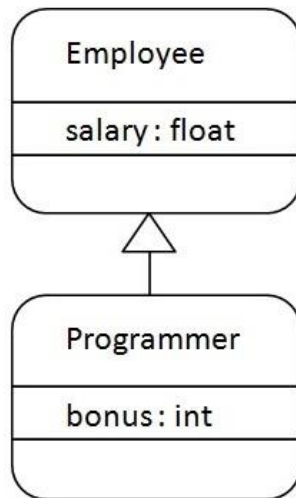
## The syntax of Java Inheritance

1. `class` Subclass-name `extends` Superclass-name
2. `{`
3. `//methods and fields`
4. `}`

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.

```
1.  class Employee{
2.      float salary=40000;
3.  }
4.  class Programmer extends Employee{
5.      int bonus=10000;
6.      public static void main(String args[]){
7.          Programmer p=new Programmer();
8.          System.out.println("Programmer salary is:"+p.salary);
9.          System.out.println("Bonus of Programmer is:"+p.bonus);
10.     }
11. }
```

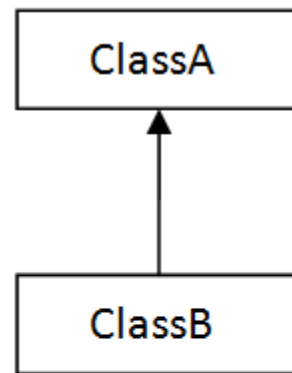
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

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

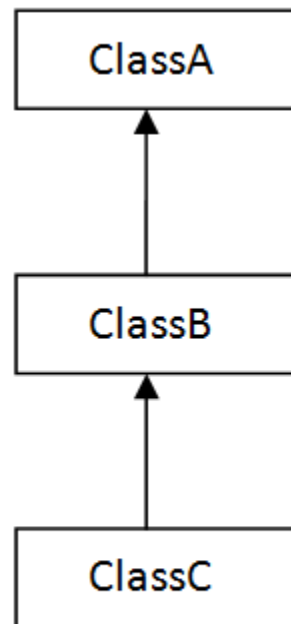
## Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

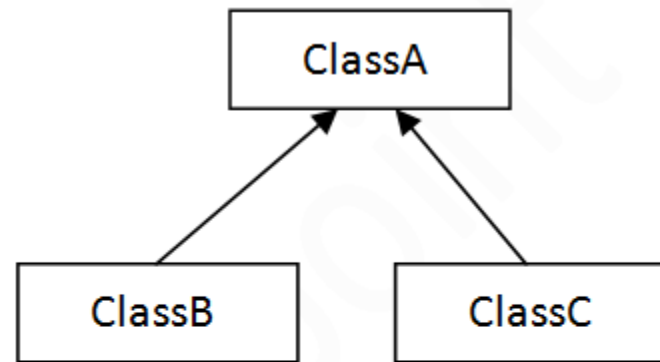
In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



1) Single

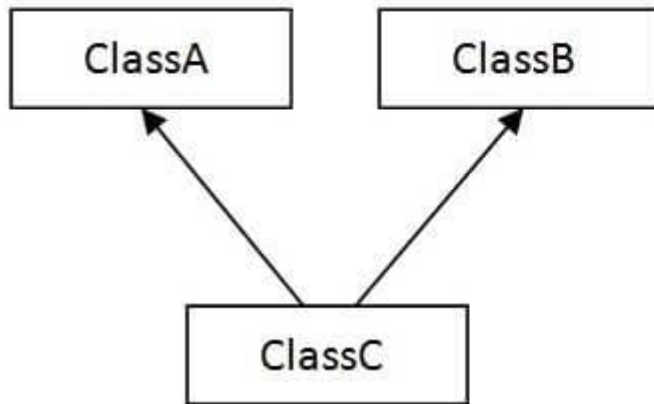


2) Multilevel

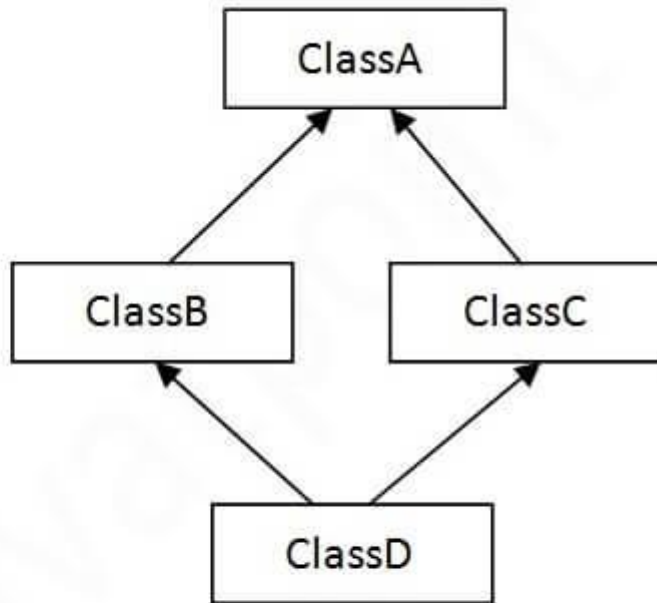


3) Hierarchical

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



4) Multiple



5) Hybrid

## Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

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

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

*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.
11.   class TestInheritance2{
12.   public static void main(String args[]){
13.   BabyDog d=new BabyDog();
14.   d.weep();
15.   d.bark();
16.   d.eat();
17.   }}
```

Output:

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

## Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

*File: TestInheritance3.java*

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



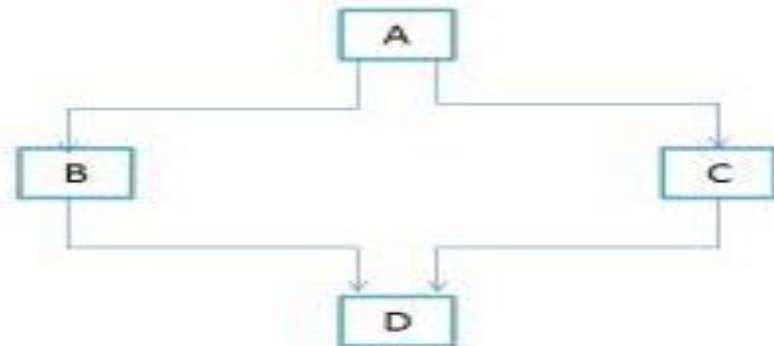
```
5.     void bark(){System.out.println("barking...");}
6.     }
7.     class Cat extends Animal{
8.     void meow(){System.out.println("meowing...");}
9.     }
10.    class TestInheritance3{
11.    public static void main(String args[]){
12.    Cat c=new Cat();
13.    c.meow();
14.    c.eat();
15.    //c.bark();//C.T.Error
16.    }}
```

Output:

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

## Hybrid Inheritance

In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple inheritance**. A typical flow diagram would look like below. A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces. yes you heard it right. By using **interfaces** you can have multiple as well as **hybrid inheritance** in Java.



(e) Hybrid Inheritance

## Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
1.  class A{
2.    void msg(){System.out.println("Hello");}
3.  }
4.  class B{
5.    void msg(){System.out.println("Welcome");}
6.  }
```

```
7.      class C extends A,B{//suppose if it were
8.
9.
10.     public static void main(String args[]){
11.         C obj=new C();
12.         obj.msg();//Now which msg() method would be invoked?
13.     }
14. }
```

Compile Time Error

## Aggregation in Java

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

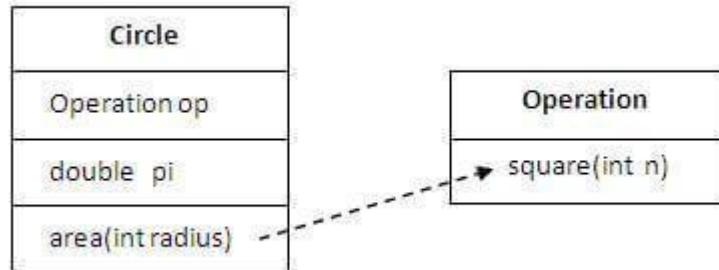
```
1.      class Employee{
2.      int id;
3.      String name;
4.      Address address;//Address is a class
5.      ...
6.      }
```

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

## Why use Aggregation?

- For Code Reusability.

## imple Example of Aggregation



In this example, we have created the reference of Operation class in the Circle class.

```
1.  class Operation{
2.      int square(int n){
3.          return n*n;
4.      }
5.  }
6.
7.  class Circle{
8.      Operation op;//aggregation
9.      double pi=3.14;
10.
11.     double area(int radius){
12.         op=new Operation();
13.         int rsquare=op.square(radius);//code reusability (i.e. delegates the method call).
```

```
14.     return pi*rsquare;
15.     }
16.
17.
18.
19.     public static void main(String args[]){
20.         Circle c=new Circle();
21.         double result=c.area(5);
22.         System.out.println(result);
23.     }
24. }
```

#### Test it Now

Output:78.5

## When use Aggregation?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

In this example, Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

### Address.java

```
1.     public class Address {
2.         String city,state,country;
3.
4.         public Address(String city, String state, String country) {
5.             this.city = city;
```

```
6.         this.state = state;
7.         this.country = country;
8.     }
9.
10. }
```

### *Emp.java*

```
1.     public class Emp {
2.         int id;
3.         String name;
4.         Address address;
5.
6.         public Emp(int id, String name,Address address) {
7.             this.id = id;
8.             this.name = name;
9.             this.address=address;
10.        }
11.
12.        void display(){
13.            System.out.println(id+" "+name);
14.            System.out.println(address.city+" "+address.state+" "+address.country);
15.        }
16.
17.        public static void main(String[] args) {
18.            Address address1=new Address("gzb","UP","india");
19.            Address address2=new Address("gno","UP","india");
20.
```

```
21.      Emp e=new Emp(111,"varun",address1);
22.      Emp e2=new Emp(112,"arun",address2);
23.
24.      e.display();
25.      e2.display();
26.
27.      }
28.      }
```

### Test it Now

```
Output:111 varun
        gzb UP india
        112 arun
        gno UP india
```

# 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.        }}
```

**Test it Now**

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.        }}
```

**Test it Now**

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.     }
```

**Test it Now**

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?*

## 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.     }
```

**Test it Now**

Output:

```
main with String[]
```

# 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.    }

```

**Test it Now**

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. }
```

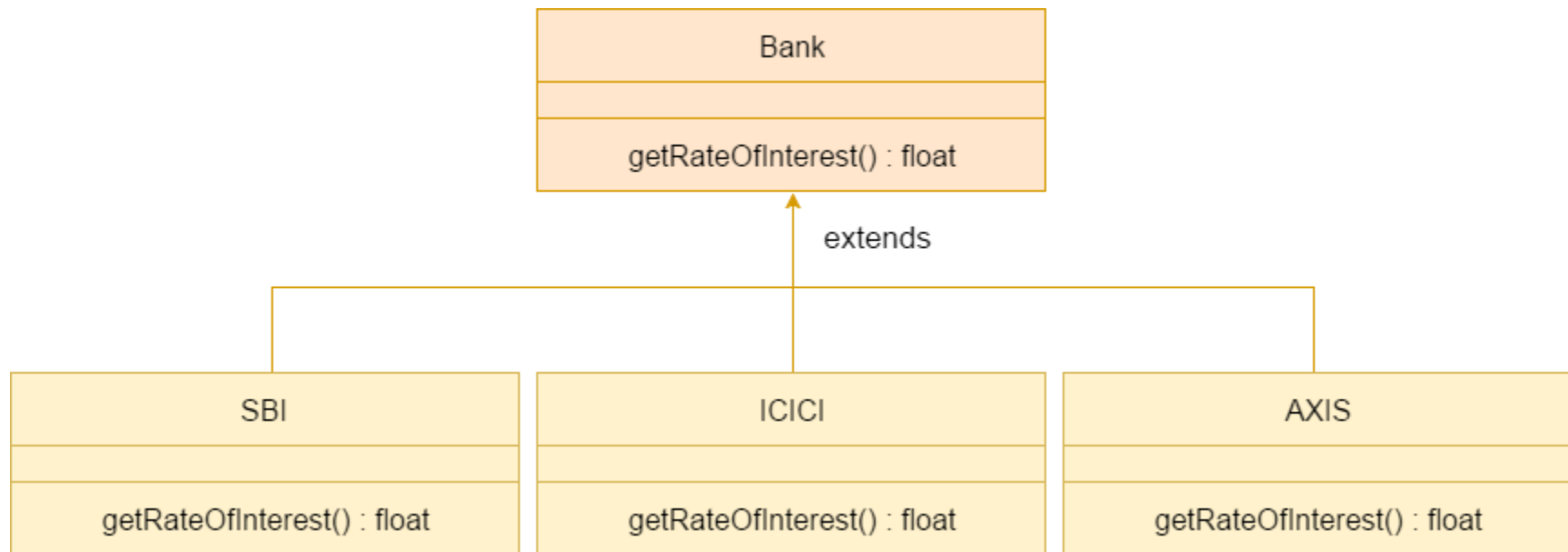
**Test it Now**

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.     }
```

### Test it Now

Output:

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

## Can we override static method?

No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

---

## Why can we not override static method?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.



---

## Can we override java main method?

No, because the main is a static method.

## Difference between method overloading and method overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

No.	Method Overloading	Method Overriding
1)	Method overloading is used <i>to increase the readability</i> of the program.	Method overriding is used <i>to provide the specific implementation</i> of the method that is already provided by its super class.
2)	Method overloading is performed <i>within class</i> .	Method overriding occurs <i>in two classes</i> that have IS-A (inheritance) relationship.
3)	In case of method overloading, <i>parameter must be different</i> .	In case of method overriding, <i>parameter must be same</i> .

4)	Method overloading is the example of <i>compile time polymorphism</i> .	Method overriding is the example of <i>run time polymorphism</i> .
5)	In java, method overloading can't be performed by changing return type of the method only. <i>Return type can be same or different</i> in method overloading. But you must have to change the parameter.	<i>Return type must be same or covariant</i> in method overriding.

## Super Keyword in Java

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

### Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

### 1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
1.      class Animal{
2.      String color="white";
3.      }
4.      class Dog extends Animal{
5.      String color="black";
6.      void printColor(){
7.      System.out.println(color);//prints color of Dog class
8.      System.out.println(super.color);//prints color of Animal class
9.      }
10.     }
11.     class TestSuper1{
12.     public static void main(String args[]){
13.     Dog d=new Dog();
14.     d.printColor();
15.     }}
```

Output:

```
black
white
```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

## 2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
1.      class Animal{
```

```
2.    void eat(){System.out.println("eating...");}
3.    }
4.    class Dog extends Animal{
5.    void eat(){System.out.println("eating bread...");}
6.    void bark(){System.out.println("barking...");}
7.    void work(){
8.    super.eat();
9.    bark();
10.   }
11.  }
12.  class TestSuper2{
13.  public static void main(String args[]){
14.  Dog d=new Dog();
15.  d.work();
16.  }}
```

Output:

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

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

### 3) super is used to invoke parent class constructor.

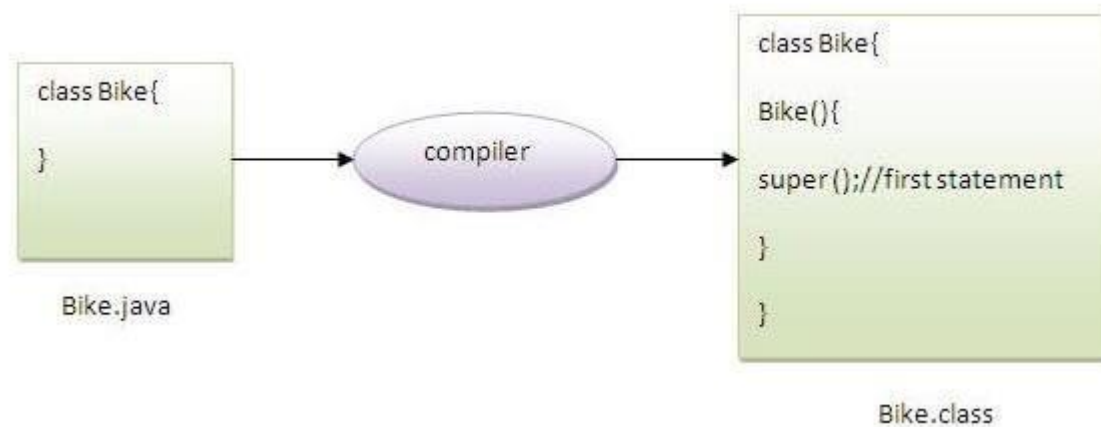
The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
1.    class Animal{
2.    Animal(){System.out.println("animal is created");}
3.    }
4.    class Dog extends Animal{
5.    Dog(){
6.    super();
7.    System.out.println("dog is created");
8.    }
9.    }
10.   class TestSuper3{
11.   public static void main(String args[]){
12.   Dog d=new Dog();
13.   }}
```

Output:

```
animal is created
dog is created
```

*Note: super() is added in each class constructor automatically by compiler if there is no super() or this().*



As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds `super()` as the first statement.

**Another example of super keyword where super() is provided by the compiler implicitly.**

```
1.  class Animal{
2.  Animal(){System.out.println("animal is created");}
3.  }
4.  class Dog extends Animal{
5.  Dog(){
6.  System.out.println("dog is created");
7.  }
8.  }
9.  class TestSuper4{
10. public static void main(String args[]){
11. Dog d=new Dog();
12. }}
```

Output:

```
animal is created  
dog is created
```

## Instance initializer block

1. [Instance initializer block](#)
2. [Example of Instance initializer block](#)
3. [What is invoked firstly instance initializer block or constructor?](#)
4. [Rules for instance initializer block](#)
5. [Program of instance initializer block that is invoked after super\(\)](#)

**Instance Initializer block** is used to initialize the instance data member. It runs each time when an object of the class is created.

