

Core java

History of Java

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

The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic". Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. Following are given significant points that describe the history of Java.

- 1) James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.
- 2) Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.

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

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

Why was Java named "Oak"?

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

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

Why is Java Programming named "Java"?

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

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

8) Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.

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

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

11) JDK 1.0 was released on January 23, 1996. After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds new features in Java.

What is Java?

Java is a programming language and a platform. Java is a high level, robust, partially 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, James Gosling and his team changed the name from Oak to Java.

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

Language : As english , hindi , marathi is a language , a language is a way by which we communicate to others . Java is a programming language , which simply means that to communicate with computers , java language is used.

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 applications 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 an enterprise application. It has advantages like high-level security, load balancing, and clustering.

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. Example : MS Office can be considered as Java SE.

2) Java EE (Java Enterprise Edition)

It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, etc. For instance : Zoom application is a prime example of Java EE.

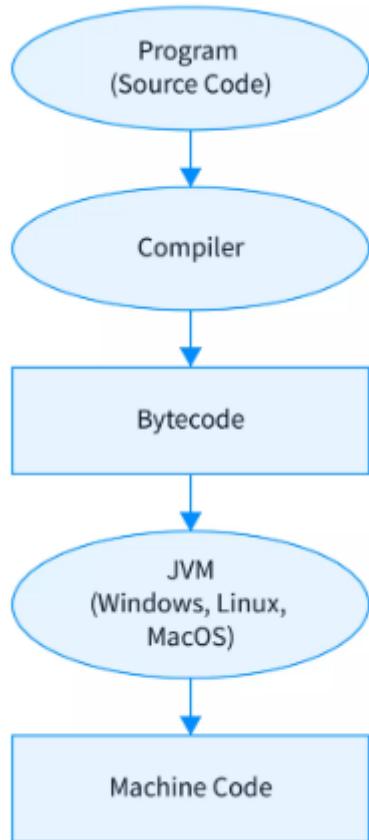
3) Java ME (Java Micro Edition)

It is a micro platform that is dedicated to mobile applications. For instance : Very famous game PubG in mobile format.

4) JavaFX

It is used to develop rich internet applications. It uses a lightweight user interface API. Used in aviation , vehicle fleet data analysis etc.

Before we go ahead , let us understand some important points in this flow :



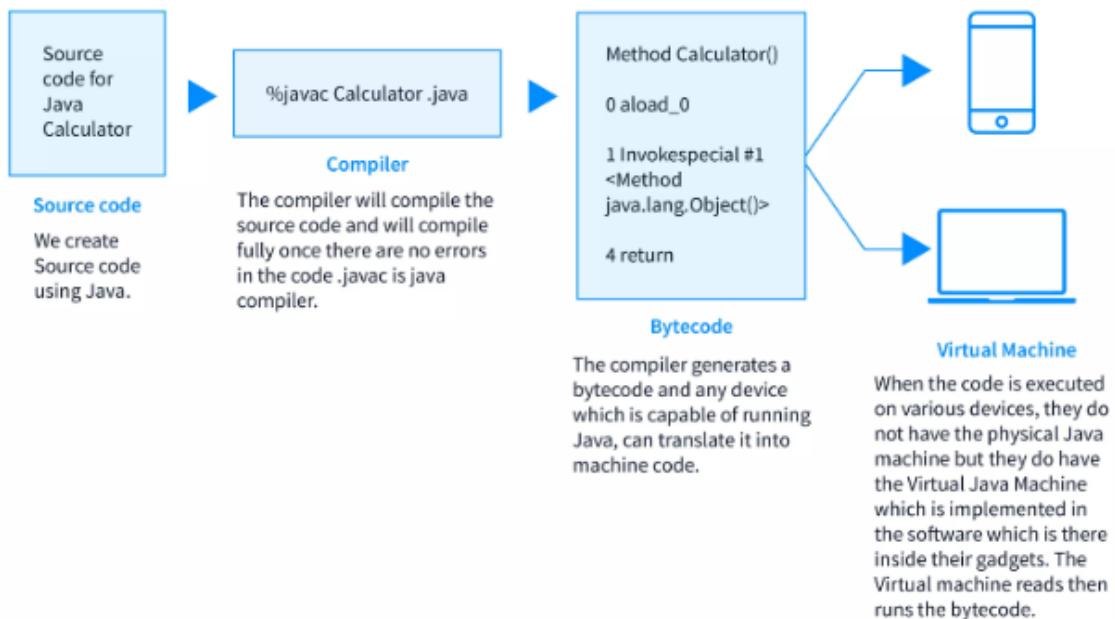
Let us understand the above process with an explanation:

- The code we write in Java is called the source code, which is written in a high-level language. A high-level language is a programmer-friendly language with statements written in English and is closer to human languages. The extension of the Java file is ".java".
- When we compile the program, the compiler compiles the ".java" file and generates a ".class" file. It contains the bytecode.
- The bytecode allows us to run the ".class" file on any other platform.
- But this bytecode requires an interpreter to execute it. Here the JVM comes into the picture. JVM has an interpreter. It executes the code piece by piece, i.e., one statement at a time, until it finds an error or is done with executing the end of the code.

Hence, bytecode is said to be between low-level language and high-level language.

Let us look at an example-

Suppose you have written the code in Java for a calculator app. This is how we can easily understand the way it gets processed.

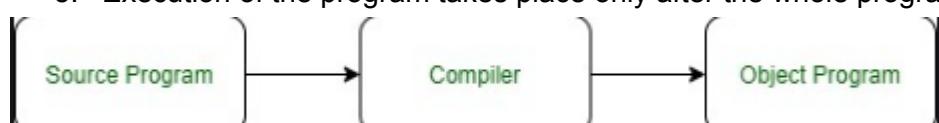


There are 2 words namely : Compiler and interpreter . Let us understand that before we move forward :

Compiler :

It is a translator which takes input i.eHigh Level language and produces an output of low-level language i.e machine or assembly language .

1. It checks all kinds of limits , ranges , errors etc .
2. Its program run time is more and occupies a larger part of memory. It has a slow speed because the compiler goes through the entire program and then translates the entire program into machine codes.
3. It converts the source code into object code.
4. As it scans the code in one go , the errors (if any) are shown at the end together.
5. Execution of the program takes place only after the whole program is compiled.



Interpreter :

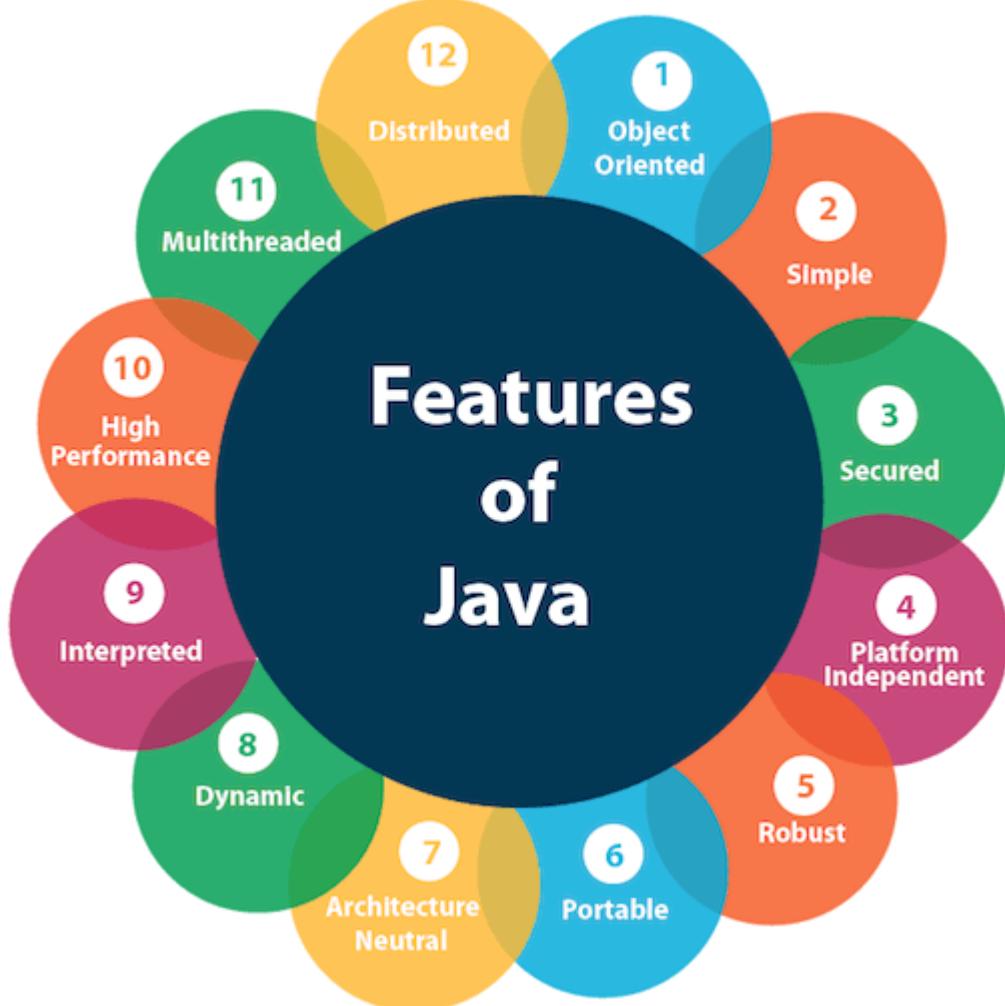
An interpreter is a program that translates a programming language into a comprehensible language .

1. It translates only one statement of the program at a time.
2. More often than not are smaller than compilers.
3. Errors are shown line by line.
4. It does not convert source code into object code instead it scans it line by line.



Features of Java

The primary objective of java programming language creation was to make it a 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.



Simple

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

- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc. (You would learn about this more in coming lectures)
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java. (You would learn about this more in coming lectures)

Object-oriented

Java is a partially object oriented programming language. Everything in Java is an object. Object-oriented means we organise our software as a combination of different types of objects that incorporate both data and behaviour.

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

However you would learn about OOPs in detail in later stages , till then just remember that it follows this methodology .

Platform Independent

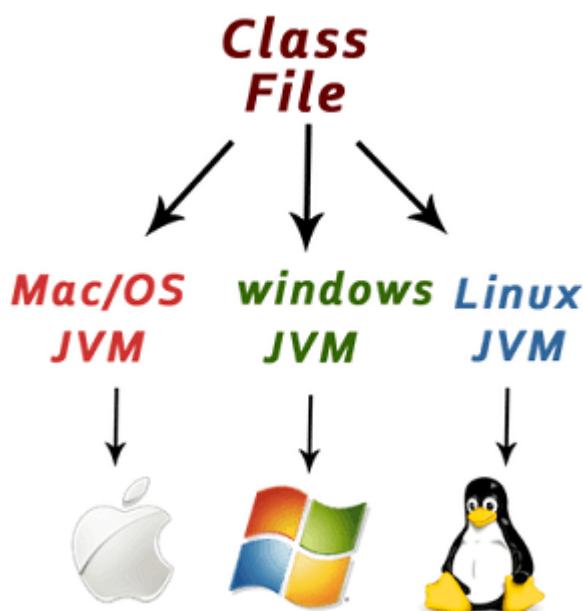
Java is platform independent because it is different from other languages like C , C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

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

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

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

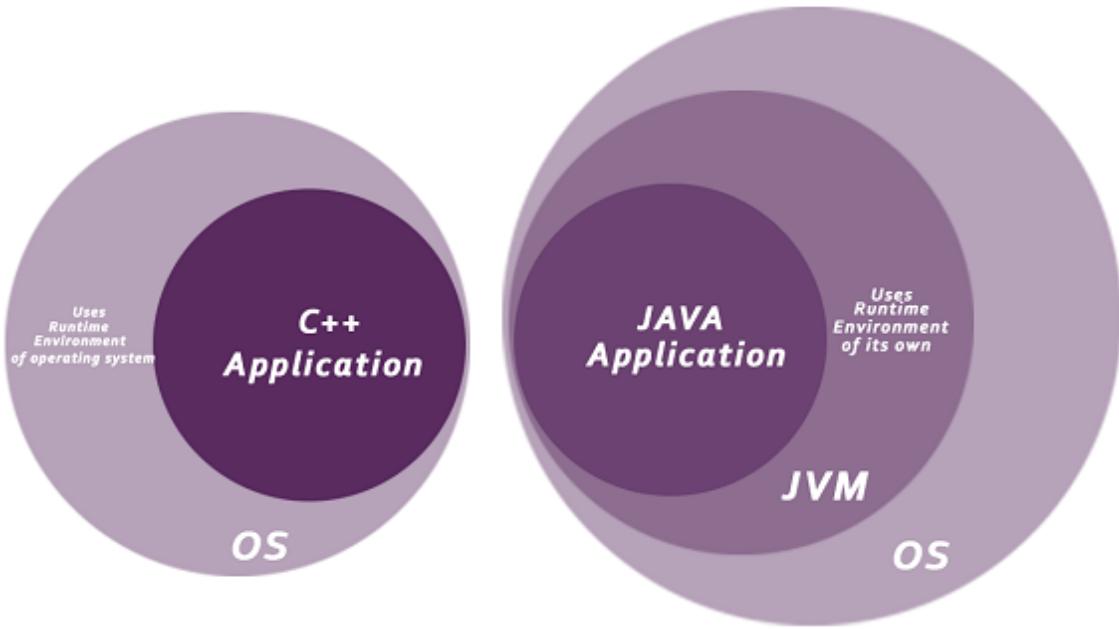
Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).



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.(You will learn more about this in coming lectures)
- Bytecode Verifier: It checks the code fragments for illegal code that can violate access rights to objects.
- Security Manager: It determines what resources a class can access such as reading and writing to the local disk.

Java provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

Robust

The English meaning of Robust is strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- Java provides automatic garbage collection 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.

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 multimedia, Web applications, etc.

Dynamic

Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++. (You would learn more about this in Class-Object chapter)

C++ vs Java

This question is generally asked in your interviews . As a developer having command over multiple languages helps you , these key differences will make you think rationally for the same.

Comparison Index	C++	Java
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in Windows-based, web-based, enterprise, and mobile applications.
Design Goal	C++ was designed for systems and applications programming. It was an extension of the C programming language.	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed to be easy to use and accessible to a broader audience.
Goto	C++ supports the goto statement.	Java doesn't support the goto statement.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by using interfaces in java .
Operator Overloading	C++ supports operator overloading .	Java doesn't support operator overloading.
Pointers	C++ supports pointers . You can write a 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.
Compiler and Interpreter	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 both compiler and interpreter. 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.

Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Thread Support	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
Documentation comment	C++ doesn't support documentation comments.	Java supports documentation comment (<code>/** ... */</code>) to create documentation for java source code.
Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not to 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.
unsigned right shift >>>	C++ doesn't support <code>>>></code> operator.	Java supports unsigned right shift <code>>>></code> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like <code>>></code> operator.
Inheritance Tree	C++ always creates a new inheritance tree.	Java always uses a single inheritance tree because all classes are the child of the Object class in Java. The Object class is the root of the inheritance tree in java.
Hardware	C++ is nearer to hardware.	Java is not so interactive with hardware.
Object-oriented	C++ is an object-oriented language. However, in the C language, a single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from <code>java.lang.Object</code> .

Difference between JDK, JRE, and JVM

We need to understand the difference between all 3 of them before we proceed any further and set up the installation of the java part .

JVM (Java Virtual Machine)

VM (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(Operating system)is different from each other. However, Java is platform independent. There are three notions of the JVM: specification, implementation, and instance.

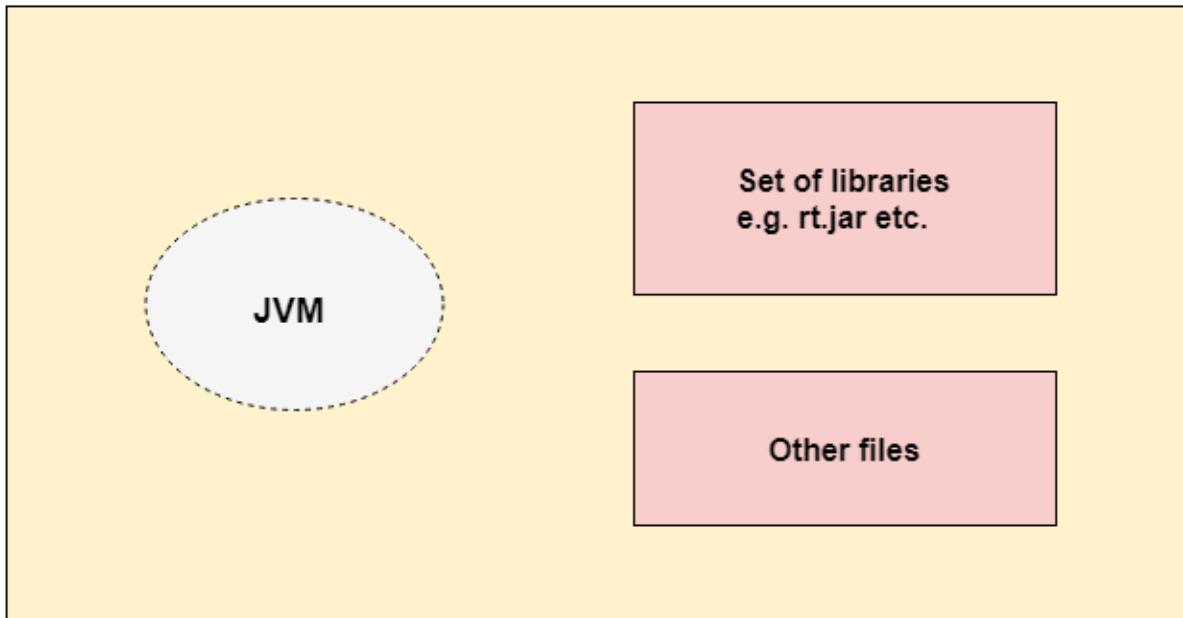
The JVM performs the following main tasks:

- 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.



JRE

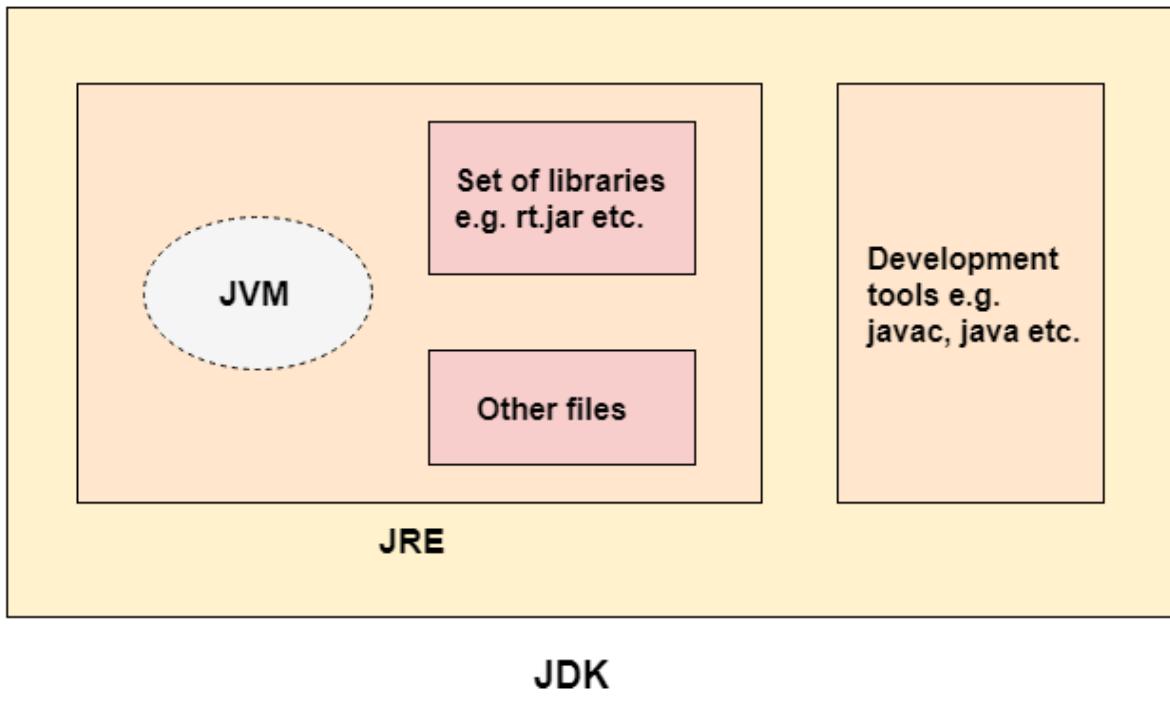
JDK

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

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

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

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



JVM (Java Virtual Machine) Architecture

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

What is JVM

It is:

1. A specification where working of Java Virtual Machine is specified. But the implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
2. Its implementation is known as JRE (Java Runtime Environment).
3. Runtime Instance: Whenever you write a 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