The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

*Que) What is the use of instance initializer block while we can directly assign a value in instance data member?*

*For example:*

1. `class Bike{`
2. `int speed=100;`
3. `}`

## Why use instance initializer block?

Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc.

---

## Example of instance initializer block

Let's see the simple example of instance initializer block that performs initialization.

```
1.  class Bike7{
2.      int speed;
3.
4.      Bike7(){System.out.println("speed is "+speed);}
5.
6.      {speed=100;}
7.
8.      public static void main(String args[]){
9.          Bike7 b1=new Bike7();
10.         Bike7 b2=new Bike7();
11.     }
12. }
```

### Test it Now

```
Output:speed is 100
       speed is 100
```

There are three places in java where you can perform operations:

1. method
  2. constructor
  3. block
-



## What is invoked first, instance initializer block or constructor?

```
1.  class Bike8{
2.      int speed;
3.
4.      Bike8(){System.out.println("constructor is invoked");}
5.
6.      {System.out.println("instance initializer block invoked");}
7.
8.      public static void main(String args[]){
9.          Bike8 b1=new Bike8();
10.         Bike8 b2=new Bike8();
11.     }
12. }
```

```
Output:instance initializer block invoked
        constructor is invoked
        instance initializer block invoked
        constructor is invoked
```

In the above example, it seems that instance initializer block is firstly invoked but NO. Instance initializer block is invoked at the time of object creation. The java compiler copies the instance initializer block in the constructor after the first statement `super()`. So firstly, constructor is invoked. Let's understand it by the figure given below:

## Rules for instance initializer block :

There are mainly three rules for the instance initializer block. They are as follows:

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after `super()` constructor call).

3. The instance initializer block comes in the order in which they appear.

## Program of instance initializer block that is invoked after super()

```
1.  class A{
2.  A(){
3.  System.out.println("parent class constructor invoked");
4.  }
5.  }
6.  class B2 extends A{
7.  B2(){
8.  super();
9.  System.out.println("child class constructor invoked");
10. }
11.
12. {System.out.println("instance initializer block is invoked");}
13.
14. public static void main(String args[]){
15. B2 b=new B2();
16. }
17. }
```

```
Output:parent class constructor invoked
        instance initializer block is invoked
        child class constructor invoked
```

# Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

## 1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

### Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

1. `class Bike9{`
2. `final int speedlimit=90; //final variable`
3. `void run(){`

```
4.      speedlimit=400;
5.      }
6.      public static void main(String args[]){
7.      Bike9 obj=new Bike9();
8.      obj.run();
9.      }
10.     }//end of class
```

**Test it Now**

Output:Compile Time Error

## 2) Java final method

If you make any method as final, you cannot override it.

### Example of final method

```
1.      class Bike{
2.      final void run(){System.out.println("running");}
3.      }
4.
5.      class Honda extends Bike{
6.      void run(){System.out.println("running safely with 100kmph");}
7.
8.      public static void main(String args[]){
9.      Honda honda= new Honda();
10.     honda.run();
11.     }
```

12. }

**Test it Now**

Output:Compile Time Error

### 3) Java final class

If you make any class as final, you cannot extend it.

#### Example of final class

```
1.    final class Bike{}
2.
3.    class Honda1 extends Bike{
4.        void run(){System.out.println("running safely with 100kmph");}
5.
6.        public static void main(String args[]){
7.            Honda1 honda= new Honda1();
8.            honda.run();
9.        }
10.    }
```

**Test it Now**

Output:Compile Time Error

## Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
1.    class Bike{
2.        final void run(){System.out.println("running...");}
3.    }
4.    class Honda2 extends Bike{
5.        public static void main(String args[]){
6.            new Honda2().run();
7.        }
8.    }
```

**Test it Now**

Output:running...

## Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

## Example of blank final variable

```
1.    class Student{  
2.    int id;  
3.    String name;  
4.    final String PAN_CARD_NUMBER;  
5.    ...  
6.    }
```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
1.    class Bike10{  
2.    final int speedlimit;//blank final variable  
3.  
4.    Bike10(){  
5.    speedlimit=70;  
6.    System.out.println(speedlimit);  
7.    }  
8.  
9.    public static void main(String args[]){  
10.    new Bike10();  
11.    }  
12. }
```

**Test it Now**

Output: 70

---

## static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

## Example of static blank final variable

```
1.      class A{
2.          static final int data;//static blank final variable
3.          static{ data=50;}
4.          public static void main(String args[]){
5.              System.out.println(A.data);
6.          }
7.      }
```

---

## Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
1.      class Bike11{
2.          int cube(final int n){
3.              n=n+2;//can't be changed as n is final
4.              n*n*n;
5.          }
6.          public static void main(String args[]){
7.              Bike11 b=new Bike11();
8.              b.cube(5);
9.          }
10.     }
```



Output: Compile Time Error

---

## Q) Can we declare a constructor final?

No, because constructor is never inherited.

## Polymorphism in Java

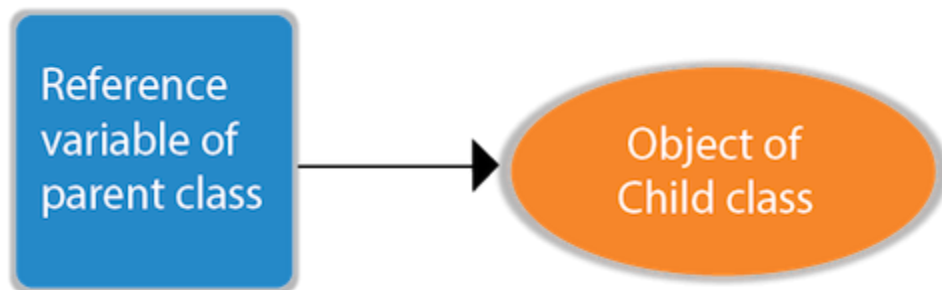
**Polymorphism in Java** is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

### Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



1. `class A{}`
2. `class B extends A{}`
1. `A a=new B();//upcasting`

For upcasting, we can use the reference variable of class type or an interface type. For Example:

1. `interface I{}`
2. `class A{}`
3. `class B extends A implements I{}`

Here, the relationship of B class would be:

```
B IS-A A
B IS-A I
B IS-A Object
```

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

## Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
1.    class Bike{
2.        void run(){System.out.println("running");}
3.    }
4.    class Splendor extends Bike{
5.        void run(){System.out.println("running safely with 60km");}
6.
7.        public static void main(String args[]){
8.            Bike b = new Splendor();//upcasting
9.            b.run();
10.        }
11.    }
```

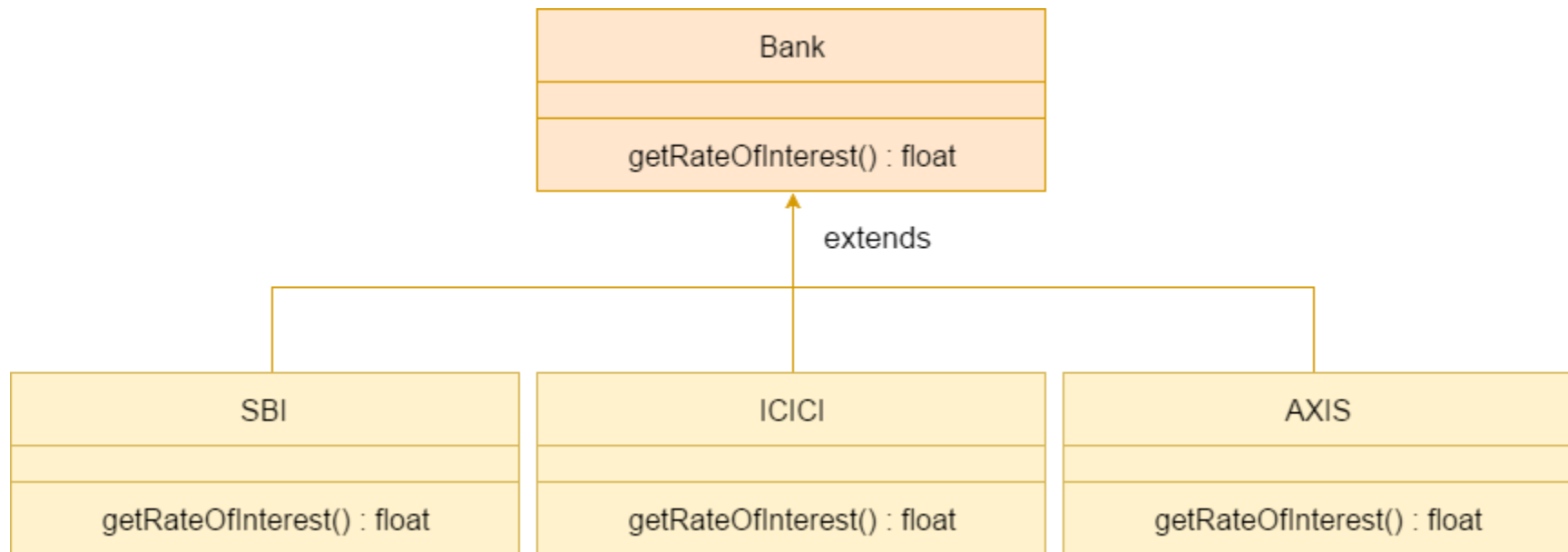
**Test it Now**

Output:

```
running safely with 60km.
```

## Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.



*Note: This example is also given in method overriding but there was no upcasting.*

```
1.  class Bank{
2.  float getRateOfInterest(){return 0;}
3.  }
4.  class SBI extends Bank{
5.  float getRateOfInterest(){return 8.4f;}
6.  }
7.  class ICICI extends Bank{
8.  float getRateOfInterest(){return 7.3f;}
9.  }
10. class AXIS extends Bank{
11. float getRateOfInterest(){return 9.7f;}
12. }
13. class TestPolymorphism{
```

```
14.    public static void main(String args[]){
15.        Bank b;
16.        b=new SBI();
17.        System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
18.        b=new ICICI();
19.        System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
20.        b=new AXIS();
21.        System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
22.    }
23. }
```

Output:

```
SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7
```

## Java Runtime Polymorphism Example: Shape

```
1.    class Shape{
2.        void draw(){System.out.println("drawing...");}
3.    }
4.    class Rectangle extends Shape{
5.        void draw(){System.out.println("drawing rectangle...");}
6.    }
7.    class Circle extends Shape{
8.        void draw(){System.out.println("drawing circle...");}
9.    }
10.   class Triangle extends Shape{
11.       void draw(){System.out.println("drawing triangle...");}
```

```
12.     }
13.     class TestPolymorphism2{
14.     public static void main(String args[]){
15.         Shape s;
16.         s=new Rectangle();
17.         s.draw();
18.         s=new Circle();
19.         s.draw();
20.         s=new Triangle();
21.         s.draw();
22.     }
23. }
```

Output:

```
drawing rectangle...
drawing circle...
drawing triangle...
```

## Java Runtime Polymorphism Example: Animal

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

```
9.     }
10.    class Lion extends Animal{
11.    void eat(){System.out.println("eating meat...");}
12.    }
13.    class TestPolymorphism3{
14.    public static void main(String[] args){
15.    Animal a;
16.    a=new Dog();
17.    a.eat();
18.    a=new Cat();
19.    a.eat();
20.    a=new Lion();
21.    a.eat();
22.    }}
```

Output:

```
eating bread...
eating rat...
eating meat...
```

## Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

```
1.    class Animal{
2.    void eat(){System.out.println("eating");}
3.    }
```

```
4.      class Dog extends Animal{
5.      void eat(){System.out.println("eating fruits");}
6.      }
7.      class BabyDog extends Dog{
8.      void eat(){System.out.println("drinking milk");}
9.      public static void main(String args[]){
10.     Animal a1,a2,a3;
11.     a1=new Animal();
12.     a2=new Dog();
13.     a3=new BabyDog();
14.     a1.eat();
15.     a2.eat();
16.     a3.eat();
17.     }
18.     }
```

Output:

```
eating
eating fruits
drinking Milk
```



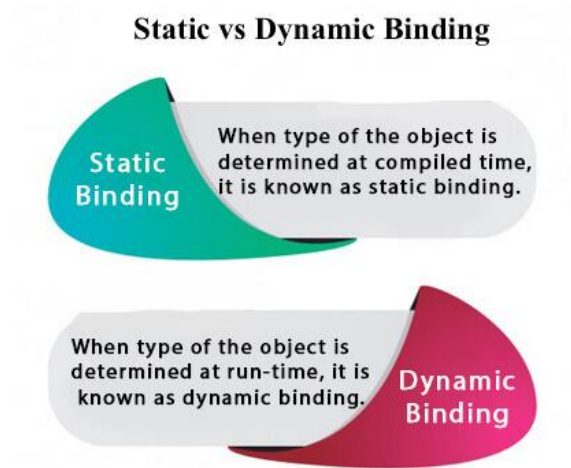
# Static Binding and Dynamic Binding



Connecting a method call to the method body is known as binding.

There are two types of binding

1. Static Binding (also known as Early Binding).
2. Dynamic Binding (also known as Late Binding).



## Static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

## Example of static binding

```
1.      class Dog{  
2.      private void eat(){System.out.println("dog is eating...");}  
3.  
4.      public static void main(String args[]){  
5.      Dog d1=new Dog();  
6.      d1.eat();  
7.      }  
8.      }
```

## Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

## Example of dynamic binding

```
1.      class Animal{  
2.      void eat(){System.out.println("animal is eating...");}  
3.      }  
4.
```

```
5.      class Dog extends Animal{  
6.      void eat(){System.out.println("dog is eating...");}  
7.  
8.      public static void main(String args[]){  
9.          Animal a=new Dog();  
10.         a.eat();  
11.     }  
12. }  
13.
```

Output:dog is eating...

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal. So compiler doesn't know its type, only its base type.

# Abstraction in Java

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the [object](#) does instead of how it does it.

## Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

## Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

### *Points to Remember*

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have [constructors](#) and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

## Rules for Java Abstract class



### Example of abstract class

```
abstract class A{}
```

## Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

### Example of abstract method

1. `abstract void` printStatus();//no method body and abstract
- 

## Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

1. `abstract class` Bike{
2. `abstract void` run();
3. }
4. `class` Honda4 `extends` Bike{
5. `void` run(){System.out.println("running safely");}
6. `public static void` main(String args[]){
7. Bike obj = `new` Honda4();
8. obj.run();
9. }
10. }

### Test it Now

```
running safely
```

# Understanding the real scenario of Abstract class

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

A **factory method** is a method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

*File: TestAbstraction1.java*

```
1.    abstract class Shape{
2.    abstract void draw();
3.    }
4.    //In real scenario, implementation is provided by others i.e. unknown by end user
5.    class Rectangle extends Shape{
6.    void draw(){System.out.println("drawing rectangle");}
7.    }
8.    class Circle1 extends Shape{
9.    void draw(){System.out.println("drawing circle");}
10.   }
11.   //In real scenario, method is called by programmer or user
12.   class TestAbstraction1{
13.   public static void main(String args[]){
14.   Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() method
15.   s.draw();
16.   }
17.   }
```

**Test it Now**

drawing circle

## Another example of Abstract class in java

*File: TestBank.java*

```
1.      abstract class Bank{
2.      abstract int getRateOfInterest();
3.      }
4.      class SBI extends Bank{
5.      int getRateOfInterest(){return 7;}
6.      }
7.      class PNB extends Bank{
8.      int getRateOfInterest(){return 8;}
9.      }
10.
11.     class TestBank{
12.     public static void main(String args[]){
13.     Bank b;
14.     b=new SBI();
15.     System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
16.     b=new PNB();
17.     System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
18.     }}
```

### Test it Now

```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```



# Abstract class having constructor, data member and methods

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

File: TestAbstraction2.java

```
1.      //Example of an abstract class that has abstract and non-abstract methods
2.      abstract class Bike{
3.          Bike(){System.out.println("bike is created");}
4.          abstract void run();
5.          void changeGear(){System.out.println("gear changed");}
6.      }
7.      //Creating a Child class which inherits Abstract class
8.      class Honda extends Bike{
9.          void run(){System.out.println("running safely..");}
10.     }
11.     //Creating a Test class which calls abstract and non-abstract methods
12.     class TestAbstraction2{
13.         public static void main(String args[]){
14.             Bike obj = new Honda();
15.             obj.run();
16.             obj.changeGear();
17.         }
18.     }
```

output

```
bike is created
running safely..
gear changed
```

*Rule: If there is an abstract method in a class, that class must be abstract.*

## Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the [interface](#). In such case, the end user may not be forced to override all the methods of the interface.

Note: If you are beginner to java, learn interface first and skip this example.