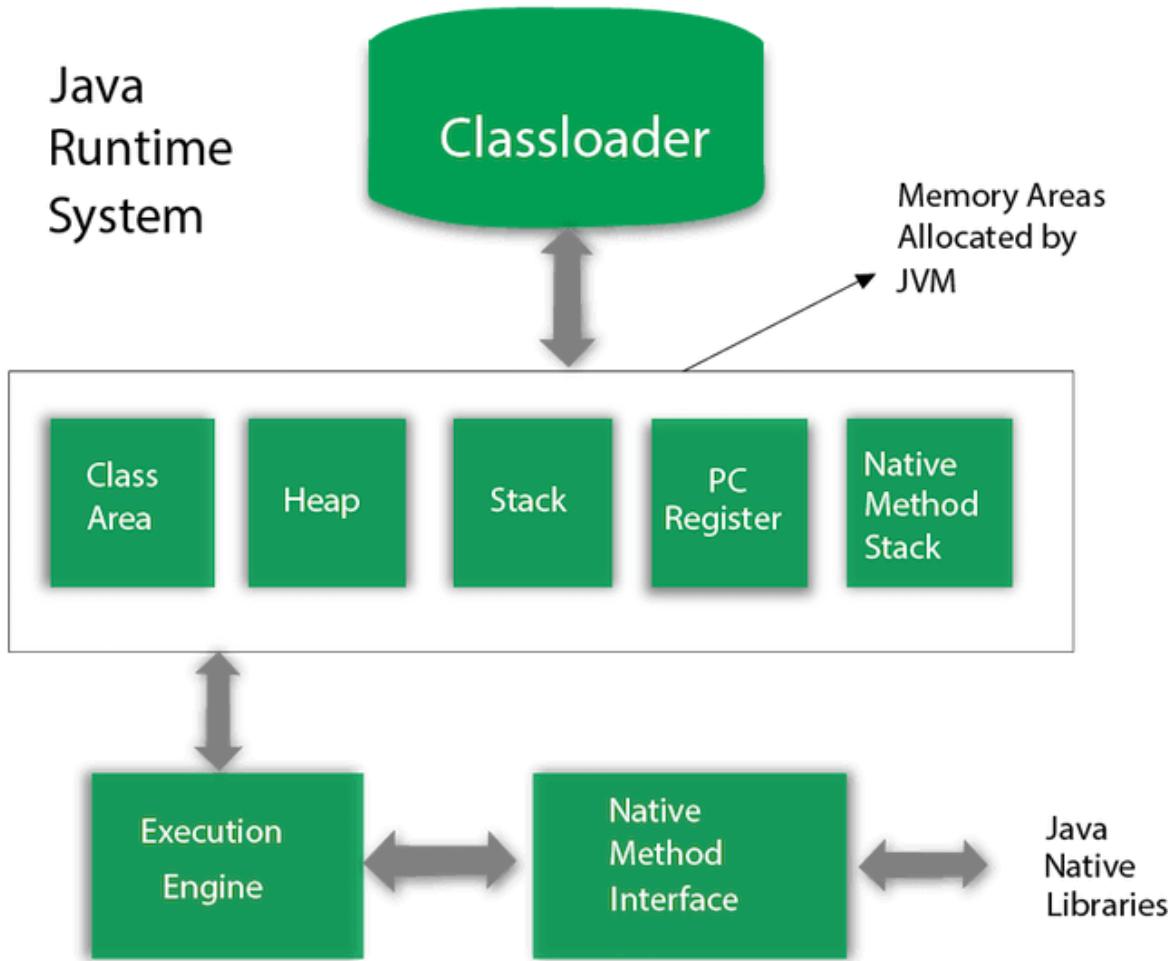
JVM provides definitions for the:

- 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.

Java Variables

A variable is a container which holds the value while the Java Program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

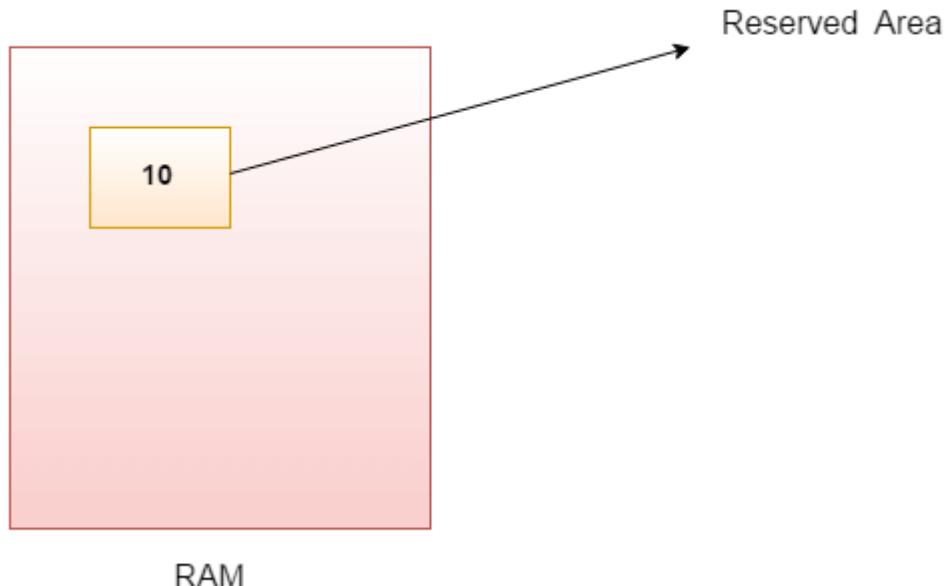
There are two types of data types in Java: primitive and non-primitive.

Variable

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

For example :

In java , we declare it as : Int data=10 ; (int means integer , so you can see in the diagram that data of 10 is stored.

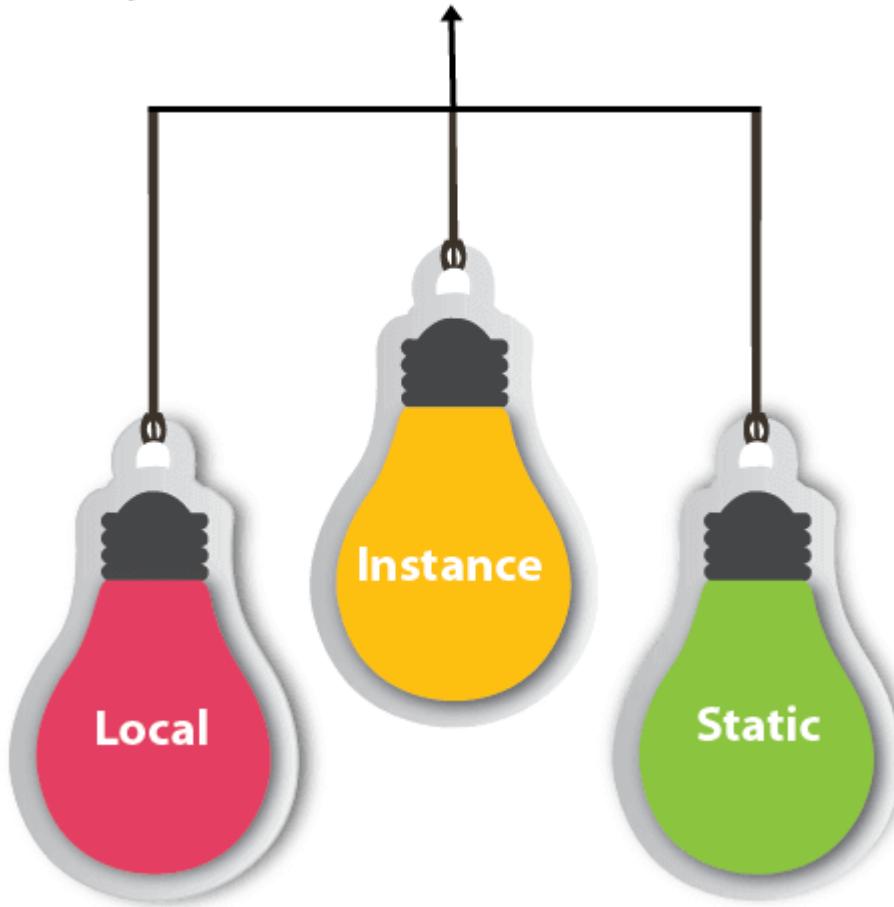


Types of Variables

There are three types of variables in Java:

- local variable
- instance variable
- static variable

Types of Variables



1) Local Variable

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

A local variable cannot be defined with "static" keyword.

2) Instance Variable

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

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

3) Static variable

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

Data Types in Java

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

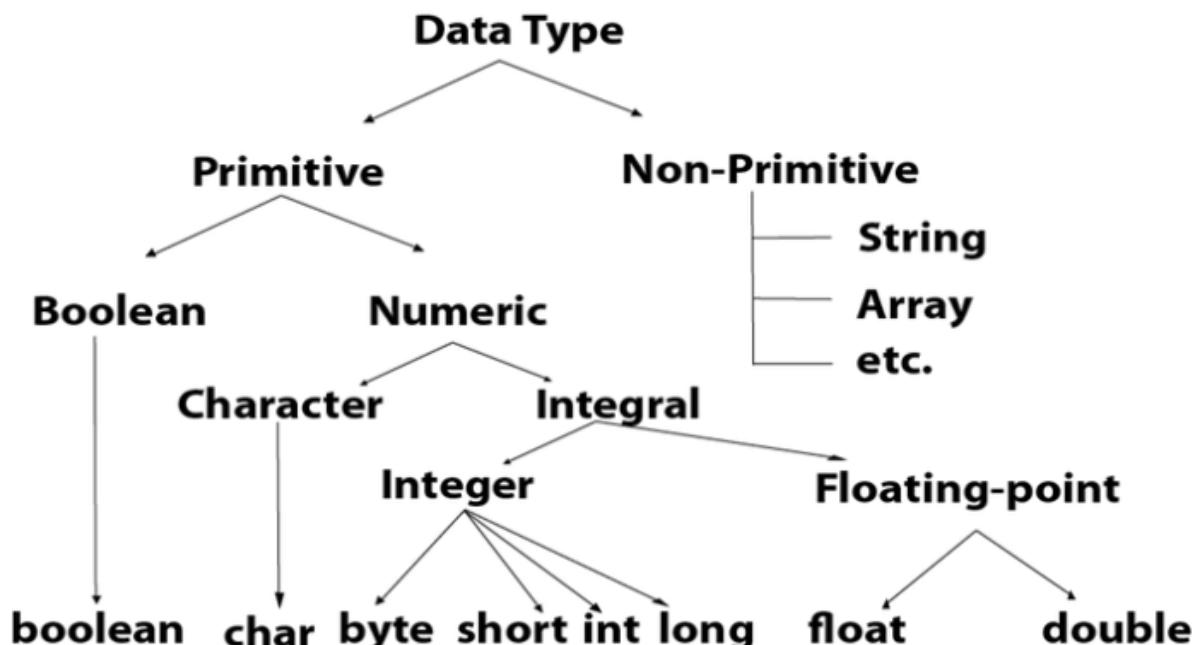
1. Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.
2. Non-primitive data types: The non-primitive data types include classes, interfaces, and arrays.

Java Primitive Data Types

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

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- double data type



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

Boolean Data Type

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

Byte Data Type

The byte data type is an example of primitive data type. It is an 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example:

1. byte a = 10, byte b = -20

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example:

1. short s = 10000, short r = -5000

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between -2,147,483,648 (- 2^{31}) to 2,147,483,647 ($2^{31}-1$) (inclusive). Its minimum value is -2,147,483,648 and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example:

1. `int a = 100000, int b = -200000`

Long Data Type

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

Example:

1. `long a = 100000L, long b = -200000L`

Float Data Type

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

Example:

1. `float f1 = 234.5f`

Double Data Type

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

Example:

`double d1 = 12.3`

Char Data Type

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

Example:

1. `char letterA = 'A'`

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

It is because java uses Unicode system not ASCII code system. The \u0000 is the lowest range of Unicode system

Unicode System

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

Why does Java use the Unicode System?

Before Unicode, there were many language standards:

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

Problem

This caused two problems:

1. A particular code value corresponds to different letters in the various language standards.
2. The encodings for languages with large character sets have variable length. Some common characters are encoded as single bytes, others require two or more bytes.

Solution

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

In Unicode, characters hold 2 bytes, so Java also uses 2 bytes for characters.

lowest value:\u0000

Highest value:\uFFFF

Operators in Java

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

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

Java Operator Precedence

Operator Type	Category	Precedence
Unary	postfix	<i>expr++ expr--</i>
	prefix	<i>++expr --expr +expr -expr ~ !</i>
Arithmetic	multiplicative	<i>* / %</i>
	additive	<i>+ -</i>
Shift	shift	<i><< >> >>></i>
Relational	comparison	<i>< > <= >= instanceof</i>
	equality	<i>== !=</i>
Bitwise	bitwise AND	<i>&</i>
	bitwise exclusive OR	<i>^</i>
	bitwise inclusive OR	<i> </i>
Logical	logical AND	<i>&&</i>
Ternary	ternary	<i>? :</i>
Assignment	assignment	<i>= += -= *= /= %= &= ^= = <<= >>= >>>=</i>

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

For example :

1. int x=10; (declaring of integer value of name x to 10)

System.out.println - It is used to give output in different lines in java .

// - This is called commenting , it is not a part of the program , but here in the example below it is used to tell you what would be its output.

X++ (the value would be incremented by 1 in next line and not there itself)

++X (It means that the value would be incremented then and there itself)

Same with minus signs .

```
System.out.println(x++);//10 (11)
```

```
System.out.println(++x);//12
```

```
System.out.println(x--);//12 (11)
```

```
System.out.println(-x);//10
```

Java Unary Operator

Example 2: ++ and --

```
int a=10;
int b=10;
System.out.println(a++ + ++a);//10+12=22
System.out.println(b++ + b++);//10+11=21
```

Java Arithmetic Operators

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

Example :

```
int a=10;
int b=5;
System.out.println(a+b);//15 ( simple addition )
System.out.println(a-b);//5 ( simple subtraction )
System.out.println(a*b);//50 ( simple multiplication )
System.out.println(a/b);//2 ( simple division )
System.out.println(a%b);//0 ( it gives solution for remainder )
```

Java AND Operator Example: Logical && and Bitwise &

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

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

Example :

```
int a=10;  
int b=5;  
int c=20;  
System.out.println(a<b&&a<c);//false && true = false  
System.out.println(a<b&a<c);//false & true = false
```

Example 2 :

```
int a=10;  
int b=5;  
int c=20;  
System.out.println(a<b&&a++<c);//false && true = false  
System.out.println(a);//10 because second condition is not checked  
System.out.println(a<b&a++<c);//false && true = false  
System.out.println(a);//11 because second condition is checked
```

Java OR Operator Example: Logical || and Bitwise |

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

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

```
int a=10;  
int b=5;  
int c=20;  
System.out.println(a>b||a<c);//true || true = true  
System.out.println(a>b|a<c);//true | true = true  
//|| vs |  
System.out.println(a>b||a++<c);//true || true = true  
System.out.println(a);//10 because second condition is not checked  
System.out.println(a>b|a++<c);//true | true = true  
System.out.println(a);//11 because second condition is checked
```

Java Assignment Operator

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

Java Assignment Operator Example

```
int a=10;  
int b=20;  
a+=4;//a=a+4 (a=10+4)
```

```
b-=4;//b=b-4 (b=20-4)
```

Example 2:

```
int a=10;  
a+=3;//10+3  
a-=4;//13-4  
a*=2;//9*2  
a/=2;//18/2
```

Java Keywords

Java keywords are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class 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 an abstract class. An abstract class can provide the implementation of the 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 the loop or switch statement. It breaks the current flow of the program at specified conditions.
4. **byte**
Java byte keyword is used to declare a variable that can hold 8-bit data values.
5. **case**
Java case keyword is used 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 the 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 number.
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 used with a variable. It is used to restrict the user from updating the value of the variable.
17. finally
 - : Java finally keyword indicates a block of code in a try-catch structure. This block is always executed whether an 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 condition becomes true. If the number of iterations is fixed, it is recommended to use a for loop.
20. if
 - : Java if keyword tests the condition. It executes the if block if the 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 the package and outside the package but through inheritance only. It can't be applied to 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 mainly used for memory management.
- 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 to parent class objects. It can be used to invoke the 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. synchronised
 - : Java synchronised keyword is used to specify the critical sections or methods in multithreaded code.
- 41. this
 - : Java this keyword can be used to refer to 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 exceptions. It is followed by an instance.
- 43. throws
 - : The Java throws keyword is used to declare an exception. Checked exceptions can be propagated with throws.

44. transient

: Java transient keyword is used in serialisation. If you define any data member as transient, it will not be serialised.

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 the while loop.

Java Control Statements

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements

- if statements
- switch statement

2. Loop statements

- do while loop
- while loop
- for loop
- for-each loop

3. Jump statements

- break statement
- continue statement

Decision-Making statements:

As the name suggests, **decision-making statements decide which statement to execute and when**. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

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 statements in Java.

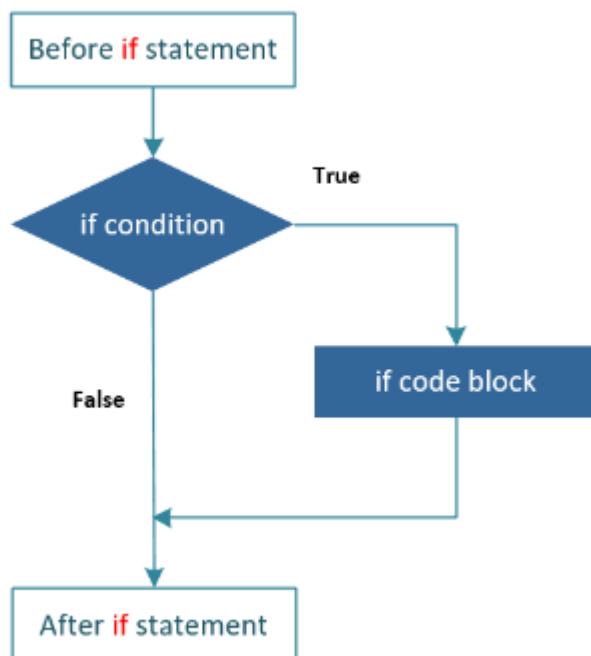
- if statement
- if-else statement
- if-else-if ladder
- nested if statement

Java if Statement

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

Syntax:

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



Example :

```
public class IfExample {
public static void main(String[] args) {
    //defining an 'age' variable
    int age=20;
    //checking the age
    if(age>18){
        System.out.print("Age is greater than 18");
    }
}
}
```

Output :

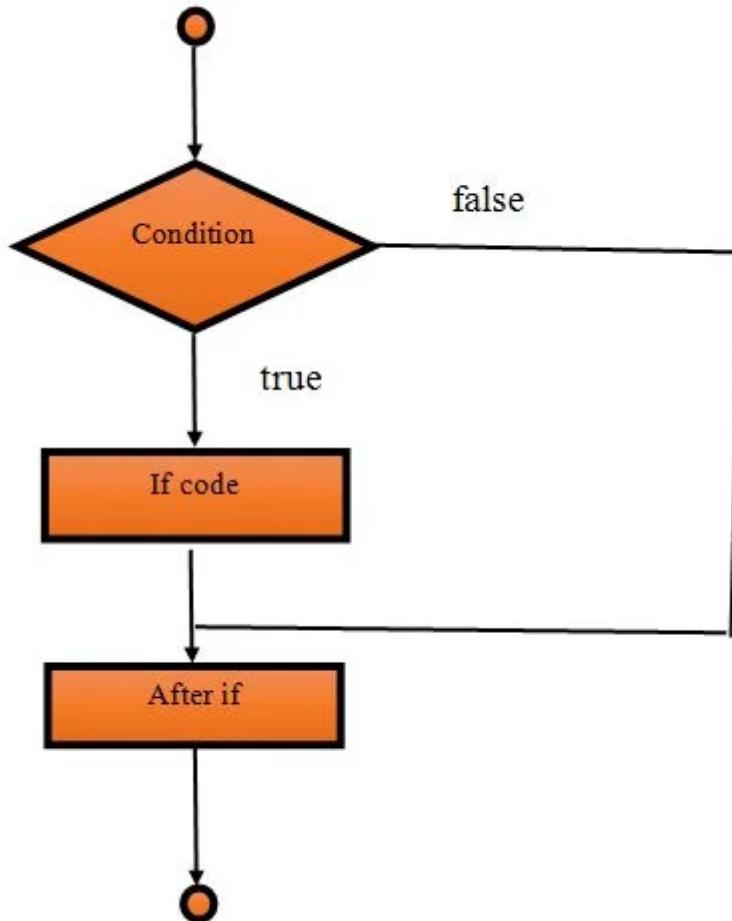
Age is greater than 18

Java if-else Statement

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

Syntax:

```
if(condition){  
//code if condition is true  
}else{  
//code if condition is false  
}
```



Example

```
public class IfElseExample {  
public static void main(String[] args) {  
    //defining a variable  
    int number=13;  
    //Check if the number is divisible by 2 or not  
    if(number%2==0){  
        System.out.println("even number");  
    }else{  
        System.out.println("odd number");  
    }  
}
```

```
}
```

Output :
odd number

Using Ternary Operator

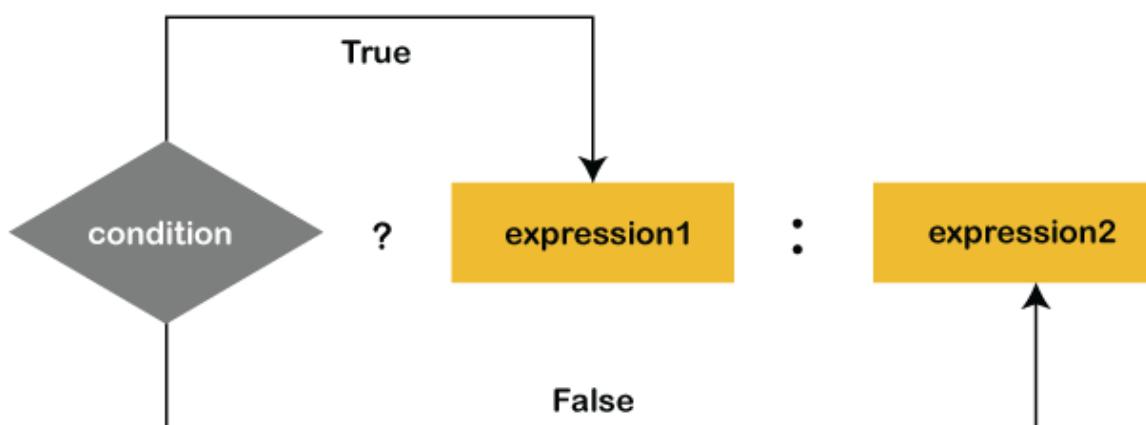
In Java, the ternary operator is a type of Java conditional operator. In this section, we will discuss the ternary operator in Java with proper examples.

The meaning of ternary is composed of three parts. The ternary operator (`? :`) consists of three operands. It is used to evaluate Boolean expressions. The operator decides which value will be assigned to the variable. It is the only conditional operator that accepts three operands. It can be used instead of the if-else statement. It makes the code much more easy, readable, and shorter.

Syntax:

- variable = (condition) ? expression1 : expression2

The above statement states that if the condition returns true, expression1 gets executed, else the expression2 gets executed and the final result stored in a variable.



Example :

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

Output : odd number

This is the same program as we have used in the above example .