```
1.  interface A{
2.      void a();
3.      void b();
4.      void c();
5.      void d();
6.  }
7.
8.  abstract class B implements A{
9.      public void c(){System.out.println("I am c");}
10. }
11.
12. class M extends B{
13.     public void a(){System.out.println("I am a");}
14.     public void b(){System.out.println("I am b");}
15.     public void d(){System.out.println("I am d");}
16. }
17.
18. class Test5{
19.     public static void main(String args[]){
20.         A a=new M();
21.         a.a();
```

```
22.      a.b();
23.      a.c();
24.      a.d();
25.      }}
```

### Test it Now

```
Output:I am a
       I am b
       I am c
       I am d
```

## 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:

1. **interface** <interface\_name>{
- 2.
3.     // declare constant fields
4.     // declare methods that abstract
5.     // by default.
6. }

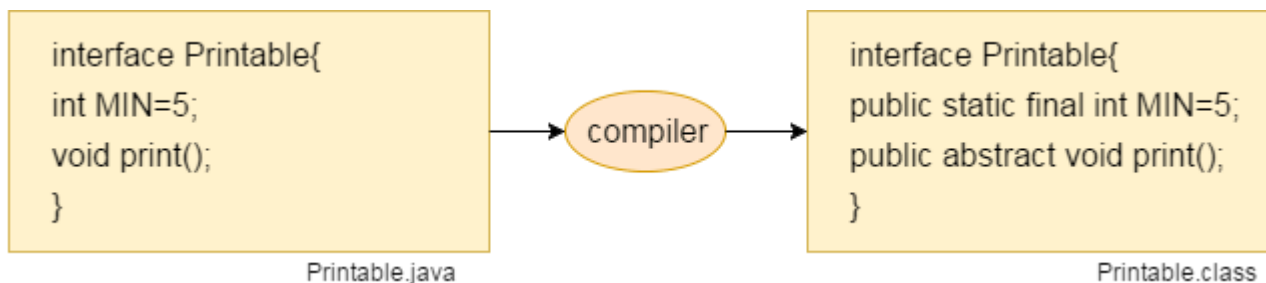
## Java 8 Interface Improvement

Since [Java 8](#), interface can have default and static methods which is discussed later.

## Internal addition by the compiler

*The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.*

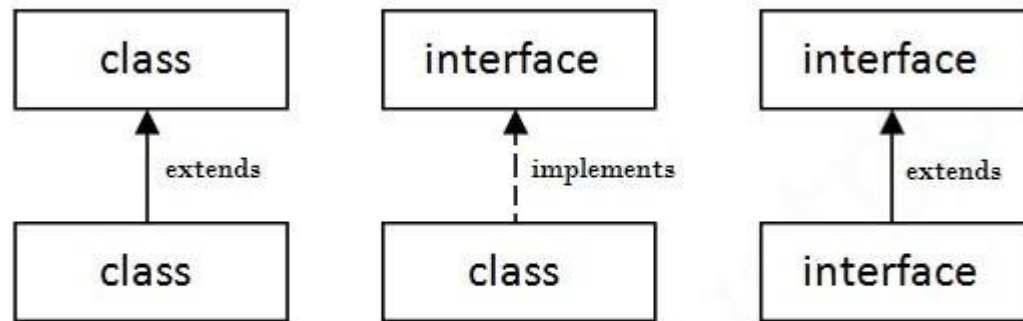
In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



---

## *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.

```
1. interface printable{  
2. void print();  
3. }
```

```
4.      class A6 implements printable{
5.      public void print(){System.out.println("Hello");}
6.
7.      public static void main(String args[]){
8.      A6 obj = new A6();
9.      obj.print();
10.     }
11.    }
```

### Test it Now

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{
```



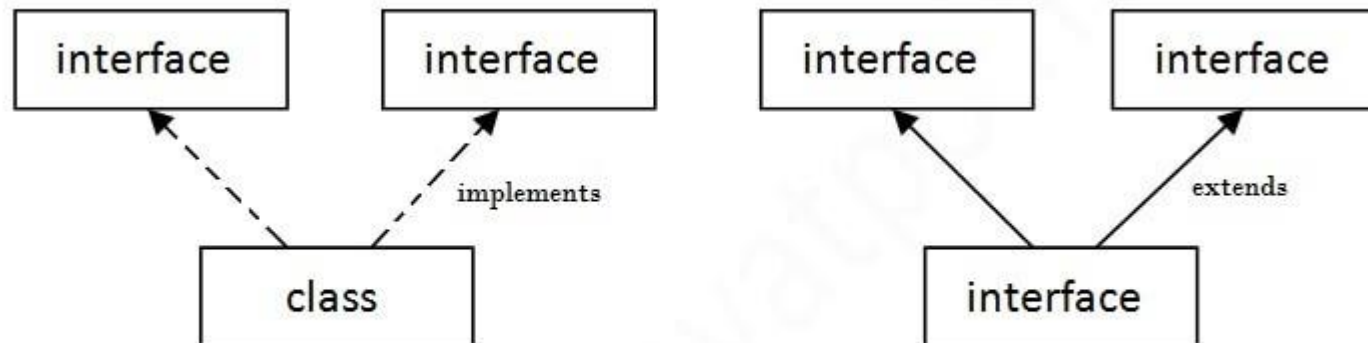
```
11.    public static void main(String[] args){  
12.        Bank b=new SBI();  
13.        System.out.println("ROI: "+b.rateOfInterest());  
14.    }
```

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:

```
1.     interface Printable{
2.     void print();
3.     }
4.     interface Showable{
5.     void print();
```

```
6.     }
7.
8.     class TestInterface3 implements Printable, Showable{
9.     public void print(){System.out.println("Hello");}
10.    public static void main(String args[]){
11.    TestInterface3 obj = new TestInterface3();
12.    obj.print();
13.    }
14.    }
```

Output:

```
Hello
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestInterface1, so there is no ambiguity.

## Interface inheritance

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

```
1.     interface Printable{
2.     void print();
3.     }
4.     interface Showable extends Printable{
5.     void show();
6.     }
7.     class TestInterface4 implements 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.         TestInterface4 obj = new TestInterface4();
13.         obj.print();
14.         obj.show();
15.     }
16. }
```

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
27
```

## Q) What is marker or tagged interface?

An interface which has no member is known as a marker or tagged interface, for example, [Serializable](#), Cloneable, Remote, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
1.     //How Serializable interface is written?
2.     public interface Serializable{
3.     }
```

---

## *Nested Interface in Java*

Note: An interface can have another interface which is known as a nested interface. We will learn it in detail in the [nested classes](#) chapter. For example:

```
1.     interface printable{
2.     void print();
3.     interface MessagePrintable{
4.         void msg();
5.     }
```

6. }

## Example of nested interface which is declared within the interface

In this example, we are going to learn how to declare the nested interface and how we can access it.

```
1.  interface Showable{
2.      void show();
3.      interface Message{
4.          void msg();
5.      }
6.  }
7.  class TestNestedInterface1 implements Showable.Message{
8.      public void msg(){System.out.println("Hello nested interface");}
9.
10.     public static void main(String args[]){
11.         Showable.Message message=new TestNestedInterface1();//upcasting here
12.         message.msg();
13.     }
14. }
```

Output:hello nested interface

As you can see in the above example, we are accessing the Message interface by its outer interface Showable because it cannot be accessed directly. It is just like almirah inside the room, we cannot access the almirah directly because we must enter the room first. In collection framework, sun microsystem has provided a nested interface Entry. Entry is the subinterface of Map i.e. accessed by Map.Entry.

# 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 <b>have abstract and non-abstract</b> methods.	Interface can have <b>only abstract</b> methods. Since Java 8, it can have <b>default and static methods</b> also.
2) Abstract class <b>doesn't support multiple inheritance.</b>	Interface <b>supports multiple inheritance.</b>
3) Abstract class <b>can have final, non-final, static and non-static variables.</b>	Interface has <b>only static and final variables.</b>
4) Abstract class <b>can provide the implementation of interface.</b>	Interface <b>can't provide the implementation of abstract class.</b>
5) The <b>abstract keyword</b> is used to declare abstract class.	The <b>interface keyword</b> is used to declare interface.



6) An <b>abstract class</b> can extend another Java class and implement multiple Java interfaces.	An <b>interface</b> can extend another Java interface only.
7) An <b>abstract class</b> can be extended using keyword "extends".	An <b>interface</b> can be implemented using keyword "implements".
8) A Java <b>abstract class</b> can have class members like private, protected, etc.	Members of a Java interface are public by default.
9) <b>Example:</b> <pre>public abstract class Shape{ public abstract void draw(); }</pre>	<b>Example:</b> <pre>public interface Drawable{ void draw(); }</pre>

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.

```
//Creating interface that has 4 methods
interface A{
void a();//bydefault, public and abstract
void b();
void c();
```

```
void d();  
}
```

//Creating abstract class that provides the implementation of one method of A interface

```
abstract class B implements A{  
public void c(){System.out.println("I am C");}  
}
```

//Creating subclass of abstract class, now we need to provide the implementation of rest of the methods

```
class M extends B{  
public void a(){System.out.println("I am a");}  
public void b(){System.out.println("I am b");}  
public void d(){System.out.println("I am d");}  
}
```

//Creating a test class that calls the methods of A interface

```
class Test5{  
public static void main(String args[]){  
A a=new M();  
a.a();  
a.b();  
a.c();  
a.d();  
}}}
```

**Test it Now**

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.

# Access Modifiers in Java

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

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

## Understanding Java Access Modifiers

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

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

### 1) Private

The private access modifier is accessible only within the 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 method. We are accessing these private members from outside the class, so there is a 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 example:

```
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

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

### 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 package, since A class is not 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 package.

### 3) Protected

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 class.

It provides more accessibility than the default modifier.

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

```
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

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

# 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.



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.     }
```

Output:

```
vijay
```

## 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*

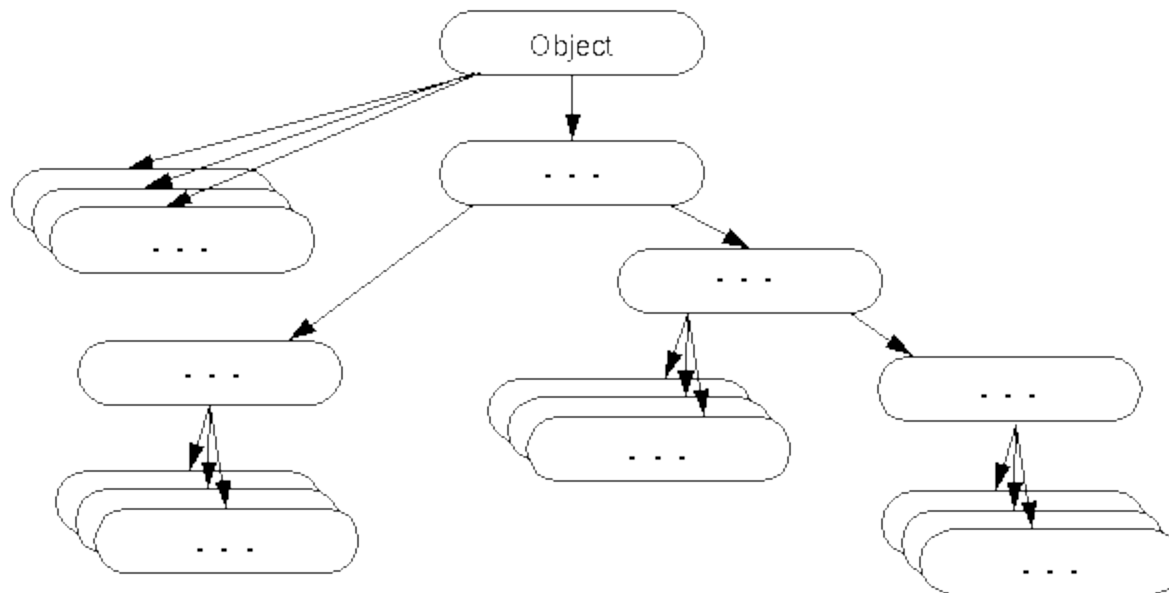
```
1.     //A Java class to test the encapsulated class Account.
2.     public class TestEncapsulation {
3.     public static void main(String[] args) {
```

```
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.     }
```

### Test it Now

Output:

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



## Methods of Object class

The Object class provides many methods. They are as follows:

Method	Description
<code>public final Class getClass()</code>	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
<code>public int hashCode()</code>	returns the hashcode number for this object.
<code>public boolean equals(Object obj)</code>	compares the given object to this object.
<code>protected Object clone() throws CloneNotSupportedException</code>	creates and returns the exact copy (clone) of this object.
<code>public String toString()</code>	returns the string representation of this object.
<code>public final void notify()</code>	wakes up single thread, waiting on this object's monitor.
<code>public final void notifyAll()</code>	wakes up all the threads, waiting on this object's monitor.
<code>public final void wait(long timeout) throws</code>	causes the current thread to wait for the specified milliseconds, until



InterruptedException	another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

# Wrapper classes in Java

The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.

Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

## Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
----------------	---------------

boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
loat	fFloat
double	Double

## Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the `valueOf()` method of wrapper classes to convert the primitive into objects.

### Wrapper class Example: Primitive to Wrapper

```
1.      //Java program to convert primitive into objects
2.      //Autoboxing example of int to Integer
3.      public class WrapperExample1{
4.      public static void main(String args[]){
5.      //Converting int into Integer
6.      int a=20;
7.      Integer i=Integer.valueOf(a);//converting int into Integer explicitly
8.      Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
9.
10.     System.out.println(a+" "+i+" "+j);
11.     }}
```

Output:

```
20 20 20
```

## Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the `intValue()` method of wrapper classes to convert the wrapper type into primitives.

### Wrapper class Example: Wrapper to Primitive

```
1.      //Java program to convert object into primitives
2.      //Unboxing example of Integer to int
3.      public class WrapperExample2{
```

```
4.    public static void main(String args[]){
5.        //Converting Integer to int
6.        Integer a=new Integer(3);
7.        int i=a.intValue();//converting Integer to int explicitly
8.        int j=a;//unboxing, now compiler will write a.intValue() internally
9.
10.       System.out.println(a+ " "+i+ " "+j);
11.    }
```

Output:

```
3 3 3
```

## Java Wrapper classes Example

```
1.    //Java Program to convert all primitives into its corresponding
2.    //wrapper objects and vice-versa
3.    public class WrapperExample3{
4.        public static void main(String args[]){
5.            byte b=10;
6.            short s=20;
7.            int i=30;
8.            long l=40;
9.            float f=50.0F;
10.           double d=60.0D;
11.           char c='a';
```

```
12.    boolean b2=true;
13.
14.    //Autoboxing: Converting primitives into objects
15.    Byte byteobj=b;
16.    Short shortobj=s;
17.    Integer intobj=i;
18.    Long longobj=l;
19.    Float floatobj=f;
20.    Double doubleobj=d;
21.    Character charobj=c;
22.    Boolean boolobj=b2;
23.
24.    //Printing objects
25.    System.out.println("---Printing object values---");
26.    System.out.println("Byte object: "+byteobj);
27.    System.out.println("Short object: "+shortobj);
28.    System.out.println("Integer object: "+intobj);
29.    System.out.println("Long object: "+longobj);
30.    System.out.println("Float object: "+floatobj);
31.    System.out.println("Double object: "+doubleobj);
32.    System.out.println("Character object: "+charobj);
33.    System.out.println("Boolean object: "+boolobj);
34.
35.    //Unboxing: Converting Objects to Primitives
36.    byte bytevalue=byteobj;
37.    short shortvalue=shortobj;
38.    int intvalue=intobj;
39.    long longvalue=longobj;
40.    float floatvalue=floatobj;
```

```
41.    double doublevalue=doubleobj;
42.    char charvalue=charobj;
43.    boolean boolvalue=boolobj;
44.    //Printing primitives
45.    System.out.println("---Printing primitive values---");
46.    System.out.println("byte value: "+bytevalue);
47.    System.out.println("short value: "+shortvalue);
48.    System.out.println("int value: "+intvalue);
49.    System.out.println("long value: "+longvalue);
50.    System.out.println("float value: "+floatvalue);
51.    System.out.println("double value: "+doublevalue);
52.    System.out.println("char value: "+charvalue);
53.    System.out.println("boolean value: "+boolvalue);
54.    }}
```

Output:

```
---Printing object values---
Byte object: 10
Short object: 20
Integer object: 30
Long object: 40
Float object: 50.0
Double object: 60.0
Character object: a
Boolean object: true
---Printing primitive values---
byte value: 10
short value: 20
int value: 30
long value: 40
float value: 50.0
double value: 60.0
char value: a
boolean value: true
```

# Java String

In [Java](#), string is basically an object that represents sequence of char values. An [array](#) of characters works same as Java string. For example:

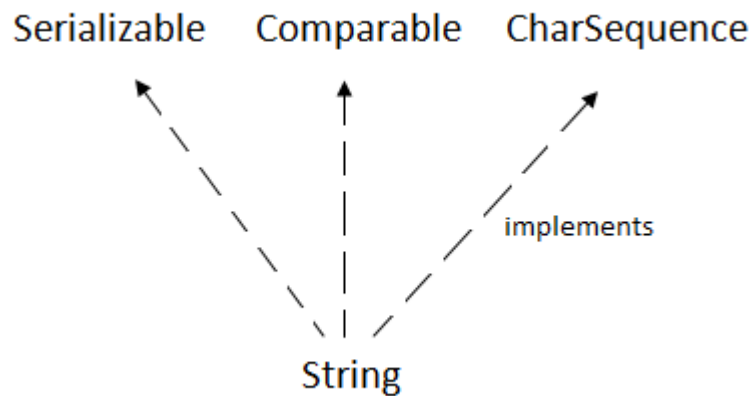
1. `char[] ch={'j','a','v','a','t','p','o','i','n','t'};`
2. `String s=new String(ch);`

is same as:

1. `String s="javatpoint";`

**Java String** class provides a lot of methods to perform operations on strings such as `compare()`, `concat()`, `equals()`, `split()`, `length()`, `replace()`, `compareTo()`, `intern()`, `substring()` etc.

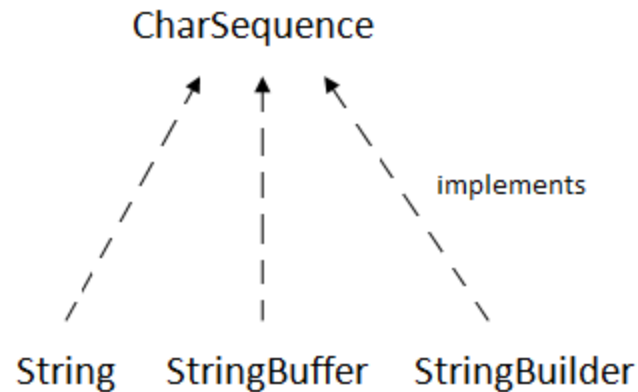
The `java.lang.String` class implements *Serializable*, *Comparable* and *CharSequence* [interfaces](#).





## CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, `StringBuffer` and `StringBuilder` classes implement it. It means, we can create strings in java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use `StringBuffer` and `StringBuilder` classes.

We will discuss immutable string later. Let's first understand what is String in Java and how to create the String object.

## What is String in java

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The `java.lang.String` class is used to create a string object.

## How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

---

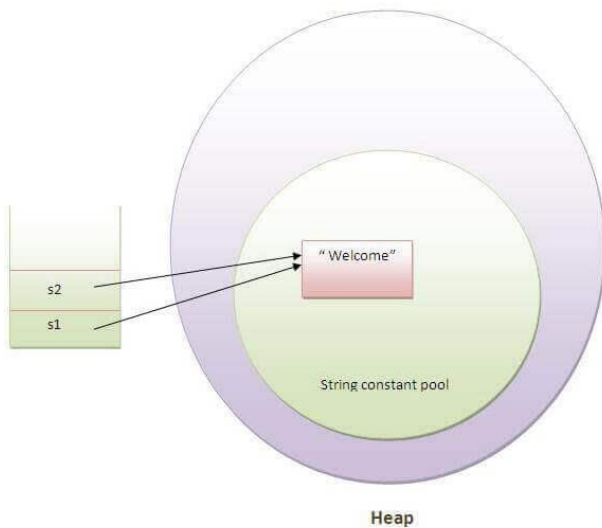
### 1) String Literal

Java String literal is created by using double quotes. For Example:

1. `String s="welcome";`

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. `String s1="Welcome";`
2. `String s2="Welcome";//It doesn't create a new instance`



In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool, that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

*Note: String objects are stored in a special memory area known as the "string constant pool".*

## Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

## 2) By new keyword

1. `String s=new String("Welcome");//creates two objects and one reference variable`

In such case, **JVM** will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

---

## Java String Example

```
1.    public class StringExample{
2.    public static void main(String args[]){
3.    String s1="java";//creating string by java string literal
4.    char ch[]={'s','t','r','i','n','g','s'};
5.    String s2=new String(ch);//converting char array to string
6.    String s3=new String("example");//creating java string by new keyword
7.    System.out.println(s1);
8.    System.out.println(s2);
9.    System.out.println(s3);
10.   }}
```

```
java
strings
example
```

## Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
-----	--------	-------------

1	<u><a href="#">char charAt(int index)</a></u>	returns char value for the particular index
2	<u><a href="#">int length()</a></u>	returns string length
3	<u><a href="#">static String format(String format, Object... args)</a></u>	returns a formatted string.
4	<u><a href="#">static String format(Locale l, String format, Object... args)</a></u>	returns formatted string with given locale.
5	<u><a href="#">String substring(int beginIndex)</a></u>	returns substring for given begin index.
6	<u><a href="#">String substring(int beginIndex, int endIndex)</a></u>	returns substring for given begin index and end index.
7	<u><a href="#">boolean contains(CharSequence s)</a></u>	returns true or false after matching the sequence of char value.
8	<u><a href="#">static String join(CharSequence delimiter, CharSequence... elements)</a></u>	returns a joined string.
9	<u><a href="#">static String join(CharSequence delimiter, Iterable&lt;? extends CharSequence&gt; elements)</a></u>	returns a joined string.

10	<a href="#"><u>boolean equals(Object another)</u></a>	checks the equality of string with the given object.
11	<a href="#"><u>boolean isEmpty()</u></a>	checks if string is empty.
12	<a href="#"><u>String concat(String str)</u></a>	concatenates the specified string.
13	<a href="#"><u>String replace(char old, char new)</u></a>	replaces all occurrences of the specified char value.
14	<a href="#"><u>String replace(CharSequence old, CharSequence new)</u></a>	replaces all occurrences of the specified CharSequence.
15	<a href="#"><u>static String equalsIgnoreCase(String another)</u></a>	compares another string. It doesn't check case.
16	<a href="#"><u>String[] split(String regex)</u></a>	returns a split string matching regex.
17	<a href="#"><u>String[] split(String regex, int limit)</u></a>	returns a split string matching regex and limit.
18	<a href="#"><u>String intern()</u></a>	returns an interned string.

19	<a href="#"><u>int indexOf(int ch)</u></a>	returns the specified char value index.
20	<a href="#"><u>int indexOf(int ch, int fromIndex)</u></a>	returns the specified char value index starting with given index.
21	<a href="#"><u>int indexOf(String substring)</u></a>	returns the specified substring index.
22	<a href="#"><u>int indexOf(String substring, int fromIndex)</u></a>	returns the specified substring index starting with given index.
23	<a href="#"><u>String toLowerCase()</u></a>	returns a string in lowercase.
24	<a href="#"><u>String toLowerCase(Locale l)</u></a>	returns a string in lowercase using specified locale.
25	<a href="#"><u>String toUpperCase()</u></a>	returns a string in uppercase.
26	<a href="#"><u>String toUpperCase(Locale l)</u></a>	returns a string in uppercase using specified locale.
27	<a href="#"><u>String trim()</u></a>	removes beginning and ending spaces of this string.

28

`static String valueOf(int value)`

converts given type into string. It is an overloaded method.

## Immutable String in Java

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once string object is created its data or state can't be changed but a new string object is created.

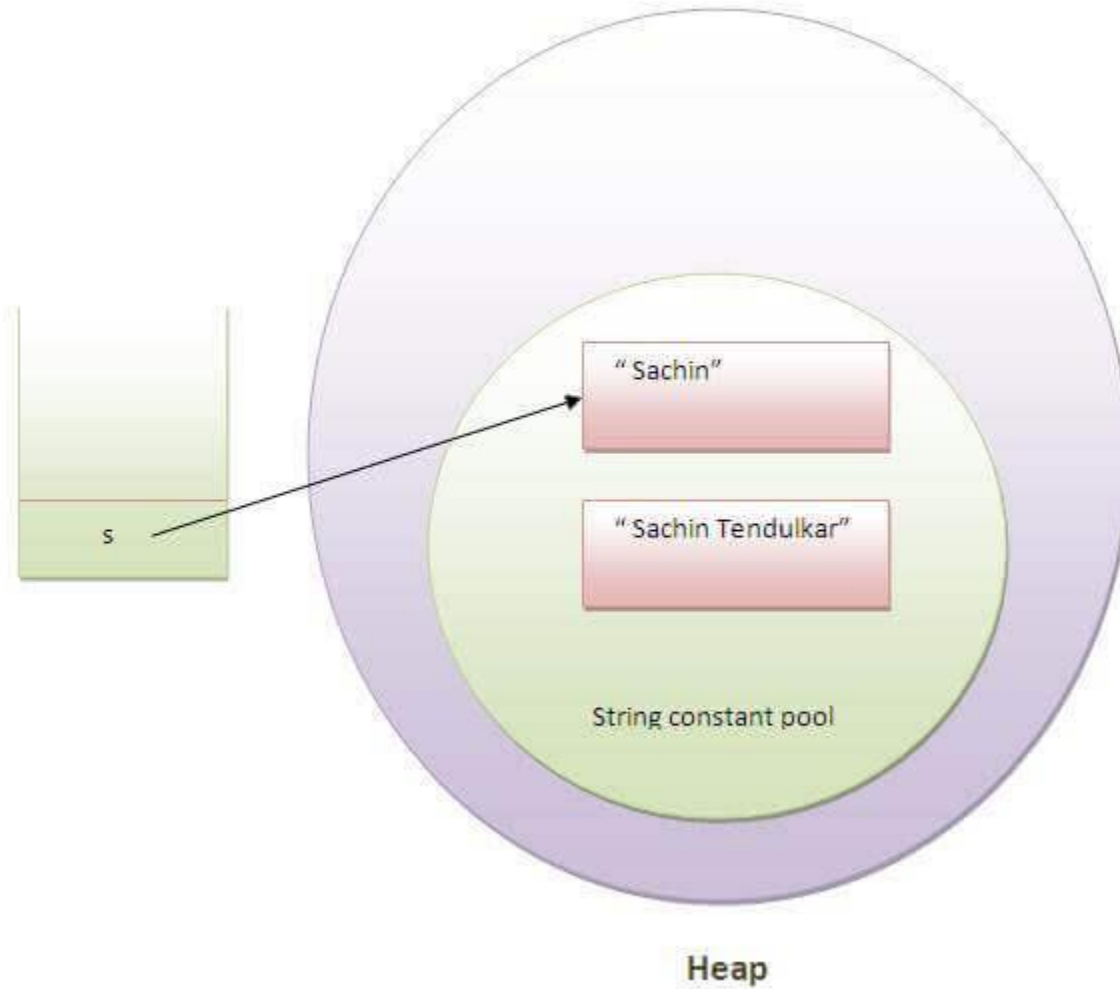
Let's try to understand the immutability concept by the example given below:

```
1.  class Testimmutablestring{
2.      public static void main(String args[]){
3.          String s="Sachin";
4.          s.concat(" Tendulkar");//concat() method appends the string at the end
5.          System.out.println(s);//will print Sachin because strings are immutable objects
6.      }
7.  }
```



Output:Sachin

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with sachintendulkar. That is why string is known as immutable.



As you can see in the above figure that two objects are created but s reference variable still refers to "Sachin" not to "Sachin Tendulkar".

But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object. For example:

```
1.      class TestImmutableString1{
2.      public static void main(String args[]){
3.          String s="Sachin";
4.          s=s.concat(" Tendulkar");
5.          System.out.println(s);
6.      }
7.  }
```

#### Test it Now

Output:Sachin Tendulkar

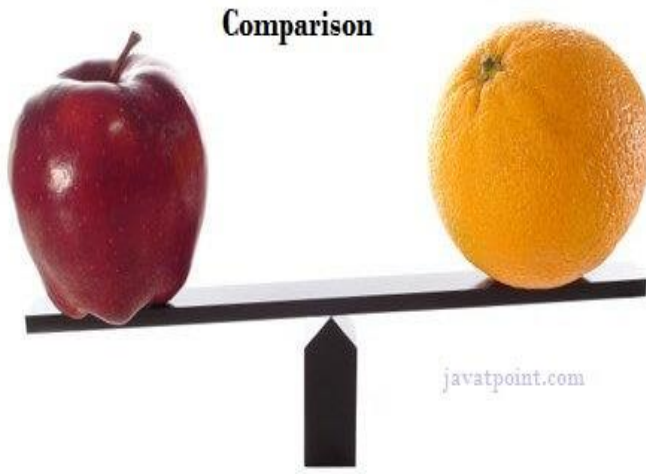
In such case, s points to the "Sachin Tendulkar". Please notice that still sachin object is not modified.

---

## Why string objects are immutable in java?

Because java uses the concept of string literal. Suppose there are 5 reference variables, all refer to one object "sachin". If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.

# Java String compare



We can compare string in java on the basis of content and reference.

It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

There are three ways to compare string in java:

1. By equals() method
2. By == operator
3. By compareTo() method

## 1) String compare by equals() method

The String equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:

- **public boolean equals(Object another)** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another)** compares this String to another string, ignoring case.

```
1.  class Teststringcomparison1{
2.      public static void main(String args[]){
3.          String s1="Sachin";
4.          String s2="Sachin";
5.          String s3=new String("Sachin");
6.          String s4="Saurav";
7.          System.out.println(s1.equals(s2));//true
8.          System.out.println(s1.equals(s3));//true
9.          System.out.println(s1.equals(s4));//false
10.     }
11. }
```

### Test it Now

```
Output:true
      true
      false
```

```
1.  class Teststringcomparison2{
2.      public static void main(String args[]){
3.          String s1="Sachin";
4.          String s2="SACHIN";
5.
6.          System.out.println(s1.equals(s2));//false
```

```
7.      System.out.println(s1.equalsIgnoreCase(s2));//true
8.      }
9.      }
```

### Test it Now

Output:

```
false
true
```

## 2) String compare by == operator

The == operator compares references not values.

```
1.      class Teststringcomparison3{
2.      public static void main(String args[]){
3.          String s1="Sachin";
4.          String s2="Sachin";
5.          String s3=new String("Sachin");
6.          System.out.println(s1==s2);//true (because both refer to same instance)
7.          System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
8.      }
9.      }
```

```
Output:true
        false
```

## 3) String compare by compareTo() method

The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables. If:

- **s1 == s2** :0
- **s1 > s2** :positive value
- **s1 < s2** :negative value

```
1.    class Teststringcomparison4{
2.    public static void main(String args[]){
3.        String s1="Sachin";
4.        String s2="Sachin";
5.        String s3="Ratan";
6.        System.out.println(s1.compareTo(s2));//0
7.        System.out.println(s1.compareTo(s3));//1(because s1>s3)
8.        System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
9.    }
10. }
```

```
Output:0
        1
        -1
```

## String Concatenation in Java

In java, string concatenation forms a new string *that is* the combination of multiple strings. There are two ways to concat string in java:

1. By + (string concatenation) operator
2. By concat() method

## 1) String Concatenation by + (string concatenation) operator

Java string concatenation operator (+) is used to add strings. For Example:

```
1.      class TestStringConcatenation1{
2.      public static void main(String args[]){
3.          String s="Sachin"+" Tendulkar";
4.          System.out.println(s);//Sachin Tendulkar
5.      }
6.  }
```

**Test it Now**

Output:Sachin Tendulkar

The **Java compiler transforms** above code to this:

```
String s=(new StringBuilder()).append("Sachin").append(" Tendulkar").toString();
```

In java, String concatenation is implemented through the StringBuilder (or StringBuffer) class and its append method. String concatenation operator produces a new string by appending the second operand onto the end of the first operand. The string concatenation operator can concat not only string but primitive values also. For Example:

```
1.      class TestStringConcatenation2{
2.      public static void main(String args[]){
3.          String s=50+30+"Sachin"+40+40;
4.          System.out.println(s);//80Sachin4040
```

```
5.     }  
6.     }
```

**Test it Now**

```
80Sachin4040
```

*Note: After a string literal, all the + will be treated as string concatenation operator.*

## 2) String Concatenation by concat() method

The String concat() method concatenates the specified string to the end of current string. Syntax:

```
1.     public String concat(String another)
```

Let's see the example of String concat() method.

```
1.     class TestStringConcatenation3{  
2.         public static void main(String args[]){  
3.             String s1="Sachin ";  
4.             String s2="Tendulkar";  
5.             String s3=s1.concat(s2);  
6.             System.out.println(s3);  
7.         }  
8.     }
```

**Test it Now**

```
Sachin Tendulkar
```



# Substring in Java

A part of string is called **substring**. In other words, substring is a subset of another string. In case of substring `startIndex` is inclusive and `endIndex` is exclusive.

*Note: Index starts from 0.*

You can get substring from the given string object by one of the two methods:

1. **public String substring(int startIndex):** This method returns new String object containing the substring of the given string from specified `startIndex` (inclusive).
2. **public String substring(int startIndex, int endIndex):** This method returns new String object containing the substring of the given string from specified `startIndex` to `endIndex`.

In case of string:

- **startIndex:** inclusive
- **endIndex:** exclusive

Let's understand the `startIndex` and `endIndex` by the code given below.

1. `String s="hello";`
2. `System.out.println(s.substring(0,2));` //he

In the above substring, 0 points to h but 2 points to e (because end index is exclusive).

## Example of java substring

1. `public class TestSubstring{`
2. `public static void main(String args[]){`

```
3.      String s="SachinTendulkar";
4.      System.out.println(s.substring(6));//Tendulkar
5.      System.out.println(s.substring(0,6));//Sachin
6.      }
7.      }
```

```
Tendulkar
Sachin
```

## Java String class methods

The java.lang.String class provides a lot of methods to work on string. By the help of these methods, we can perform operations on string such as trimming, concatenating, converting, comparing, replacing strings etc.

Java String is a powerful concept because everything is treated as a string if you submit any form in window based, web based or mobile application.

Let's see the important methods of String class.

### Java String toUpperCase() and toLowerCase() method

The java string toUpperCase() method converts this string into uppercase letter and string toLowerCase() method into lowercase letter.

```
1.      String s="Sachin";
2.      System.out.println(s.toUpperCase());//SACHIN
3.      System.out.println(s.toLowerCase());//sachin
4.      System.out.println(s);//Sachin(no change in original)
```

**Test it Now**

```
SACHIN
sachin
Sachin
```

## Java String trim() method

The string trim() method eliminates white spaces before and after string.

1. String s=" Sachin ";
2. System.out.println(s);// Sachin
3. System.out.println(s.trim());//Sachin

**Test it Now**

```
Sachin
Sachin
```

## Java String startsWith() and endsWith() method

1. String s="Sachin";
2. System.out.println(s.startsWith("Sa"));//true
3. System.out.println(s.endsWith("n"));//true

**Test it Now**

```
true
true
```

## Java String charAt() method

The string charAt() method returns a character at specified index.

1. String s="Sachin";
2. System.out.println(s.charAt(0));//S
3. System.out.println(s.charAt(3));//h

**Test it Now**

```
S
h
```

## Java String length() method

The string length() method returns length of the string.

1. String s="Sachin";
2. System.out.println(s.length());//6

**Test it Now**

6

## Java String intern() method

A pool of strings, initially empty, is maintained privately by the class String.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

1. String s=new String("Sachin");
2. String s2=s.intern();
3. System.out.println(s2);//Sachin

**Test it Now**

Sachin

## Java String valueOf() method

The string valueOf() method converts given type such as int, long, float, double, boolean, char and char array into string.

1. int a=10;
2. String s=String.valueOf(a);
3. System.out.println(s+10);

Output:

```
1010
```

## Java String replace() method

The string replace() method replaces all occurrence of first sequence of character with second sequence of character.