Java if-else-if ladder Statement

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

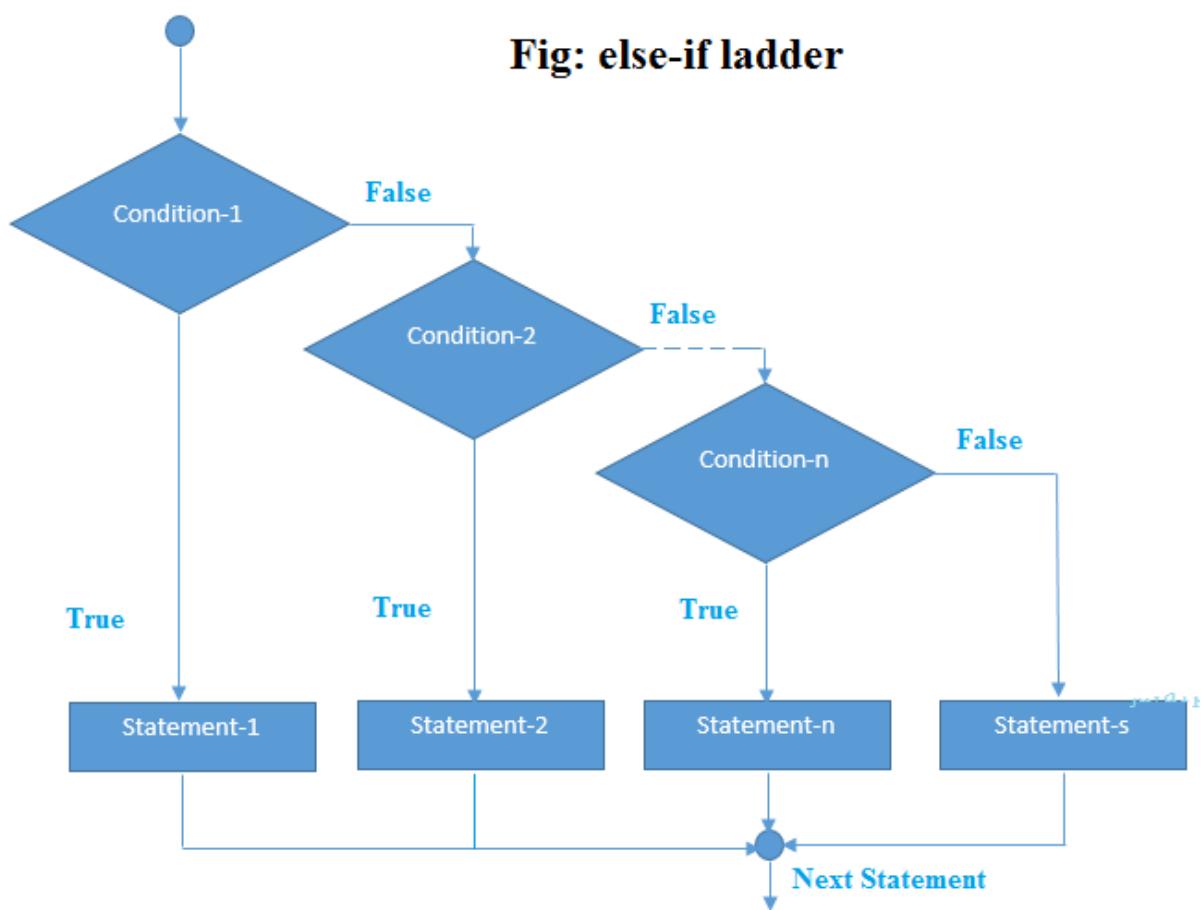
Syntax:

```

if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}

```

Fig: else-if ladder



Example :

```

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

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

```

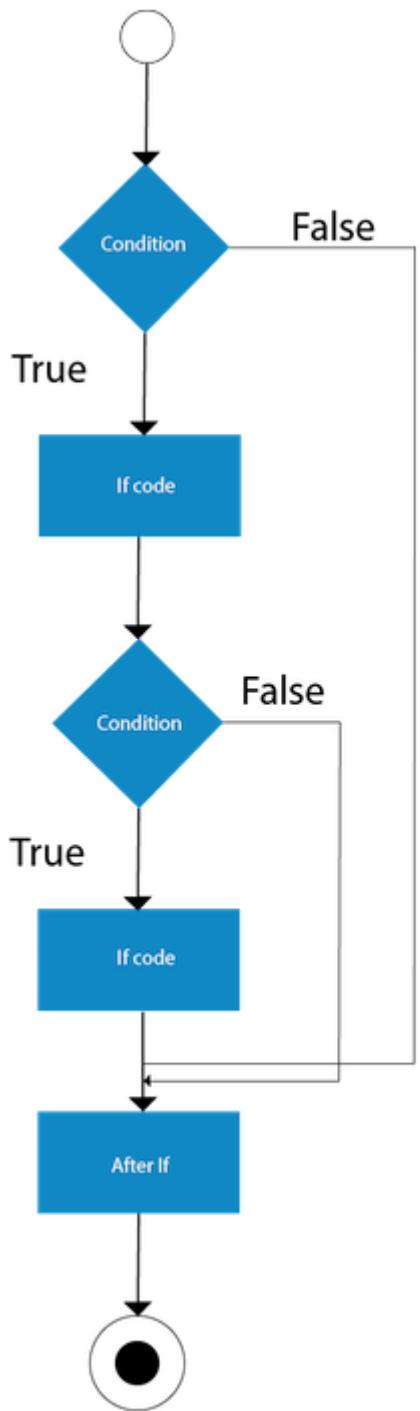
```
        System.out.println("D grade");
    }
else if(marks>=60 && marks<70){
    System.out.println("C grade");
}
else if(marks>=70 && marks<80){
    System.out.println("B grade");
}
else if(marks>=80 && marks<90){
    System.out.println("A grade");
}else if(marks>=90 && marks<100){
    System.out.println("A+ grade");
}else{
    System.out.println("Invalid!");
}
}
}
Output : C grade
```

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 the outer if block condition is true.

Syntax:

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



Java Switch Statement

The Java switch statement executes one statement from multiple conditions. It is like an if-else-if ladder statement. The switch statement works with `byte`, `short`, `int`, `long`, `enum`

types, **String** and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use strings in the switch statement.

In other words, the switch statement tests the equality of a variable against multiple values.

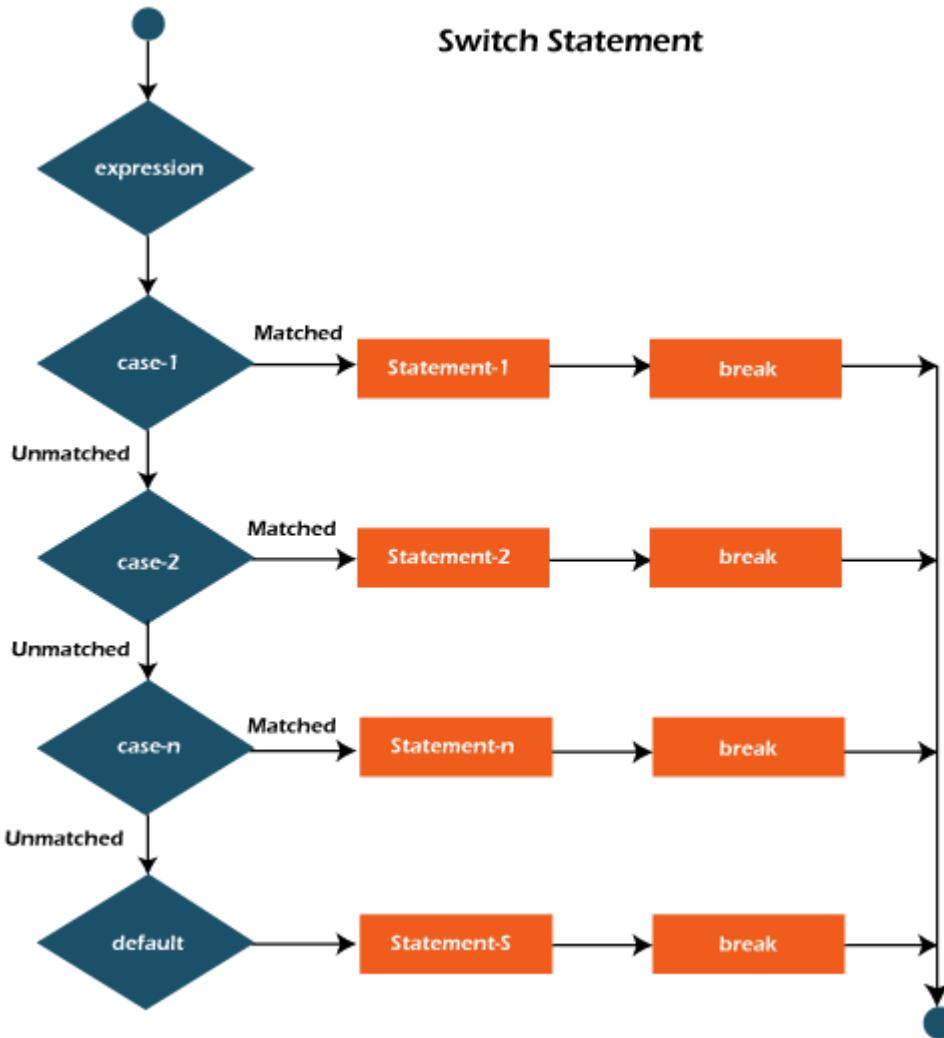
Points to Remember

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

Syntax:

```
switch(expression){  
    case value1:  
        //code to be executed;  
        break; //optional  
    case value2:  
        //code to be executed;  
        break; //optional  
    .....  
  
    default:  
        code to be executed if all cases are not matched;  
}
```

Flowchart of Switch Statement



Example:

```

SwitchExample.java
public class SwitchExample {
public static void main(String[] args) {
    //Declaring a variable for switch expression
    int number=20;
    //Switch expression
    switch(number){
        //Case statements
        case 10: System.out.println("10");
        break;
        case 20: System.out.println("20");
        break;
        case 30: System.out.println("30");
        break;
        //Default case statement
        default:System.out.println("Not in 10, 20 or 30");
    }
}
  