1. String s1="Java is a programming language. Java is a platform. Java is an Island.";
2. String replaceString=s1.replace("Java","Kava");//replaces all occurrences of "Java" to "Kava"
3. System.out.println(replaceString);

Output:

```
Kava is a programming language. Kava is a platform. Kava is an Island.
```

# Java StringBuffer class

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

*Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.*

## Important Constructors of StringBuffer class

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

## Important methods of StringBuffer class

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.

public synchronized StringBuffer	insert(int offset, String s)	is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	is used to return the character at the specified position.

public int	length()	is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

## What is mutable string

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

### 1) StringBuffer append() method

The append() method concatenates the given argument with this string.

```

1.    class StringBufferExample{
2.    public static void main(String args[]){
3.        StringBuffer sb=new StringBuffer("Hello ");
4.        sb.append("Java");//now original string is changed
5.        System.out.println(sb);//prints Hello Java
6.    }
7.    }
```



## 2) StringBuffer insert() method

The insert() method inserts the given string with this string at the given position.

```
1.      class StringBufferExample2{
2.      public static void main(String args[]){
3.      StringBuffer sb=new StringBuffer("Hello ");
4.      sb.insert(1,"Java");//now original string is changed
5.      System.out.println(sb);//prints HJavaello
6.      }
7.      }
```

## 3) StringBuffer replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

```
1.      class StringBufferExample3{
2.      public static void main(String args[]){
3.      StringBuffer sb=new StringBuffer("Hello");
4.      sb.replace(1,3,"Java");
5.      System.out.println(sb);//prints HJavallo
6.      }
7.      }
```

## 4) StringBuffer delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
1.      class StringBufferExample4{
```

```
2.    public static void main(String args[]){
3.        StringBuffer sb=new StringBuffer("Hello");
4.        sb.delete(1,3);
5.        System.out.println(sb);//prints Hlo
6.    }
7.    }
```

## 5) StringBuffer reverse() method

The reverse() method of StringBuffer class reverses the current string.

```
1.    class StringBufferExample5{
2.        public static void main(String args[]){
3.            StringBuffer sb=new StringBuffer("Hello");
4.            sb.reverse();
5.            System.out.println(sb);//prints olleH
6.        }
7.    }
```

## Java StringBuilder class

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

## Important Constructors of StringBuilder class

Constructor	Description
-------------	-------------

<code>StringBuilder()</code>	creates an empty string Builder with the initial capacity of 16.
<code>StringBuilder(String str)</code>	creates a string Builder with the specified string.
<code>StringBuilder(int length)</code>	creates an empty string Builder with the specified capacity as length.

## Important methods of StringBuilder class

Method	Description
<code>public StringBuilder append(String s)</code>	is used to append the specified string with this string. The <code>append()</code> method is overloaded like <code>append(char)</code> , <code>append(boolean)</code> , <code>append(int)</code> , <code>append(float)</code> , <code>append(double)</code> etc.
<code>public StringBuilder insert(int offset, String s)</code>	is used to insert the specified string with this string at the specified position. The <code>insert()</code> method is overloaded like <code>insert(int, char)</code> , <code>insert(int, boolean)</code> , <code>insert(int, int)</code> , <code>insert(int, float)</code> , <code>insert(int, double)</code> etc.
<code>public StringBuilder replace(int startIndex, int endIndex, String str)</code>	is used to replace the string from specified <code>startIndex</code> and <code>endIndex</code> .

<code>public StringBuilder delete(int startIndex, int endIndex)</code>	is used to delete the string from specified startIndex and endIndex.
<code>public StringBuilder reverse()</code>	is used to reverse the string.
<code>public int capacity()</code>	is used to return the current capacity.
<code>public void ensureCapacity(int minimumCapacity)</code>	is used to ensure the capacity at least equal to the given minimum.
<code>public char charAt(int index)</code>	is used to return the character at the specified position.
<code>public int length()</code>	is used to return the length of the string i.e. total number of characters.
<code>public String substring(int beginIndex)</code>	is used to return the substring from the specified beginIndex.
<code>public String substring(int beginIndex, int endIndex)</code>	is used to return the substring from the specified beginIndex and endIndex.

## Java StringBuilder Examples

Let's see the examples of different methods of StringBuilder class.

## 1) StringBuilder append() method

The StringBuilder append() method concatenates the given argument with this string.

```
1.      class StringBuilderExample{
2.      public static void main(String args[]){
3.      StringBuilder sb=new StringBuilder("Hello ");
4.      sb.append("Java");//now original string is changed
5.      System.out.println(sb);//prints Hello Java
6.      }
7.      }
```

## 2) StringBuilder insert() method

The StringBuilder insert() method inserts the given string with this string at the given position.

```
1.      class StringBuilderExample2{
2.      public static void main(String args[]){
3.      StringBuilder sb=new StringBuilder("Hello ");
4.      sb.insert(1,"Java");//now original string is changed
5.      System.out.println(sb);//prints HJavaello
6.      }
7.      }
```

## 3) StringBuilder replace() method

The StringBuilder replace() method replaces the given string from the specified beginIndex and endIndex.

```
1.      class StringBuilderExample3{
2.      public static void main(String args[]){
```

```
3.     StringBuilder sb=new StringBuilder("Hello");
4.     sb.replace(1,3,"Java");
5.     System.out.println(sb);//prints HJavallo
6.     }
7.     }
```

#### 4) StringBuilder delete() method

The delete() method of StringBuilder class deletes the string from the specified beginIndex to endIndex.

```
1.     class StringBuilderExample4{
2.     public static void main(String args[]){
3.     StringBuilder sb=new StringBuilder("Hello");
4.     sb.delete(1,3);
5.     System.out.println(sb);//prints Hlo
6.     }
7.     }
```

#### 5) StringBuilder reverse() method

The reverse() method of StringBuilder class reverses the current string.

```
1.     class StringBuilderExample5{
2.     public static void main(String args[]){
3.     StringBuilder sb=new StringBuilder("Hello");
4.     sb.reverse();
5.     System.out.println(sb);//prints olleH
6.     }
7.     }
```

## Difference between String and StringBuffer

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

No.	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you concat strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

## Difference between StringBuffer and StringBuilder

Java provides three classes to represent a sequence of characters: String, StringBuffer, and StringBuilder. The String class is an immutable class whereas StringBuffer and StringBuilder classes are mutable. There are many differences between StringBuffer and StringBuilder. The StringBuilder class is introduced since JDK 1.5.

A list of differences between StringBuffer and StringBuilder are given below:

No.	StringBuffer	StringBuilder
1)	StringBuffer is <i>synchronized</i> i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is <i>non-synchronized</i> i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is <i>less efficient</i> than StringBuilder.	StringBuilder is <i>more efficient</i> than StringBuffer.

## Java toString() method

If you want to represent any object as a string, **toString() method** comes into existence.

The toString() method returns the string representation of the object.

If you print any object, java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depends on your implementation.

### Advantage of Java toString() method

By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.

---



## Understanding problem without toString() method

Let's see the simple code that prints reference.

```
1.  class Student{
2.      int rollno;
3.      String name;
4.      String city;
5.
6.      Student(int rollno, String name, String city){
7.          this.rollno=rollno;
8.          this.name=name;
9.          this.city=city;
10.     }
11.
12.     public static void main(String args[]){
13.         Student s1=new Student(101,"Raj","lucknow");
14.         Student s2=new Student(102,"Vijay","ghaziabad");
15.
16.         System.out.println(s1);//compiler writes here s1.toString()
17.         System.out.println(s2);//compiler writes here s2.toString()
18.     }
19. }
```

```
Output:Student@1fee6fc
        Student@1eed786
```

As you can see in the above example, printing s1 and s2 prints the hashcode values of the objects but I want to print the values of these objects. Since java compiler internally calls toString() method, overriding this method will return the

specified values. Let's understand it with the example given below:

## Example of Java toString() method

Now let's see the real example of toString() method.

```
1.      class Student{
2.      int rollno;
3.      String name;
4.      String city;
5.
6.      Student(int rollno, String name, String city){
7.      this.rollno=rollno;
8.      this.name=name;
9.      this.city=city;
10.     }
11.
12.     public String toString(){//overriding the toString() method
13.     return rollno+" "+name+" "+city;
14.     }
15.     public static void main(String args[]){
16.     Student s1=new Student(101,"Raj","lucknow");
17.     Student s2=new Student(102,"Vijay","ghaziabad");
18.
19.     System.out.println(s1);//compiler writes here s1.toString()
20.     System.out.println(s2);//compiler writes here s2.toString()
21. }
```

22.        }

```
Output:101 Raj lucknow
       102 Vijay Ghaziabad
```

## StringTokenizer in Java

1. [StringTokenizer](#)
2. [Methods of StringTokenizer](#)
3. [Example of StringTokenizer](#)

The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class. We will discuss about the StreamTokenizer class in I/O chapter.

### *Constructors of StringTokenizer class*

There are 3 constructors defined in the StringTokenizer class.

Constructor	Description
StringTokenizer(String str)	creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	creates StringTokenizer with specified string and delimiter.

StringTokenizer(String str, String delim, boolean returnValue)	creates StringTokenizer with specified string, delimiter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.
--	---

### *Methods of StringTokenizer class*

The 6 useful methods of StringTokenizer class are as follows:

Public method	Description
boolean hasMoreTokens()	checks if there is more tokens available.
String nextToken()	returns the next token from the StringTokenizer object.
String nextToken(String delim)	returns the next token based on the delimiter.
boolean hasMoreElements()	same as hasMoreTokens() method.
Object nextElement()	same as nextToken() but its return type is Object.

int countTokens()	returns the total number of tokens.
-------------------	-------------------------------------

## Java String charAt()

The **java string charAt()** method returns *a char value at the given index number*.

The index number starts from 0 and goes to n-1, where n is length of the string. It returns **StringIndexOutOfBoundsException** if given index number is greater than or equal to this string length or a negative number.

### Java String charAt() method example

```
1. public class CharAtExample{
2.     public static void main(String args[]){
3.         String name="javatpoint";
4.         char ch=name.charAt(4);//returns the char value at the 4th index
5.         System.out.println(ch);
6.     }}
```

Output:

t

# Java String compareTo()

The **java string compareTo()** method compares the given string with current string lexicographically. It returns positive number, negative number or 0.

It compares strings on the basis of Unicode value of each character in the strings.

If first string is lexicographically greater than second string, it returns positive number (difference of character value). If first string is less than second string lexicographically, it returns negative number and if first string is lexicographically equal to second string, it returns 0.

1. **if** s1 > s2, it returns positive number
2. **if** s1 < s2, it returns negative number
3. **if** s1 == s2, it returns 0

## Java String compareTo() method example

1. **public class** CompareToExample{
2. **public static void** main(String args[]){
3. String s1="hello";
4. String s2="hello";
5. String s3="meklo";
6. String s4="hemlo";
7. String s5="flag";
8. System.out.println(s1.compareTo(s2));*//0 because both are equal*
9. System.out.println(s1.compareTo(s3));*//-5 because "h" is 5 times lower than "m"*
10. System.out.println(s1.compareTo(s4));*//-1 because "l" is 1 times lower than "m"*
11. System.out.println(s1.compareTo(s5));*//2 because "h" is 2 times greater than "f"*
12. }}

Output:

```
0  
-5  
-1  
2
```

## Java String concat

The **java string concat()** method *combines specified string at the end of this string*. It returns combined string. It is like appending another string.

### Java String concat() method example

```
1.    public class ConcatExample{  
2.    public static void main(String args[]){  
3.        String s1="java string";  
4.        s1.concat("is immutable");  
5.        System.out.println(s1);  
6.        s1=s1.concat(" is immutable so assign it explicitly");  
7.        System.out.println(s1);  
8.    }}
```

```
java string  
java string is immutable so assign it explicitly
```

### Java String concat() Method Example 2

Let's see an example where we are concatenating multiple string objects.

```
1.      public class ConcatExample2 {
2.          public static void main(String[] args) {
3.              String str1 = "Hello";
4.              String str2 = "Javatpoint";
5.              String str3 = "Reader";
6.              // Concatenating one string
7.              String str4 = str1.concat(str2);
8.              System.out.println(str4);
9.              // Concatenating multiple strings
10.             String str5 = str1.concat(str2).concat(str3);
11.             System.out.println(str5);
12.         }
13.     }
```

Output:

```
HelloJavatpoint
HelloJavatpointReader
```

## Java String concat() Method Example 3

Let's see an example where we are concatenating spaces and special chars to the string object.

```
1.      public class ConcatExample3 {
2.          public static void main(String[] args) {
3.              String str1 = "Hello";
4.              String str2 = "Javatpoint";
5.              String str3 = "Reader";
6.              // Concatenating Space among strings
7.              String str4 = str1.concat(" ").concat(str2).concat(" ").concat(str3);
```



```
8.         System.out.println(str4);
9.         // Concatenating Special Chars
10.        String str5 = str1.concat("!!!");
11.        System.out.println(str5);
12.        String str6 = str1.concat("@").concat(str2);
13.        System.out.println(str6);
14.    }
15. }
```

Output:

```
Hello Javatpoint Reader
Hello!!!
Hello@Javatpoint
```

## Java String contains()

The **java string contains()** method searches the sequence of characters in this string. It returns *true* if sequence of char values are found in this string otherwise returns *false*.

### Java String contains() method example

```
1.     class ContainsExample{
2.     public static void main(String args[]){
3.         String name="what do you know about me";
4.         System.out.println(name.contains("do you know"));
5.         System.out.println(name.contains("about"));
6.         System.out.println(name.contains("hello"));
7.     }}
```

```
true
true
false
```

## Java String contains() Method Example 2

The contains() method searches case sensitive char sequence. If the argument is not case sensitive, it returns false. Let's see an example below.

```
1.    public class ContainsExample2 {
2.        public static void main(String[] args) {
3.            String str = "Hello Javatpoint readers";
4.            boolean isContains = str.contains("Javatpoint");
5.            System.out.println(isContains);
6.            // Case Sensitive
7.            System.out.println(str.contains("javatpoint")); // false
8.        }
9.    }
```

```
true
false
```

## Java String contains() Method Example 3

The contains() method is helpful to find a char-sequence in the string. We can use it in control structure to produce search based result. Let us see an example below.

```
1.    public class ContainsExample3 {
2.        public static void main(String[] args) {
3.            String str = "To learn Java visit Javatpoint.com";
4.            if(str.contains("Javatpoint.com")) {
```

```
5.         System.out.println("This string contains javatpoint.com");
6.     }else
7.         System.out.println("Result not found");
8.     }
9. }
```

Output:

```
This string contains javatpoint.com
```

## Java String endsWith()

The **java string endsWith()** method checks if this string ends with given suffix. It returns true if this string ends with given suffix else returns false.

### Java String endsWith() method example

```
1.     public class EndsWithExample{
2.     public static void main(String args[]){
3.         String s1="java by javatpoint";
4.         System.out.println(s1.endsWith("t"));
5.         System.out.println(s1.endsWith("point"));
6.     }}
```

Output:

```
true
true
```

### Java String endsWith() Method Example 2

```
1.    public class EndsWithExample2 {
2.        public static void main(String[] args) {
3.            String str = "Welcome to Javatpoint.com";
4.            System.out.println(str.endsWith("point"));
5.            if(str.endsWith(".com")) {
6.                System.out.println("String ends with .com");
7.            }else System.out.println("It does not end with .com");
8.        }
9.    }
```

Output:

```
false
String ends with .com
```

## Java String equals()

The **java string equals()** method compares the two given strings based on the content of the string. If any character is not matched, it returns false. If all characters are matched, it returns true.

The String equals() method overrides the equals() method of Object class.

## Java String equals() method example

```
1.    public class EqualsExample{
2.        public static void main(String args[]){
3.            String s1="javatpoint";
4.            String s2="javatpoint";
5.            String s3="JAVATPOINT";
6.            String s4="python";
7.            System.out.println(s1.equals(s2));//true because content and case is same
```

```
8.      System.out.println(s1.equals(s3)); //false because case is not same
9.      System.out.println(s1.equals(s4)); //false because content is not same
10.     }}
```

```
true
false
false
```

## Java String equals() Method Example 2

The equals() method compares two strings and can be used in if-else control structure.

```
1.      public class EqualsExample {
2.          public static void main(String[] args) {
3.              String s1 = "javatpoint";
4.              String s2 = "javatpoint";
5.              String s3 = "Javatpoint";
6.              System.out.println(s1.equals(s2)); // True because content is same
7.              if (s1.equals(s3)) {
8.                  System.out.println("both strings are equal");
9.              }else System.out.println("both strings are unequal");
10.         }
11.     }
```

```
true
both strings are unequal
```

## Java String equalsIgnoreCase()

The **String equalsIgnoreCase()** method compares the two given strings on the basis of content of the string irrespective of case of the string. It is like equals() method but doesn't check case. If any character is not matched, it returns false otherwise it returns true.

## Java String equalsIgnoreCase() method example

```
1.    public class EqualsIgnoreCaseExample{
2.    public static void main(String args[]){
3.        String s1="javatpoint";
4.        String s2="javatpoint";
5.        String s3="JAVATPOINT";
6.        String s4="python";
7.        System.out.println(s1.equalsIgnoreCase(s2));//true because content and case both are same
8.        System.out.println(s1.equalsIgnoreCase(s3));//true because case is ignored
9.        System.out.println(s1.equalsIgnoreCase(s4));//false because content is not same
10.    }}
```

```
true
true
false
```

## Java String getBytes()

The **java string getBytes()** method returns the byte array of the string. In other words, it returns sequence of bytes.

## Java String getBytes() method example

```
1.    public class StringGetBytesExample{
2.    public static void main(String args[]){
3.        String s1="ABCDEFGFG";
4.        byte[] barr=s1.getBytes();
5.        for(int i=0;i<barr.length;i++){
```

```
6.      System.out.println(barr[i]);
7.      }
8.      }}
```

Output:

```
65
66
67
68
69
70
71
```

## Java String getBytes() Method Example 2

This method returns a byte array that again can be passed to String constructor to get String.