```

```
    }
}
}
```

Java Nested Switch Statement

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

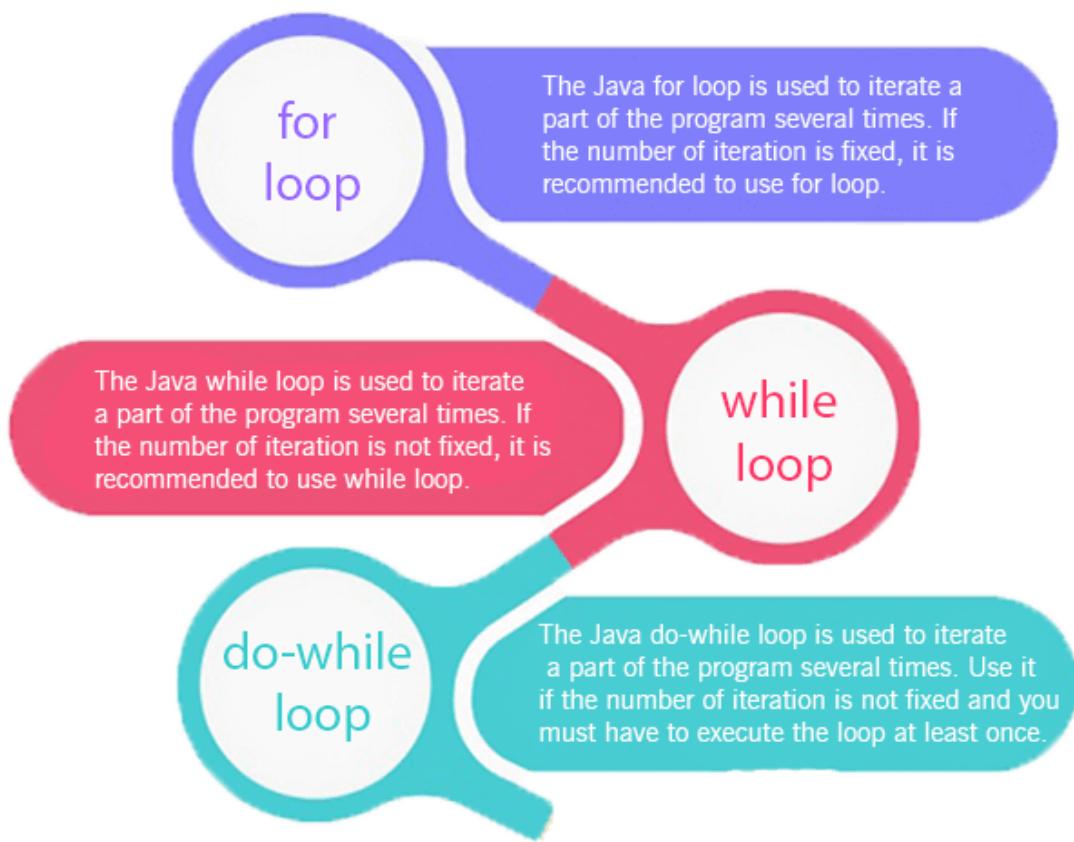
Example:

```
NestedSwitchExample.java
//Java Program to demonstrate the use of Java Nested Switch
public class NestedSwitchExample {
    public static void main(String args[])
    {
        //C - CSE, E - ECE, M - Mechanical
        char branch = 'C';
        int collegeYear = 4;
        switch( collegeYear )
        {
            case 1:
                System.out.println("English, Maths, Science");
                break;
            case 2:
                switch( branch )
                {
                    case 'C':
                        System.out.println("Operating System, Java, Data Structure");
                        break;
                    case 'E':
                        System.out.println("Micro processors, Logic switching theory");
                        break;
                    case 'M':
                        System.out.println("Drawing, Manufacturing Machines");
                        break;
                }
                break;
            case 3:
                switch( branch )
                {
                    case 'C':
                        System.out.println("Computer Organization, MultiMedia");
                        break;
                    case 'E':
                        System.out.println("Fundamentals of Logic Design, Microelectronics");
                        break;
                }
        }
    }
}
```

```
case 'M':
    System.out.println("Internal Combustion Engines, Mechanical Vibration");
    break;
}
break;
case 4:
    switch( branch )
{
    case 'C':
        System.out.println("Data Communication and Networks, MultiMedia");
        break;
    case 'E':
        System.out.println("Embedded System, Image Processing");
        break;
    case 'M':
        System.out.println("Production Technology, Thermal Engineering");
        break;
}
break;
}
}
```

Loops in Java

The Java loop is used to iterate a part of the program several times. There are three types of for loops in Java.



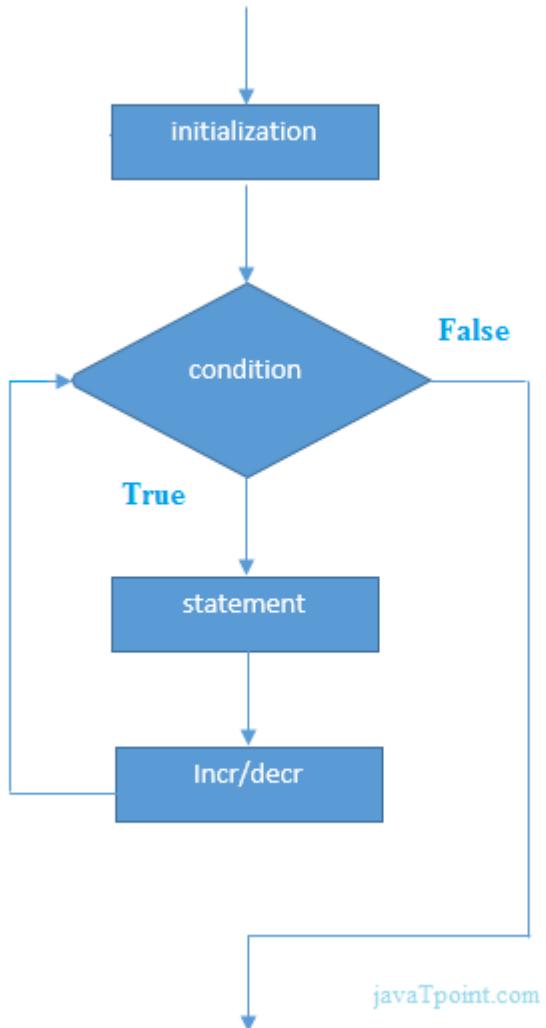
For Loop

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. Increment/Decrement: It increments or decrements the variable value. It is an optional condition.
4. Statement: The statement of the loop is executed each time until the second condition is false.

Syntax:

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



Example:

ForExample.java

```

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

```

Java Nested for Loop

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

Example:

```
NestedForExample.java
public class NestedForExample {
public static void main(String[] args) {
//loop of i
for(int i=1;i<=3;i++){
//loop of j
for(int j=1;j<=3;j++){
    System.out.println(i+" "+j);
}
}
}
}
```

Java Infinitive for Loop

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

Syntax:

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

Example:

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

Java While Loop

The Java while loop is used to iterate a part of the program repeatedly until the specified Boolean condition is true. As soon as the Boolean condition becomes false, the loop automatically stops.

The while loop is considered as a repeating if statement. If the number of iteration is not fixed, it is recommended to use the while loop.

Syntax:

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

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. When the condition becomes false, we exit the while loop.

Example:

`i <=100`

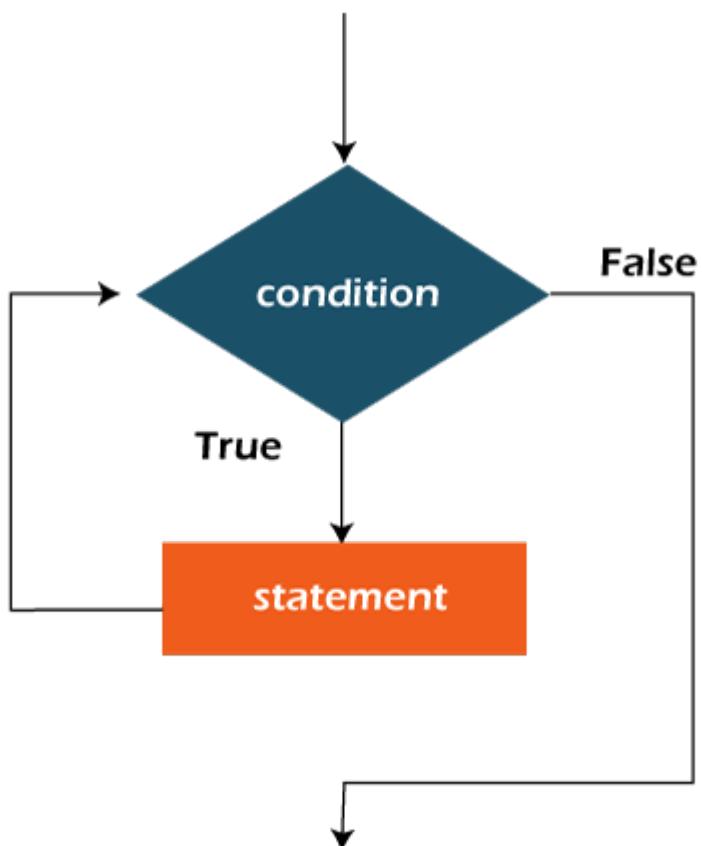
2. Update expression: Every time the loop body is executed, this expression increments or decrements loop variable.

Example:

`i++;`

Flowchart of Java While Loop

Here, the important thing about while loop is that, sometimes it may not even execute. If the condition to be tested results into false, the loop body is skipped and first statement after the while loop will be executed.



Example:

In the below example, we print integer values from 1 to 10. Unlike the for loop, we separately need to initialize and increment the variable used in the condition (here, i). Otherwise, the loop will execute infinitely.

WhileExample.java

```
public class WhileExample {  
    public static void main(String[] args) {  
        int i=1;  
        while(i<=10){
```

```
        System.out.println(i);
        i++;
    }
}
}
```

Java Infinitive While Loop

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

Syntax:

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

Example:

```
WhileExample2.java
public class WhileExample2 {
public static void main(String[] args) {
// setting the infinite while loop by passing true to the condition
while(true){
    System.out.println("infinitive while loop");
}
}
}
```

Java do-while Loop

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

Java do-while loop is called an exit control loop. Therefore, unlike while loop and for loop, the do-while check the condition at the end of loop body. The Java do-while loop is executed at least once because condition is checked after loop body.

Syntax:

```
do{
//code to be executed / loop body
//update statement
}while (condition);
```

The different parts of do-while loop:

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. As soon as the condition becomes false, loop breaks automatically.

Example:

```
i <=100
```

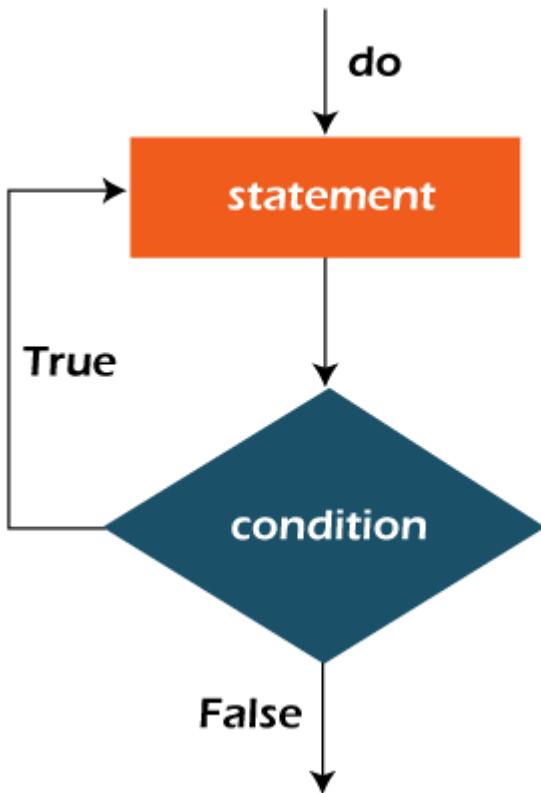
2. Update expression: Every time the loop body is executed, the this expression increments or decrements loop variable.

Example:

```
i++;
```

Note: The do block is executed at least once, even if the condition is false.

Flowchart of do-while loop:



Example:

In the below example, we print integer values from 1 to 10. Unlike the for loop, we separately need to initialize and increment the variable used in the condition (here, i). Otherwise, the loop will execute infinitely.

DoWhileExample.java

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

Java Infinitive do-while Loop

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

Syntax:

```

do{
//code to be executed
}while(true);
Example:
DoWhileExample2.java
public class DoWhileExample2 {
public static void main(String[] args) {
do{
    System.out.println("infinitive do while loop");
}while(true);
}
}

```

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.