```
1.      public class StringGetBytesExample2 {
2.          public static void main(String[] args) {
3.              String s1 = "ABCDEFGH";
4.              byte[] barr = s1.getBytes();
5.              for(int i=0;i<barr.length;i++){
6.                  System.out.println(barr[i]);
7.              }
8.              // Getting string back
9.              String s2 = new String(barr);
10.             System.out.println(s2);
11.         }
12.     }
```

Output:

```
65
66
67
68
69
70
71
ABCDEFG
```

## Java String getChars()

The **java string getChars()** method copies the content of this string into specified char array. There are 4 arguments passed in getChars() method. The signature of getChars() method is given below:

### Signature

The signature or syntax of string getChars() method is given below:

1. **public void** getChars(**int** srcBeginIndex, **int** srcEndIndex, **char**[] destination, **int** dstBeginIndex)
- 

### Returns

It doesn't return any value.

---

### Throws

It throws StringIndexOutOfBoundsException if beginIndex is greater than endIndex.



---

## Java String getChars() method example

```
1.    public class StringGetCharsExample{
2.    public static void main(String args[]){
3.        String str = new String("hello javatpoint how r u");
4.        char[] ch = new char[10];
5.        try{
6.            str.getChars(6, 16, ch, 0);
7.            System.out.println(ch);
8.        }catch(Exception ex){System.out.println(ex);}
9.    }
```

**Test it Now**

Output:

```
javatpoint
```

## Java String getChars() Method Example 2

It throws an exception if index value exceeds array range. Let's see an example.

```
1.    public class StringGetCharsExample2 {
2.    public static void main(String[] args) {
3.        String str = new String("Welcome to Javatpoint");
4.        char[] ch = new char[20];
5.        try {
6.            str.getChars(1, 26, ch, 0);
7.            System.out.println(ch);
```

```
8.         } catch (Exception e) {  
9.             System.out.println(e);  
10.        }  
11.    }  
12. }
```

Output:

```
java.lang.StringIndexOutOfBoundsException: offset 10, count 14, length 20
```

## Java String indexOf()

The **java string indexOf()** method returns index of given character value or substring. If it is not found, it returns -1. The index counter starts from zero.

### Signature

There are 4 types of indexOf method in java. The signature of indexOf methods are given below:

No.	Method	Description
1	int indexOf(int ch)	returns index position for the given char value
2	int indexOf(int ch, int fromIndex)	returns index position for the given char value and from index
3	int indexOf(String substring)	returns index position for the given substring

4	int indexOf(String substring, int fromIndex)	returns index position for the given substring and from index
---	--	---

## Java String indexOf() method example

```

1.  public class IndexOfExample{
2.  public static void main(String args[]){
3.  String s1="this is index of example";
4.  //passing substring
5.  int index1=s1.indexOf("is");//returns the index of is substring
6.  int index2=s1.indexOf("index");//returns the index of index substring
7.  System.out.println(index1+" "+index2);//2 8
8.
9.  //passing substring with from index
10. int index3=s1.indexOf("is",4);//returns the index of is substring after 4th index
11. System.out.println(index3);//5 i.e. the index of another is
12.
13. //passing char value
14. int index4=s1.indexOf('s');//returns the index of s char value
15. System.out.println(index4);//3
16. }}
```

```

2  8
5
3
```

## Java String indexOf(String substring) Method Example

This method takes substring as an argument and returns index of first character of the substring.

```

1.  public class IndexOfExample2 {
2.      public static void main(String[] args) {
3.          String s1 = "This is indexOf method";
4.          // Passing Substring
5.          int index = s1.indexOf("method"); //Returns the index of this substring
6.          System.out.println("index of substring "+index);
7.      }
8.
9.  }

```

```
index of substring 16
```

## Java String indexOf(String substring, int fromIndex) Method Example

This method takes substring and index as arguments and returns index of first character occurred after the given *fromIndex*.

```

1.  public class IndexOfExample3 {
2.      public static void main(String[] args) {
3.          String s1 = "This is indexOf method";
4.          // Passing substring and index
5.          int index = s1.indexOf("method", 10); //Returns the index of this substring
6.          System.out.println("index of substring "+index);
7.          index = s1.indexOf("method", 20); // It returns -1 if substring does not found
8.          System.out.println("index of substring "+index);
9.      }
10. }

```

```
index of substring 16
index of substring -1
```

## Java String indexOf(int char, int fromIndex) Method Example

This method takes char and index as arguments and returns index of first character occurred after the given *fromIndex*.

```
1.    public class IndexOfExample4 {
2.        public static void main(String[] args) {
3.            String s1 = "This is indexOf method";
4.            // Passing char and index from
5.            int index = s1.indexOf('e', 12); //Returns the index of this char
6.            System.out.println("index of char "+index);
7.        }
8.    }
```

```
index of char 17
```

## Java String isEmpty()

The **java string isEmpty()** method checks if this string is empty or not. It returns *true*, if length of string is 0 otherwise *false*. In other words, true is returned if string is empty otherwise it returns false.

## Java String isEmpty() method example

```
1.    public class IsEmptyExample{
2.        public static void main(String args[]){
3.            String s1="";
4.            String s2="javatpoint";
5.
6.            System.out.println(s1.isEmpty());
7.            System.out.println(s2.isEmpty());
8.        }}
```

```
true  
false
```

## Java String isEmpty() Method Example 2

```
1.      public class IsEmptyExample2 {  
2.          public static void main(String[] args) {  
3.              String s1="";  
4.              String s2="Javatpoint";  
5.              // Either length is zero or isEmpty is true  
6.              if(s1.length()==0 || s1.isEmpty())  
7.                  System.out.println("String s1 is empty");  
8.              else System.out.println("s1");  
9.              if(s2.length()==0 || s2.isEmpty())  
10.                  System.out.println("String s2 is empty");  
11.              else System.out.println(s2);  
12.          }  
13.      }
```

```
String s1 is empty  
Javatpoint
```

## Java String join()

The **java string join()** method returns a string joined with given delimiter. In string join method, delimiter is copied for each elements.

In case of null element, "null" is added. The join() method is included in java string since JDK 1.8.

There are two types of join() methods in java string.

## Java String join() method example

```
1. public class StringJoinExample{
2.     public static void main(String args[]){
3.         String joinString1=String.join("-", "welcome", "to", "javatpoint");
4.         System.out.println(joinString1);
5.     }}
```

```
welcome-to-javatpoint
```

## Java String join() Method Example 2

We can use delimiter to format the string as we did in the below example to show date and time.

```
1. public class StringJoinExample2 {
2.     public static void main(String[] args) {
3.         String date = String.join("/", "25", "06", "2018");
4.         System.out.print(date);
5.         String time = String.join(":", "12", "10", "10");
6.         System.out.println(" "+time);
7.     }
8. }
```

```
25/06/2018 12:10:10
```

## Java String lastIndexOf()

**The java string lastIndexOf() method returns last index of the given character value or substring. If it is not found, it returns -1. The index counter starts from zero** **Signature**

There are 4 types of lastIndexOf method in java. The signature of lastIndexOf methods are given below:

No.	Method	Description
1	int lastIndexOf(int ch)	returns last index position for the given char value
2	int lastIndexOf(int ch, int fromIndex)	returns last index position for the given char value and from index
3	int lastIndexOf(String substring)	returns last index position for the given substring
4	int lastIndexOf(String substring, int fromIndex)	returns last index position for the given substring and from index

## Signature

There are 4 types of lastIndexOf method in java. The signature of lastIndexOf methods are given below:

No.	Method	Description
1	int lastIndexOf(int ch)	returns last index position for the given char value



2	int lastIndexOf(int ch, int fromIndex)	returns last index position for the given char value and from index
3	int lastIndexOf(String substring)	returns last index position for the given substring
4	int lastIndexOf(String substring, int fromIndex)	returns last index position for the given substring and from index

## ava String lastIndexOf() method example

```

1. public class LastIndexOfExample{
2.     public static void main(String args[]){
3.         String s1="this is index of example";//there are 2 's' characters in this sentence
4.         int index1=s1.lastIndexOf('s');//returns last index of 's' char value
5.         System.out.println(index1);//6
6.     }}

```

Output:

```
6
```

## Java String lastIndexOf(int ch, int fromIndex) Method Example

Here, we are finding last index from the string by specifying *fromIndex*

```

1. public class LastIndexOfExample2 {

```

```
2.      public static void main(String[] args) {  
3.          String str = "This is index of example";  
4.          int index = str.lastIndexOf('s',5);  
5.          System.out.println(index);  
6.      }  
7.  }
```

Output:

```
3
```

---

## Java String lastIndexOf(String substring) Method Example

It returns the last index of the substring.

```
1.      public class LastIndexOfExample3 {  
2.          public static void main(String[] args) {  
3.              String str = "This is last index of example";  
4.              int index = str.lastIndexOf("of");  
5.              System.out.println(index);  
6.          }  
7.      }
```

Output:

```
19
```

## Java String lastIndexOf(String substring, int fromIndex) Method Example

It returns the last index of the substring from the *fromIndex*.

```
1.    public class LastIndexOfExample4 {
2.        public static void main(String[] args) {
3.            String str = "This is last index of example";
4.            int index = str.lastIndexOf("of", 25);
5.            System.out.println(index);
6.            index = str.lastIndexOf("of", 10);
7.            System.out.println(index); // -1, if not found
8.        }
9.    }
```

Output:

```
19
-1
```

## Java String length()

The **java string length()** method length of the string. It returns count of total number of characters. The length of java string is same as the unicode code units of the string.

### Java String length() method example

```
1.    public class LengthExample{
2.        public static void main(String args[]){
3.            String s1="javatpoint";
```

```
4.      String s2="python";
5.      System.out.println("string length is: "+s1.length());//10 is the length of javatpoint string
6.      System.out.println("string length is: "+s2.length());//6 is the length of python string
7.      }}
```

```
string length is: 10
string length is: 6
```

## Java String replace()

The **java string replace()** method returns a string replacing all the old char or CharSequence to new char or CharSequence.

Since JDK 1.5, a new replace() method is introduced, allowing you to replace a sequence of char values.

### Java String replace(char old, char new) method example

```
1.      public class ReplaceExample1{
2.      public static void main(String args[]){
3.      String s1="javatpoint is a very good website";
4.      String replaceString=s1.replace('a','e');//replaces all occurrences of 'a' to 'e'
5.      System.out.println(replaceString);
6.      }}
```

```
jevetpoint is e very good website
```

### Java String replace(CharSequence target, CharSequence replacement) method example

```
1.      public class ReplaceExample2{
2.      public static void main(String args[]){
3.      String s1="my name is khan my name is java";
```

```
4.      String replaceString=s1.replace("is","was");//replaces all occurrences of "is" to "was"
5.      System.out.println(replaceString);
6.      }}
```

```
my name was khan my name was java
```

## Java String replace() Method Example 3

```
1.      public class ReplaceExample3 {
2.          public static void main(String[] args) {
3.              String str = "oooooo-hhhh-oooooo";
4.              String rs = str.replace("h","s"); // Replace 'h' with 's'
5.              System.out.println(rs);
6.              rs = rs.replace("s","h"); // Replace 's' with 'h'
7.              System.out.println(rs);
8.          }
9.      }
```

```
oooooo-ssss-oooooo
oooooo-hhhh-oooooo
```

## Java String replaceAll()

The **java string replaceAll()** method returns a string replacing all the sequence of characters matching regex and replacement string.

### ava String replaceAll() example: replace character

Let's see an example to replace all the occurrences of **a single character**.

```
1.      public class ReplaceAllExample1{
```

```
2.    public static void main(String args[]){
3.    String s1="javatpoint is a very good website";
4.    String replaceString=s1.replaceAll("a","e");//replaces all occurrences of "a" to "e"
5.    System.out.println(replaceString);
6.    }}
```

```
jvetpoint is e very good website
```

## Java String replaceAll() example: replace word

Let's see an example to replace all the occurrences of **single word or set of words**.

```
1.    public class ReplaceAllExample2{
2.    public static void main(String args[]){
3.    String s1="My name is Khan. My name is Bob. My name is Sonoo.";
4.    String replaceString=s1.replaceAll("is","was");//replaces all occurrences of "is" to "was"
5.    System.out.println(replaceString);
6.    }}
```

```
My name was Khan. My name was Bob. My name was Sonoo.
```

## Java String split()

The **java string split()** method splits this string against given regular expression and returns a char array.

## Java String split() method example

The given example returns total number of words in a string excluding space only. It also includes special characters.

```
1.    public class SplitExample{
```

```
2.    public static void main(String args[]){
3.        String s1="java string split method by javatpoint";
4.        String[] words=s1.split("\\s");//splits the string based on whitespace
5.        //using java foreach loop to print elements of string array
6.        for(String w:words){
7.            System.out.println(w);
8.        }
9.    }}
```

```
java
string
split
method
by
javatpoint
```

## Java String startsWith()

The **java string startsWith()** method checks if this string starts with given prefix. It returns true if this string starts with given prefix else returns false.

## Java String startsWith() method example

```
1.    public class StartsWithExample{
2.        public static void main(String args[]){
3.            String s1="java string split method by javatpoint";
4.            System.out.println(s1.startsWith("ja"));
5.            System.out.println(s1.startsWith("java string"));
6.        }}
```

Output:

```
true
true
```

## Java String startsWith(String prefix, int offset) Method Example

This is overloaded method of `startsWith()` method which is used to pass one extra argument (offset) to the function. This method works from the passed offset. Let's see an example.

```
1.      public class StartsWithExample2 {
2.          public static void main(String[] args) {
3.              String str = "Javatpoint";
4.              System.out.println(str.startsWith("J")); // True
5.              System.out.println(str.startsWith("a")); // False
6.              System.out.println(str.startsWith("a",1)); // True
7.          }
8.      }
```

Output:

```
true
false
true
```

## Java String substring()

The **java string substring()** method returns a part of the string.

We pass begin index and end index number position in the java substring method where start index is inclusive and end index is exclusive. In other words, start index starts from 0 whereas end index starts from 1.



## Java String substring() method example

```
1.    public class SubstringExample{
2.    public static void main(String args[]){
3.        String s1="javatpoint";
4.        System.out.println(s1.substring(2,4));//returns va
5.        System.out.println(s1.substring(2));//returns vatpoint
6.    }}
```

```
va
vatpoint
```

## Java String toCharArray()

The **java string toCharArray()** method converts this string into character array. It returns a newly created character array, its length is similar to this string and its contents are initialized with the characters of this string.

## Java String toCharArray() method example

```
1.    public class StringToCharArrayExample{
2.    public static void main(String args[]){
3.        String s1="hello";
4.        char[] ch=s1.toCharArray();
5.        for(int i=0;i<ch.length;i++){
6.            System.out.print(ch[i]);
7.        }
8.    }}
```

Output:

```
hello
```

## Java String toCharArray() Method Example 2

Let's see one more example of char array. It is useful method which returns char array from the string without writing any custom code.

```
1.      public class StringToCharArrayExample2 {
2.          public static void main(String[] args) {
3.              String s1 = "Welcome to Javatpoint";
4.              char[] ch = s1.toCharArray();
5.              int len = ch.length;
6.              System.out.println("Char Array length: " + len);
7.              System.out.println("Char Array elements: ");
8.              for (int i = 0; i < len; i++) {
9.                  System.out.println(ch[i]);
10.             }
11.         }
12.     }
```

Output:

```
Char Array length: 21
Char Array elements:
W
e
l
c
o
m
e
t
```

## Java String toLowerCase()

The **java string toLowerCase()** method returns the string in lowercase letter. In other words, it converts all characters of the string into lower case letter.

The toLowerCase() method works same as toLowerCase(Locale.getDefault()) method. It internally uses the default locale.

### Java String toLowerCase() method example

```
1.    public class StringLowerExample{  
2.    public static void main(String args[]){  
3.        String s1="JAVATPOINT HELLO stRIng";  
4.        String s1lower=s1.toLowerCase();  
5.        System.out.println(s1lower);  
6.    }}
```

Output:

```
javatpoint hello string
```

## Java String toUpperCase()

The **java string toUpperCase()** method returns the string in uppercase letter. In other words, it converts all characters of the string into upper case letter.

The toUpperCase() method works same as toUpperCase(Locale.getDefault()) method. It internally uses the default locale.

### Java String toUpperCase() method example

```
1.      public class StringUpperExample{
2.      public static void main(String args[]){
3.      String s1="hello string";
4.      String s1upper=s1.toUpperCase();
5.      System.out.println(s1upper);
6.      }}
```

Output:

```
HELLO STRING
```

## Java String trim()

The **java string trim()** method eliminates leading and trailing spaces. The unicode value of space character is '\u0020'. The trim() method in java string checks this unicode value before and after the string, if it exists then removes the spaces and returns the omitted string.

*The string trim() method doesn't omits middle spaces.*

### Java String trim() method example

```
1.    public class StringTrimExample{
2.    public static void main(String args[]){
3.    String s1=" hello string ";
4.    System.out.println(s1+"javatpoint");//without trim()
5.    System.out.println(s1.trim()+"javatpoint");//with trim()
6.    }}
```

```
hello string   javatpoint
hello stringjavatpoint
```

## Java String valueOf()

The **java string valueOf()** method converts different types of values into string. By the help of string valueOf() method, you can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

### Java String valueOf() method example

```
1.    public class StringValueOfExample{
2.    public static void main(String args[]){
3.    int value=30;
4.    String s1=String.valueOf(value);
5.    System.out.println(s1+10);//concatenating string with 10
6.    }}
```

Output:

```
3010
```

# Exception Handling in Java

The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

In this page, we will learn about Java exceptions, its type and the difference between checked and unchecked exceptions.

## What is Exception in Java

**Dictionary Meaning:** Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

---

## What is Exception Handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

### Advantage of Exception Handling

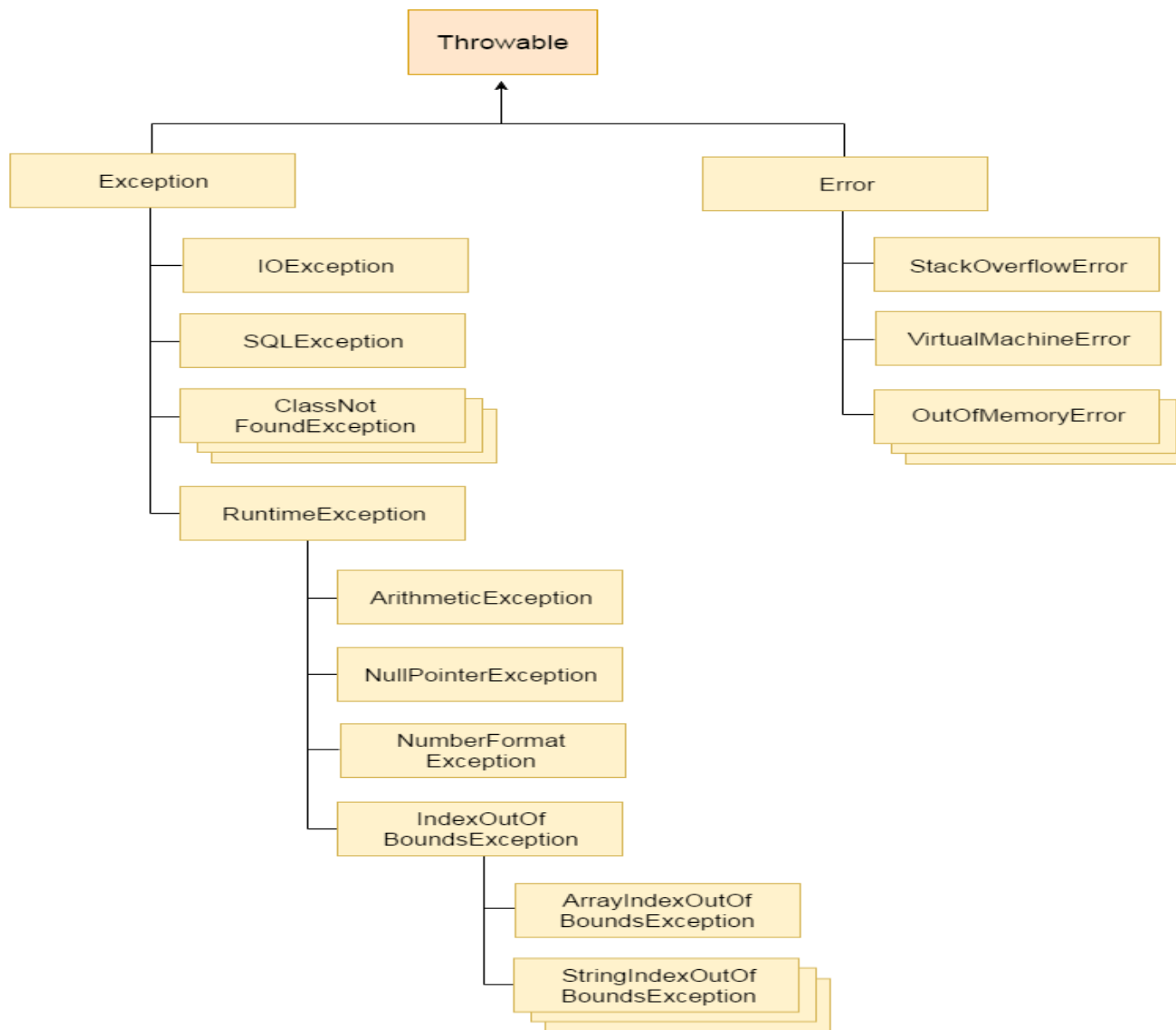
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

```
statement 1;  
statement 2;  
statement 3;  
statement 4;  
statement 5;//exception occurs  
statement 6;  
statement 7;  
statement 8;  
statement 9;  
statement 10;
```

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in **Java**

## Hierarchy of Java Exception classes

The `java.lang.Throwable` class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:





## Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error



# Difference between Checked and Unchecked Exceptions

## 1) Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

## 2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

## 3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

# Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.

catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

## Java Exception Handling Example

Let's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.

```
1. public class JavaExceptionExample{
2.     public static void main(String args[]){
3.         try{
4.             //code that may raise exception
5.             int data=100/0;
6.         }catch(ArithmeticException e){System.out.println(e);}
7.         //rest code of the program
8.         System.out.println("rest of the code...");
```

9.            }

10.           }

**Test it Now**

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero  
rest of the code...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

## Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

### 1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

1.            **int** a=50/0;//ArithmeticException

### 2) A scenario where NullPointerException occurs

If we have a null value in any **variable**, performing any operation on the variable throws a NullPointerException.

1. String s=**null**;
2. System.out.println(s.length());**//NullPointerException**

### 3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a **string** variable that has characters, converting this variable into digit will occur NumberFormatException.

1. String s="**abc**";
2. **int** i=Integer.parseInt(s);**//NumberFormatException**

### 4) A scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

1. **int** a[]=**new int**[**5**];
2. a[**10**]=**50**; **//ArrayIndexOutOfBoundsException**

## Java try-catch block

### Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

## Syntax of Java try-catch

1. **try**{
2. *//code that may throw an exception*
3. **}catch**(Exception\_class\_Name ref){}

## Syntax of try-finally block

1. **try**{
2. *//code that may throw an exception*
3. **}finally**{}

## Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception ( i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

## Problem without exception handling

Let's try to understand the problem if we don't use a try-catch block.

## Example 1

```
1.      public class TryCatchExample1 {  
2.  
3.          public static void main(String[] args) {  
4.  
5.              int data=50/0; //may throw exception  
6.  
7.              System.out.println("rest of the code");  
8.  
9.          }  
10.  
11.     }
```

**Test it Now**

### Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.



## Solution by exception handling

Let's see the solution of the above problem by a java try-catch block.

### Example 2

```
1.      public class TryCatchExample2 {
2.
3.          public static void main(String[] args) {
4.              try
5.              {
6.                  int data=50/0; //may throw exception
7.              }
8.              //handling the exception
9.              catch(ArithmeticException e)
10.             {
11.                 System.out.println(e);
12.             }
13.             System.out.println("rest of the code");
14.         }
15.
16.     }
```

**Test it Now**

**Output:**



```
java.lang.ArithmeticException: / by zero  
rest of the code
```

Now, as displayed in the above example, the **rest of the code** is executed, i.e., the **rest of the code** statement is printed.

## Example 3

In this example, we also kept the code in a try block that will not throw an exception.

```
1.      public class TryCatchExample3 {  
2.  
3.          public static void main(String[] args) {  
4.              try  
5.              {  
6.                  int data=50/0; //may throw exception  
7.                      // if exception occurs, the remaining statement will not execute  
8.                  System.out.println("rest of the code");  
9.              }  
10.             // handling the exception  
11.             catch(ArithmeticException e)  
12.             {  
13.                 System.out.println(e);  
14.             }  
15.  
16.     }
```

17.  
18. }

**Test it Now**

### Output:

```
java.lang.ArithmeticException: / by zero
```

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

## Example 4

Here, we handle the exception using the parent class exception.

```
1. public class TryCatchExample4 {  
2.  
3.     public static void main(String[] args) {  
4.         try  
5.         {  
6.             int data=50/0; //may throw exception  
7.         }  
8.         // handling the exception by using Exception class  
9.         catch(Exception e)  
10.        {  
11.            System.out.println(e);  
12.        }  
13.        System.out.println("rest of the code");
```

```
14.     }
15.
16.     }
```

**Test it Now**

### Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

## Example 5

Let's see an example to print a custom message on exception.

```
1.     public class TryCatchExample5 {
2.
3.         public static void main(String[] args) {
4.             try
5.             {
6.                 int data=50/0; //may throw exception
7.             }
8.             // handling the exception
9.             catch(Exception e)
10.            {
11.                // displaying the custom message
12.                System.out.println("Can't divided by zero");
```

```
13.         }
14.     }
15.
16. }
```

**Test it Now**

### Output:

```
Can't divided by zero
```

## Example 6

Let's see an example to resolve the exception in a catch block.

```
1.     public class TryCatchExample6 {
2.
3.         public static void main(String[] args) {
4.             int i=50;
5.             int j=0;
6.             int data;
7.             try
8.             {
9.                 data=i/j; //may throw exception
10.            }
11.            // handling the exception
12.            catch(Exception e)
```

```
13.      {
14.          // resolving the exception in catch block
15.          System.out.println(i/(j+2));
16.      }
17.  }
18. }
```

**Test it Now**

**Output:**

25

## Example 7

In this example, along with try block, we also enclose exception code in a catch block.

```
1.      public class TryCatchExample7 {
2.
3.          public static void main(String[] args) {
4.
5.              try
6.              {
7.                  int data1=50/0; //may throw exception
8.
9.              }
10.         // handling the exception
```

```
11.         catch(Exception e)
12.         {
13.             // generating the exception in catch block
14.             int data2=50/0; //may throw exception
15.
16.         }
17.     System.out.println("rest of the code");
18. }
19. }
```

**Test it Now**

### Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.

## Example 8

In this example, we handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).

```
1.     public class TryCatchExample8 {
2.
3.         public static void main(String[] args) {
4.             try
```

```
5.      {
6.      int data=50/0; //may throw exception
7.
8.      }
9.      // try to handle the ArithmeticException using ArrayIndexOutOfBoundsException
10.     catch(ArrayIndexOutOfBoundsException e)
11.     {
12.         System.out.println(e);
13.     }
14.     System.out.println("rest of the code");
15. }
16.
17. }
```

**Test it Now**

### Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

## Example 9

Let's see an example to handle another unchecked exception.

```
1.     public class TryCatchExample9 {
2.
3.     public static void main(String[] args) {
```

```
4.      try
5.      {
6.      int arr[] = {1,3,5,7};
7.      System.out.println(arr[10]); //may throw exception
8.      }
9.      // handling the array exception
10.     catch(ArrayIndexOutOfBoundsException e)
11.     {
12.         System.out.println(e);
13.     }
14.     System.out.println("rest of the code");
15. }
16.
17. }
```

**Test it Now**

### Output:

```
java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code
```

## Example 10

Let's see an example to handle checked exception.

```
1.      import java.io.FileNotFoundException;
```




```
2.      import java.io.PrintWriter;
3.
4.      public class TryCatchExample10 {
5.
6.          public static void main(String[] args) {
7.
8.
9.              PrintWriter pw;
10.             try {
11.                 pw = new PrintWriter("jtp.txt"); //may throw exception
12.                 pw.println("saved");
13.             }
14.             // providing the checked exception handler
15.             catch (FileNotFoundException e) {
16.
17.                 System.out.println(e);
18.             }
19.             System.out.println("File saved successfully");
20.         }
21.     }
```

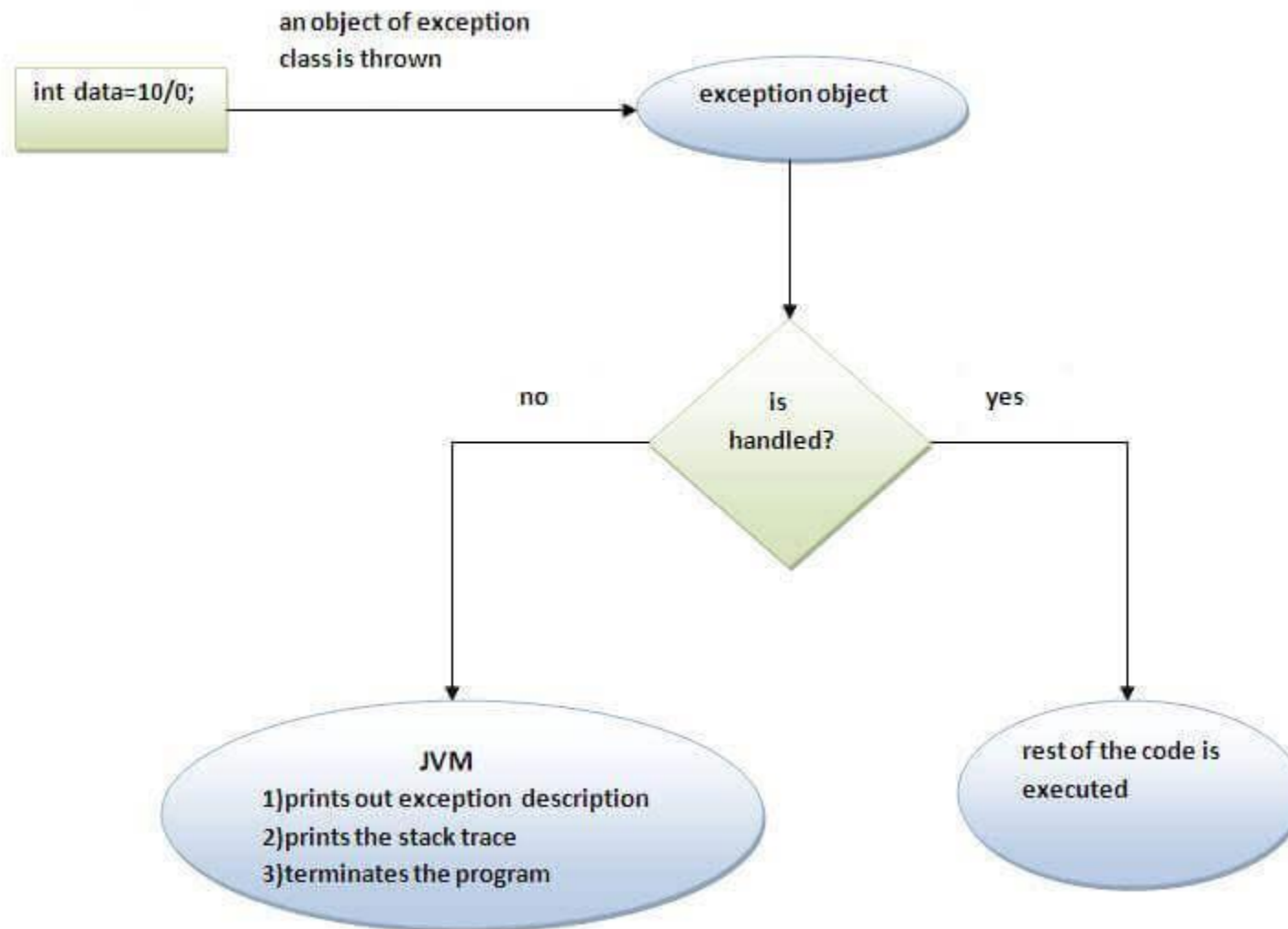
**Test it Now**

### Output:

```
File saved successfully
```



## Internal working of java try-catch block



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

## Java catch multiple exceptions

### Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

### Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for `ArithmeticException` must come before catch for `Exception`.

### Example 1

Let's see a simple example of java multi-catch block.

```
1.      public class MultipleCatchBlock1 {
2.
3.          public static void main(String[] args) {
4.
5.              try{
6.                  int a[]=new int[5];
7.                  a[5]=30/0;
8.              }
9.              catch(ArithmeticException e)
10.                 {
11.                     System.out.println("Arithmetic Exception occurs");
12.                 }
13.              catch(ArrayIndexOutOfBoundsException e)
14.                 {
15.                     System.out.println("ArrayIndexOutOfBoundsException Exception occurs");
16.                 }
17.              catch(Exception e)
18.                 {
19.                     System.out.println("Parent Exception occurs");
20.                 }
21.              System.out.println("rest of the code");
22.          }
```

23. }

**Test it Now**

### Output:

```
Arithmetic Exception occurs  
rest of the code
```

## Example 2

```
1. public class MultipleCatchBlock2 {  
2.  
3.     public static void main(String[] args) {  
4.  
5.         try{  
6.             int a[]=new int[5];  
7.  
8.             System.out.println(a[10]);  
9.         }  
10.        catch(ArithmeticException e)  
11.        {  
12.            System.out.println("Arithmetic Exception occurs");  
13.        }  
14.        catch(ArrayIndexOutOfBoundsException e)  
15.        {  
16.            System.out.println("ArrayIndexOutOfBoundsException occurs");
```

```
17.         }
18.         catch(Exception e)
19.         {
20.             System.out.println("Parent Exception occurs");
21.         }
22.         System.out.println("rest of the code");
23.     }
24. }
```

**Test it Now**

### Output:

```
ArrayIndexOutOfBoundsException Exception occurs
rest of the code
```

## Example 3

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is invoked.

```
1.     public class MultipleCatchBlock3 {
2.
3.         public static void main(String[] args) {
4.
5.             try{
6.                 int a[]=new int[5];
7.                 a[5]=30/0;
```

```
8.         System.out.println(a[10]);
9.     }
10.    catch(ArithmeticException e)
11.    {
12.        System.out.println("Arithmetic Exception occurs");
13.    }
14.    catch(ArrayIndexOutOfBoundsException e)
15.    {
16.        System.out.println("ArrayIndexOutOfBounds Exception occurs");
17.    }
18.    catch(Exception e)
19.    {
20.        System.out.println("Parent Exception occurs");
21.    }
22.    System.out.println("rest of the code");
23. }
24. }
```

**Test it Now**

### Output:

```
Arithmetic Exception occurs
rest of the code
```

## Example 4

In this example, we generate `NullPointerException`, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class **Exception** will be invoked.