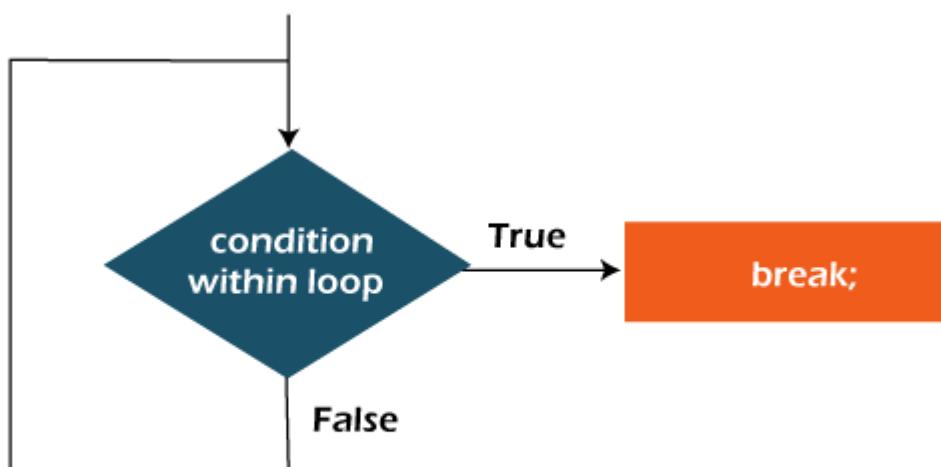
The Java break statement is used to break loop or switch statement. It breaks the current flow of the program at specified conditions. In the case of an inner loop, it breaks only the inner loop.

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

Syntax:

```
jump-statement;
break;
```

Flowchart of Break Statement



Flowchart of break statement

Java Break Statement with Loop

Example:

BreakExample.java

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

Java Break Statement with Inner Loop

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

Example:

BreakExample2.java

```
//Java Program to illustrate the use of break statement  
//inside an inner loop  
public class BreakExample2 {  
    public static void main(String[] args) {  
        //outer loop  
        for(int i=1;i<=3;i++){  
            //inner loop  
            for(int j=1;j<=3;j++){  
                if(i==2&&j==2){  
                    //using break statement inside the inner loop  
                    break;  
                }  
                System.out.println(i+" "+j);  
            }  
        }  
    }  
}
```

Java Break Statement in while loop

Example:

BreakWhileExample.java

```
//Java Program to demonstrate the use of break statement  
//inside the while loop.
```

```

public class BreakWhileExample {
    public static void main(String[] args) {
        //while loop
        int i=1;
        while(i<=10){
            if(i==5){
                //using break statement
                i++;
                break;//it will break the loop
            }
            System.out.println(i);
            i++;
        }
    }
}

```

Java Break Statement in do-while loop

Example:

```

BreakDoWhileExample.java
//Java Program to demonstrate the use of break statement
//inside the Java do-while loop.
public class BreakDoWhileExample {
    public static void main(String[] args) {
        //declaring variable
        int i=1;
        //do-while loop
        do{
            if(i==5){
                //using break statement
                i++;
                break;//it will break the loop
            }
            System.out.println(i);
            i++;
        }while(i<=10);
    }
}

```

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 the case of an inner loop, it continues the inner loop only.

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

Syntax:

```
jump-statement;  
continue;
```

Java Continue Statement Example

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

Java Continue Statement with Inner Loop

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

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

```
}
```

Java Continue Statement in while loop

```
ContinueWhileExample.java
//Java Program to demonstrate the use of continue statement
//inside the while loop.
public class ContinueWhileExample {
public static void main(String[] args) {
    //while loop
    int i=1;
    while(i<=10){
        if(i==5){
            //using continue statement
            i++;
            continue;//it will skip the rest statement
        }
        System.out.println(i);
        i++;
    }
}
```

Java Continue Statement in do-while Loop

```
ContinueDoWhileExample.java
//Java Program to demonstrate the use of continue statement
//inside the Java do-while loop.
public class ContinueDoWhileExample {
public static void main(String[] args) {
    //declaring variable
    int i=1;
    //do-while loop
    do{
        if(i==5){
            //using continue statement
            i++;
            continue;//it will skip the rest statement
        }
        System.out.println(i);
        i++;
    }while(i<=10);
}
```

Java Comments

The [Java](#) comments are the statements in a program that are not executed by the compiler and interpreter.

Why do we use comments in a code?

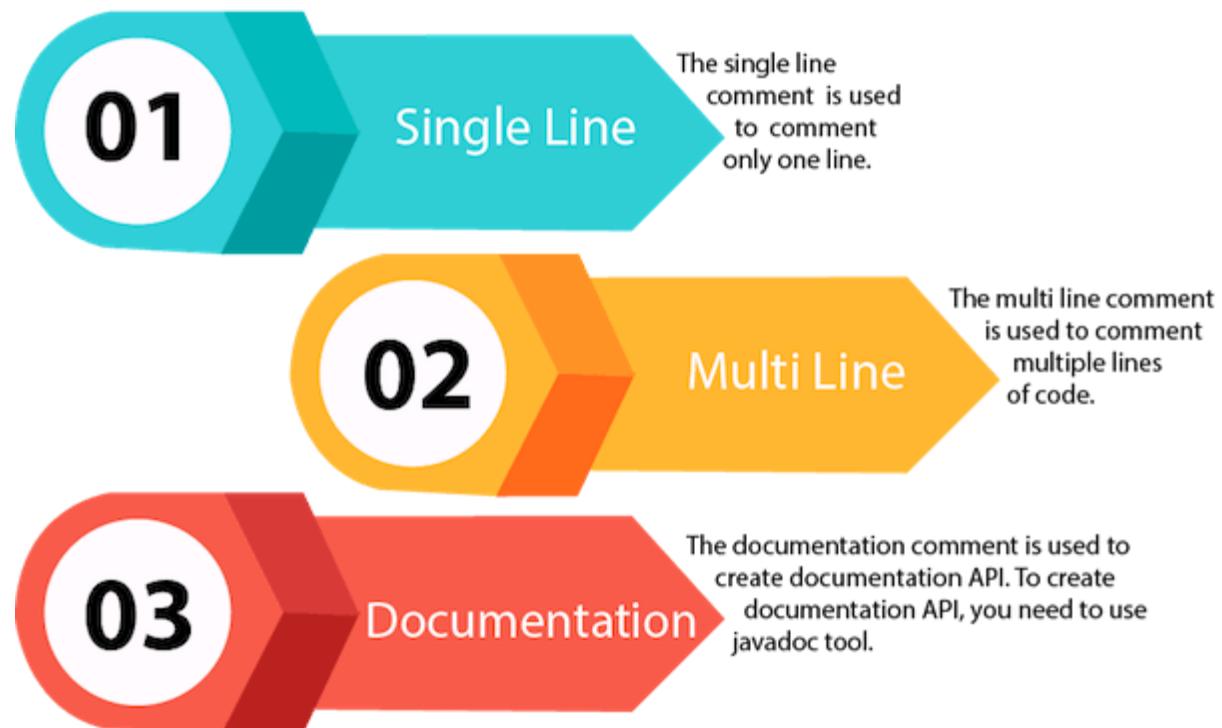
- Comments are used to make the program more readable by adding the details of the code.
- It makes it easy to maintain the code and to find the errors easily.
- The comments can be used to provide information or explanation about the variable, method, class, or any statement.
- It can also be used to prevent the execution of program code while testing the alternative code.

Types of Java Comments

There are three types of comments in Java.

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 of the code. It is the widely used and easiest way of commenting on statements.

Single line comments start with two forward slashes (//). Any text in front of // is not executed by Java.

Syntax:

```
//This is single line comment
```

Let's use single line comment in a Java program.

```
CommentExample1.java
public class CommentExample1 {
    public static void main(String[] args) {
        int i=10; // i is a variable with value 10
        System.out.println(i); //printing the variable i
    }
}
```

2) Java Multi Line Comment

The multi-line comment is used to comment multiple lines of code. It can be used to explain a complex code snippet or to comment multiple lines of code at a time (as it will be difficult to use single-line comments there).

Multi-line comments are placed between /* and */. Any text between /* and */ is not executed by Java.

Syntax:

```
/*
This
is
multi line
comment
*/
```

Note: Usually // is used for short comments and /* */ is used for longer comments.

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.

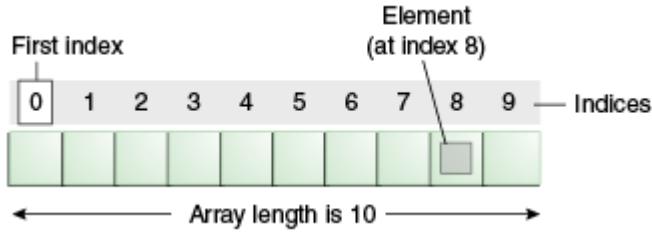
Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

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.

```
//Java Program to illustrate how to declare, instantiate, initialize  
//and traverse the Java array.
```

```
class Testarray{  
public static void main(String args[]){  
int a[]={};//declaration and instantiation
```

```

a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}

```

Declaration, Instantiation and Initialization of Java Array

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

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

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

```

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

```

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.

```
//Java Program to demonstrate the way of passing an anonymous array
//to method.
```

```

public class TestAnonymousArray{
//creating a method which receives an array as a parameter
static void printArray(int arr[]){
for(int i=0;i<arr.length;i++)
System.out.println(arr[i]);
}

```

```

public static void main(String args[]){
printArray(new int[]{10,22,44,66});//passing anonymous array to method
}

```

ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an `ArrayIndexOutOfBoundsException` if length of the array is negative, equal to the array size or greater than the array size while traversing the array.

```

//Java Program to demonstrate the case of
//ArrayIndexOutOfBoundsException in a Java Array.
public class TestArrayException{
public static void main(String args[]){
int arr[]={50,60,70,80};
for(int i=0;i<=arr.length;i++){
System.out.println(arr[i]);
}
}

```

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

```

dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];

```

Example to instantiate Multidimensional Array in Java

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

Example to initialize Multidimensional Array in Java

```

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

```

Example of Multidimensional Java Array

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

```

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

```

```
}}
```

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.

```
//Java Program to illustrate the jagged array  
class TestJaggedArray{
```

```
    public static void main(String[] args){  
        //declaring a 2D array with odd columns  
        int arr[][] = new int[3][];  
        arr[0] = new int[3];  
        arr[1] = new int[4];  
        arr[2] = new int[2];  
        //initializing a jagged array  
        int count = 0;  
        for (int i=0; i<arr.length; i++){  
            for(int j=0; j<arr[i].length; j++)  
                arr[i][j] = count++;  
  
        //printing the data of a jagged array  
        for (int i=0; i<arr.length; i++){  
            for (int j=0; j<arr[i].length; j++){  
                System.out.print(arr[i][j]+" ");  
            }  
            System.out.println();//new line  
        }  
    }  
}
```

What is the class name of the Java array?

In Java, an array is an object. For array object, a proxy class is created whose name can be obtained by `getClass().getName()` method on the object.

```
//Java Program to get the class name of array in Java
```

```
class Testarray4{  
    public static void main(String args[]){  
        //declaration and initialization of array  
        int arr[]={4,4,5};  
        //getting the class name of Java array  
        Class c=arr.getClass();  
        String name=c.getName();  
        //printing the class name of Java array  
        System.out.println(name);  
  
    }}
```

Java OOPs Concepts

Simula is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

Smalltalk is considered the first truly object-oriented programming language.

The popular object-oriented languages are [Java](#), [C#](#), [PHP](#), [Python](#), [C++](#), etc.