```
1. public class MultipleCatchBlock4 {
2.
3.     public static void main(String[] args) {
4.
5.         try{
6.             String s=null;
7.             System.out.println(s.length());
8.         }
9.         catch(ArithmeticException e)
10.            {
11.                System.out.println("Arithmetic Exception occurs");
12.            }
13.        catch(ArrayIndexOutOfBoundsException e)
14.            {
15.                System.out.println("ArrayIndexOutOfBoundsException occurs");
16.            }
17.        catch(Exception e)
18.            {
19.                System.out.println("Parent Exception occurs");
20.            }
21.        System.out.println("rest of the code");
22.    }
23. }
```

Test it Now



### Output:

```
Parent Exception occurs  
rest of the code
```

## Example 5

Let's see an example, to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).

```
1.    class MultipleCatchBlock5{  
2.        public static void main(String args[]){  
3.            try{  
4.                int a[]=new int[5];  
5.                a[5]=30/0;  
6.            }  
7.            catch(Exception e){System.out.println("common task completed");}  
8.            catch(ArithmeticException e){System.out.println("task1 is completed");}  
9.            catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}  
10.        System.out.println("rest of the code...");  
11.    }  
12. }
```

**Test it Now**

### Output:

```
Compile-time error
```

# Java Nested try block

The try block within a try block is known as nested try block in java.

## Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

## Syntax:

```
1. ....
2. try
3. {
4.     statement 1;
5.     statement 2;
6.     try
7.     {
8.         statement 1;
9.         statement 2;
10.    }
```

```
11.      catch(Exception e)
12.      {
13.      }
14.  }
15.      catch(Exception e)
16.      {
17.      }
18.      ....
```

## Java nested try example

Let's see a simple example of java nested try block.

```
1.      class Excep6{
2.      public static void main(String args[]){
3.      try{
4.      try{
5.      System.out.println("going to divide");
6.      int b =39/0;
7.      }catch(ArithmeticException e){System.out.println(e);}
8.
9.      try{
10.     int a[]=new int[5];
11.     a[5]=4;
```

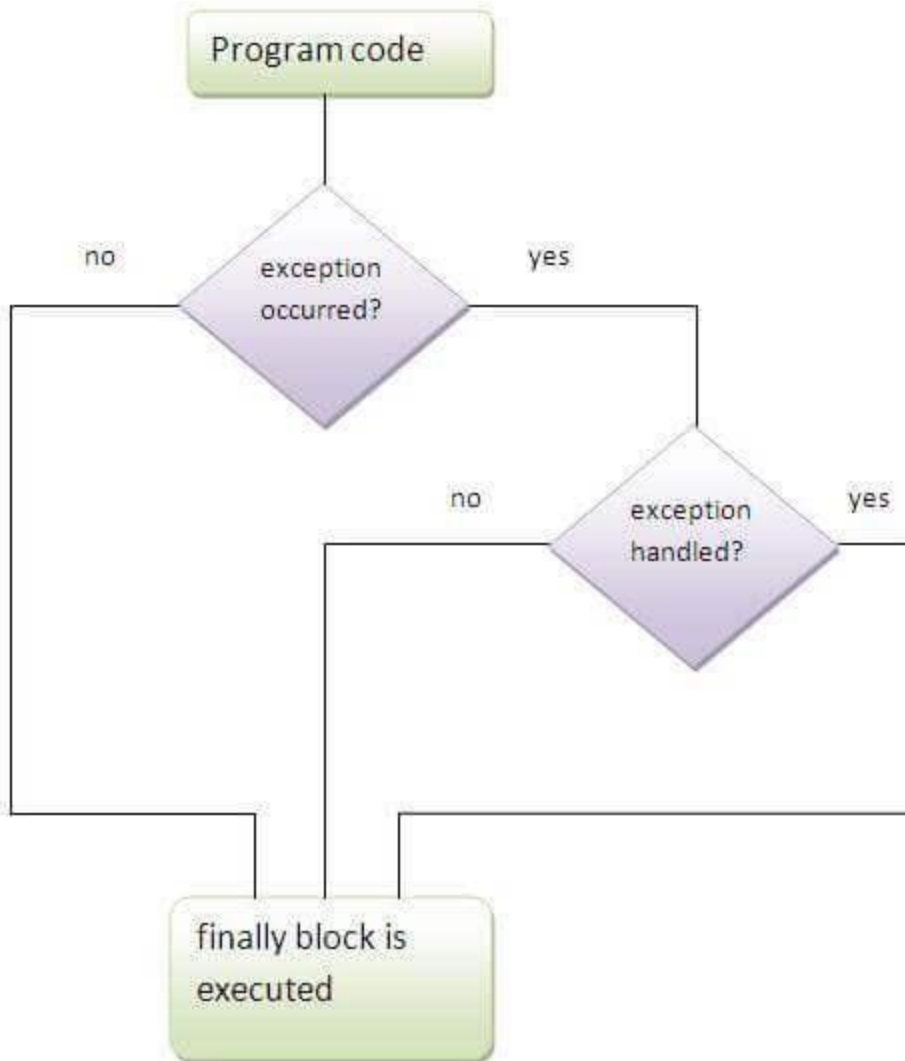
```
12.         }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
13.
14.         System.out.println("other statement");
15.     }catch(Exception e){System.out.println("handeled");}
16.
17.         System.out.println("normal flow..");
18.     }
19. }
```

## Java finally block

**Java finally block** is a block that is used to execute important code such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.



*Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).*

## Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

## Usage of Java finally

Let's see the different cases where java finally block can be used.

### Case 1

Let's see the java finally example where **exception doesn't occur**.

```
1.  class TestFinallyBlock{
2.      public static void main(String args[]){
3.          try{
4.              int data=25/5;
5.              System.out.println(data);
6.          }
7.          catch(NullPointerException e){System.out.println(e);}
8.          finally{System.out.println("finally block is always executed");}
9.          System.out.println("rest of the code...");
10.     }
```

11. }

### Test it Now

Output:5

finally block is always executed

rest of the code...

## Case 2

Let's see the java finally example where **exception occurs and not handled**.

```
1. class TestFinallyBlock1{
2.     public static void main(String args[]){
3.         try{
4.             int data=25/0;
5.             System.out.println(data);
6.         }
7.         catch(NullPointerException e){System.out.println(e);}
8.         finally{System.out.println("finally block is always executed");}
9.         System.out.println("rest of the code...");
10.    }
11. }
```

### Test it Now

Output:finally block is always executed

Exception in thread main java.lang.ArithmeticException:/ by zero

## Case 3

Let's see the java finally example where **exception occurs and handled**.

```
1.  public class TestFinallyBlock2{
2.      public static void main(String args[]){
3.          try{
4.              int data=25/0;
5.              System.out.println(data);
6.          }
7.          catch(ArithmeticException e){System.out.println(e);}
8.          finally{System.out.println("finally block is always executed");}
9.          System.out.println("rest of the code...");
10.     }
11. }
```

#### Test it Now

```
Output:Exception in thread main java.lang.ArithmeticException:/ by zero
        finally block is always executed
        rest of the code...
```

**Rule:** For each try block there can be zero or more catch blocks, but only one finally block.

**Note:** The finally block will not be executed if program exits(either by calling `System.exit()` or by causing a fatal error that causes the process to abort).



# Java throw exception

## Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

1. **throw** exception;

Let's see the example of throw IOException.

1. **throw new** IOException("sorry device error");

## java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
1. public class TestThrow1{
2.     static void validate(int age){
3.         if(age<18)
4.             throw new ArithmeticException("not valid");
5.         else
6.             System.out.println("welcome to vote");
7.     }
8.     public static void main(String args[]){
9.         validate(13);
10.        System.out.println("rest of the code..");
11.    }
12. }
```

**Test it Now**

Output:

```
Exception in thread main java.lang.ArithmeticException: not valid
```

## Java Exception propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are

caught or until they reach the very bottom of the call stack. This is called exception propagation.

*Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).*

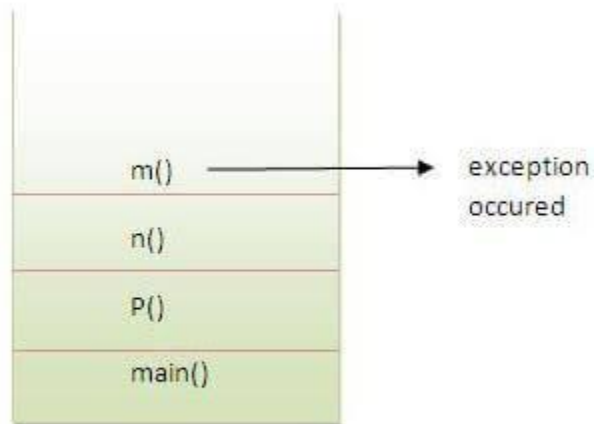
### **Program of Exception Propagation**

```
1.  class TestExceptionPropagation1{
2.      void m(){
3.          int data=50/0;
4.      }
5.      void n(){
6.          m();
7.      }
8.      void p(){
9.          try{
10.             n();
11.         }catch(Exception e){System.out.println("exception handled");}
12.     }
13.     public static void main(String args[]){
14.         TestExceptionPropagation1 obj=new TestExceptionPropagation1();
15.         obj.p();
16.         System.out.println("normal flow...");
17.     }
18. }
```

**Test it Now**

Output:exception handled

normal flow...



Call Stack

In the above example exception occurs in m() method where it is not handled,so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

**Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).**

**Program which describes that checked exceptions are not propagated**

1. **class** TestExceptionPropagation2{
2. **void** m(){

```
3.         throw new java.io.IOException("device error");//checked exception
4.     }
5.     void n(){
6.         m();
7.     }
8.     void p(){
9.         try{
10.            n();
11.        }catch(Exception e){System.out.println("exception handeled");}
12.    }
13.    public static void main(String args[]){
14.        TestExceptionPropagation2 obj=new TestExceptionPropagation2();
15.        obj.p();
16.        System.out.println("normal flow");
17.    }
18. }
```

**Test it Now**

Output:Compile Time Error

## Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

## Syntax of java throws

1.       return\_type method\_name() **throws** exception\_class\_name{
2.       //method code
3.       }

## Which exception should be declared

**Ans)** checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

## Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

## Java throws example

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
1.    import java.io.IOException;
2.    class Testthrows1{
3.        void m()throws IOException{
4.            throw new IOException("device error");//checked exception
5.        }
6.        void n()throws IOException{
7.            m();
8.        }
9.        void p(){
10.           try{
11.               n();
12.           }catch(Exception e){System.out.println("exception handled");}
13.        }
14.        public static void main(String args[]){
15.            Testthrows1 obj=new Testthrows1();
16.            obj.p();
17.            System.out.println("normal flow...");
18.        }
19.    }
```

**Test it Now**

Output:

```
exception handled  
normal flow...
```

*Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.*

There are two cases:

1. **Case1:**You caught the exception i.e. handle the exception using try/catch.
2. **Case2:**You declare the exception i.e. specifying throws with the method.

## Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
1.  import java.io.*;  
2.  class M{  
3.      void method()throws IOException{  
4.          throw new IOException("device error");  
5.      }  
6.  }  
7.  public class Testthrows2{  
8.      public static void main(String args[]){  
9.          try{
```



```

10.         M m=new M();
11.         m.method();
12.     }catch(Exception e){System.out.println("exception handled");}
13.
14.         System.out.println("normal flow...");
15.     }
16. }

```

### Test it Now

Output:exception handled

normal flow...

## Case2: You declare the exception

- A)In case you declare the exception, if exception does not occur, the code will be executed fine.
- B)In case you declare the exception if exception occurs, an exception will be thrown at runtime because throws does not handle the exception.

### ***A)Program if exception does not occur***

```

1.     import java.io.*;
2.     class M{
3.         void method()throws IOException{
4.             System.out.println("device operation performed");
5.         }
6.     }

```

```

7.      class Testthrows3{
8.          public static void main(String args[]){throws IOException{//declare exception
9.              M m=new M();
10.             m.method();
11.
12.             System.out.println("normal flow...");
13.         }
14.     }

```

#### Test it Now

Output:device operation performed

normal flow...

#### ***B)Program if exception occurs***

```

1.      import java.io.*;
2.      class M{
3.          void method(){throws IOException{
4.              throw new IOException("device error");
5.          }
6.      }
7.      class Testthrows4{
8.          public static void main(String args[]){throws IOException{//declare exception
9.              M m=new M();
10.             m.method();

```

```
11.  
12.         System.out.println("normal flow...");  
13.     }  
14. }
```

**Test it Now**

Output:Runtime Exception

## Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.

5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.
----	---------------------------------------	---

## Java throw example

```
1.    void m(){  
2.    throw new ArithmeticException("sorry");  
3.    }
```

## Java throws example

```
1.    void m()throws ArithmeticException{  
2.    //method code  
3.    }
```

## Java throw and throws example

```
1.    void m()throws ArithmeticException{  
2.    throw new ArithmeticException("sorry");  
3.    }
```

## Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

## Java final example

```
1.    class FinalExample{  
2.    public static void main(String[] args){  
3.    final int x=100;  
4.    x=200;//Compile Time Error  
5.    }}
```

## Java finally example

```
1.    class FinallyExample{  
2.    public static void main(String[] args){  
3.    try{  
4.    int x=300;  
5.    }catch(Exception e){System.out.println(e);}  
6.    finally{System.out.println("finally block is executed");}  
7.    }}
```

## Java finalize example

```
1.    class FinalizeExample{  
2.    public void finalize(){System.out.println("finalize called");}
```

```
3.    public static void main(String[] args){
4.        FinalizeExample f1=new FinalizeExample();
5.        FinalizeExample f2=new FinalizeExample();
6.        f1=null;
7.        f2=null;
8.        System.gc();
9.    }}
```

## ExceptionHandling with MethodOverriding in Java

There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:

- **If the superclass method does not declare an exception**
  - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
- **If the superclass method declares an exception**
  - If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

### If the superclass method does not declare an exception

1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

```
1.  import java.io.*;
2.  class Parent{
3.      void msg(){System.out.println("parent");}
4.  }
5.
6.  class TestExceptionChild extends Parent{
7.      void msg()throws IOException{
8.          System.out.println("TestExceptionChild");
9.      }
10.     public static void main(String args[]){
11.         Parent p=new TestExceptionChild();
12.         p.msg();
13.     }
14. }
```

Test it Now

Output:Compile Time Error

2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.



```
1.  import java.io.*;
2.  class Parent{
3.      void msg(){System.out.println("parent");}
4.  }
5.
6.  class TestExceptionChild1 extends Parent{
7.      void msg()throws ArithmeticException{
8.          System.out.println("child");
9.      }
10.     public static void main(String args[]){
11.         Parent p=new TestExceptionChild1();
12.         p.msg();
13.     }
14. }
```

#### Test it Now

Output:child

### If the superclass method declares an exception

1) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

### Example in case subclass overridden method declares parent exception

```
1.  import java.io.*;
2.  class Parent{
3.      void msg()throws ArithmeticException{System.out.println("parent");}
4.  }
5.
6.  class TestExceptionChild2 extends Parent{
7.      void msg()throws Exception{System.out.println("child");}
8.
9.      public static void main(String args[]){
10.         Parent p=new TestExceptionChild2();
11.         try{
12.             p.msg();
13.         }catch(Exception e){}
14.     }
15. }
```

**Test it Now**

Output:Compile Time Error

## Example in case subclass overridden method declares same exception

```
1.  import java.io.*;
2.  class Parent{
```

```

3.     void msg()throws Exception{System.out.println("parent");}
4.     }
5.
6.     class TestExceptionChild3 extends Parent{
7.         void msg()throws Exception{System.out.println("child");}
8.
9.         public static void main(String args[]){
10.            Parent p=new TestExceptionChild3();
11.            try{
12.                p.msg();
13.            }catch(Exception e){}
14.        }
15.    }

```

**Test it Now**

Output:child

## Example in case subclass overridden method declares subclass exception

```

1.     import java.io.*;
2.     class Parent{
3.         void msg()throws Exception{System.out.println("parent");}
4.     }
5.
6.     class TestExceptionChild4 extends Parent{

```

```
7.      void msg()throws ArithmeticException{System.out.println("child");}
8.
9.      public static void main(String args[]){
10.         Parent p=new TestExceptionChild4();
11.         try{
12.             p.msg();
13.         }catch(Exception e){}
14.     }
15. }
```

**Test it Now**

Output:child

## Example in case subclass overridden method declares no exception

```
1.      import java.io.*;
2.      class Parent{
3.          void msg()throws Exception{System.out.println("parent");}
4.      }
5.
6.      class TestExceptionChild5 extends Parent{
7.          void msg(){System.out.println("child");}
8.
9.          public static void main(String args[]){
10.             Parent p=new TestExceptionChild5();
```

```
11.      try{
12.          p.msg();
13.      }catch(Exception e){}
14.      }
15.  }
```

**Test it Now**

Output:child

## Java Custom Exception

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

Let's see a simple example of java custom exception.

```
1.      class InvalidAgeException extends Exception{
2.          InvalidAgeException(String s){
3.              super(s);
```

```
4.     }
5.     }
```

```
1.     class TestCustomException1{
2.
3.         static void validate(int age)throws InvalidAgeException{
4.             if(age<18)
5.                 throw new InvalidAgeException("not valid");
6.             else
7.                 System.out.println("welcome to vote");
8.         }
9.
10.        public static void main(String args[]){
11.            try{
12.                validate(13);
13.            }catch(Exception m){System.out.println("Exception occured: "+m);}
14.
15.            System.out.println("rest of the code..");
16.        }
17.    }
```

### Test it Now

```
Output:Exception occured: InvalidAgeException:not valid
       rest of the code...
```