The main aim of object-oriented programming is to implement real-world entities, for example, object, classes, abstraction, inheritance, polymorphism, etc.

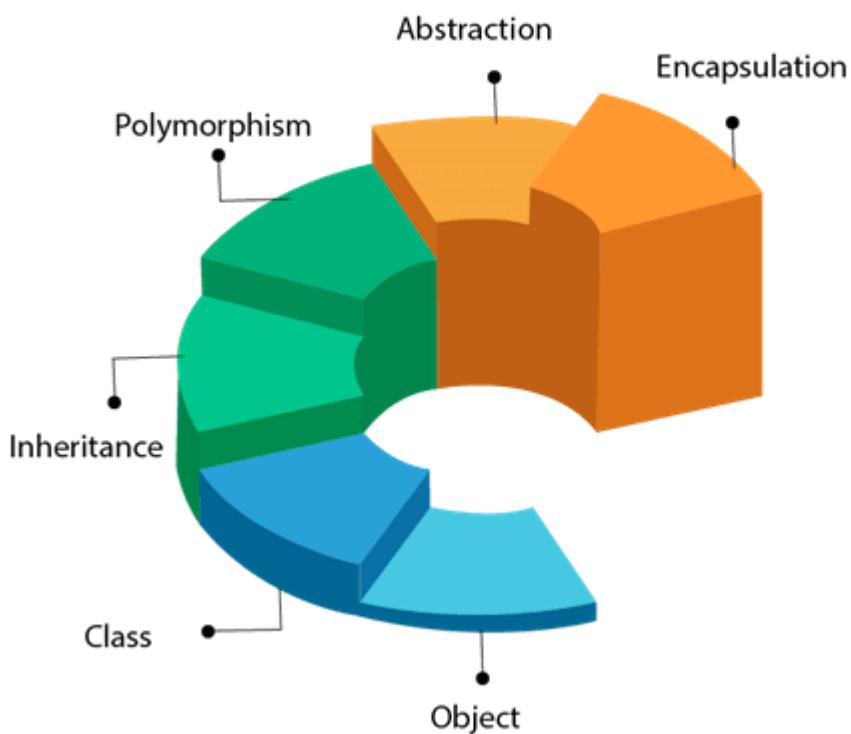
OOPs (Object-Oriented Programming System)

Object means a real-world entity such as a pen, chair, table, computer, watch, etc.

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

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

OOPs (Object-Oriented Programming System)



Object



Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

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

Inheritance

When one object acquires all the properties and behaviours 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.

Java Naming Convention

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.

But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

Advantage of Naming Conventions in Java

By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

Naming Conventions of the Different Identifiers

The following table shows the popular conventions used for the different identifiers.

Identifiers Type	Naming Rules	Examples
Class	<p>It should start with the uppercase letter.</p> <p>It should be a noun such as Color, Button, System, Thread, etc.</p> <p>Use appropriate words, instead of acronyms.</p>	<pre>public class Employee { //code snippet }</pre>
Interface	<p>It should start with the uppercase letter.</p> <p>It should be an adjective such as Runnable, Remote, ActionListener.</p> <p>Use appropriate words, instead of acronyms.</p>	<pre>interface Printable { //code snippet }</pre>
Method	<p>It should start with lowercase letter.</p> <p>It should be a verb such as main(), print(), println().</p> <p>If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().</p>	<pre>class Employee { // method void draw() { //code snippet } }</pre>

Variable	<p>It should start with a lowercase letter such as id, name.</p> <p>It should not start with the special characters like & (ampersand), \$ (dollar), _ (underscore).</p> <p>If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.</p> <p>Avoid using one-character variables such as x, y, z.</p>	<pre>class Employee { // variable int id; //code snippet }</pre>
Package	<p>It should be a lowercase letter such as java, lang.</p> <p>If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.</p>	<pre>//package package com.javatpoint; class Employee { //code snippet }</pre>
Constant	<p>It should be in uppercase letters such as RED, YELLOW.</p> <p>If the name contains multiple words, it should be separated by an underscore(_) such as MAX_PRIORITY.</p> <p>It may contain digits but not as the first letter.</p>	<pre>class Employee { //constant static final int MIN_AGE = 18; //code snippet }</pre>

CamelCase in Java naming conventions

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

Objects and Classes in Java

In this page, we will learn about Java objects and classes. In object-oriented programming technique, we design a program using objects and classes.

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

What is an object in Java

Objects: Real World Examples

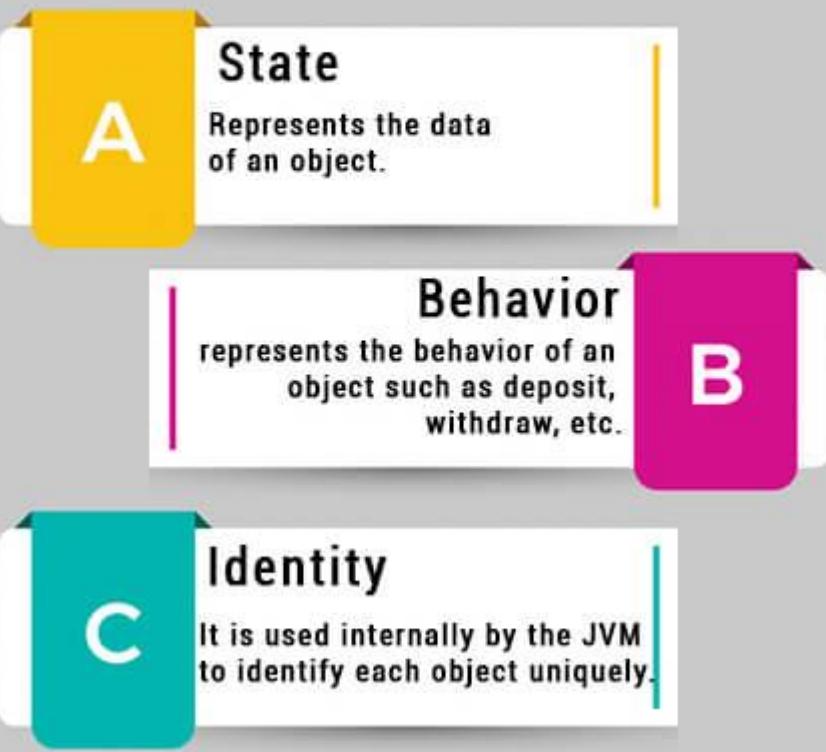


An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

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

Characteristics of Object



For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

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

AD

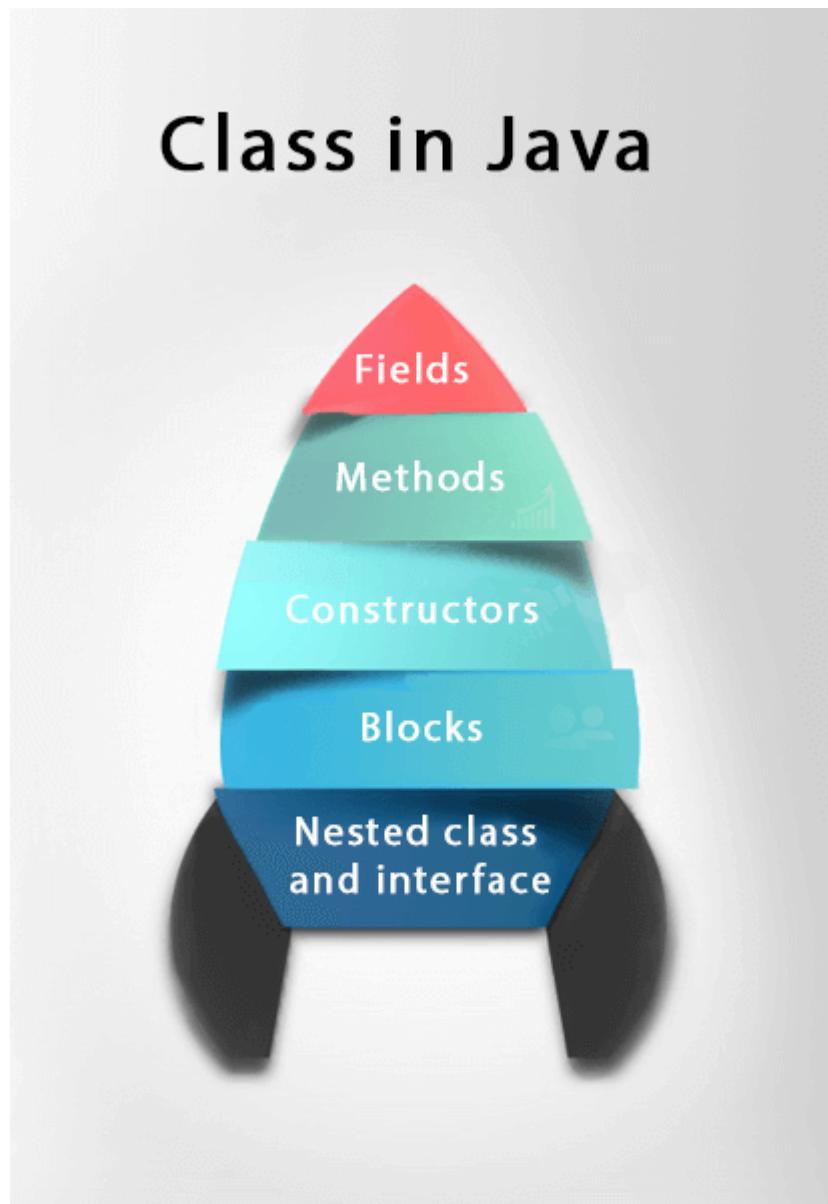
What is a class in Java

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

A class in Java can contain:

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

Class in Java



Syntax to declare a class:

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

Instance variable in Java

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

Method in Java

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

Advantage of Method

- Code Reusability
 - Code Optimization
-

new keyword in Java

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

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
    //defining fields
    int id;//field or data member or instance variable
    String name;
    //creating main method inside the Student class
    public static void main(String args[]){
        //Creating an object or instance
        Student s1=new Student();//creating an object of Student
        //Printing values of the object
        System.out.println(s1.id);//accessing member through reference variable
        System.out.println(s1.name);
    }
}
```

Object and Class Example: main outside the class

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

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

File: TestStudent1.java

```
//Java Program to demonstrate having the main method in
//another class
//Creating Student class.
```

```

class Student{
    int id;
    String name;
}
//Creating another class TestStudent1 which contains the main method
class TestStudent1{
    public static void main(String args[]){
        Student s1=new Student();
        System.out.println(s1.id);
        System.out.println(s1.name);
    }
}

```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference

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

AD

File: TestStudent2.java

```

class Student{
    int id;
    String name;
}
class TestStudent2{
    public static void main(String args[]){
        Student s1=new Student();
        s1.id=101;
        s1.name="Sonoo";
        System.out.println(s1.id+" "+s1.name);//printing members with a white space
    }
}

```

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

File: TestStudent3.java

```

class Student{
    int id;
    String name;
}
class TestStudent3{
    public static void main(String args[]){
        //Creating objects
        Student s1=new Student();

```

```

Student s2=new Student();
//Initializing objects
s1.id=101;
s1.name="Sonoo";
s2.id=102;
s2.name="Amit";
//Printing data
System.out.println(s1.id+" "+s1.name);
System.out.println(s2.id+" "+s2.name);
}
}

```

2) Object and Class Example: Initialization through method

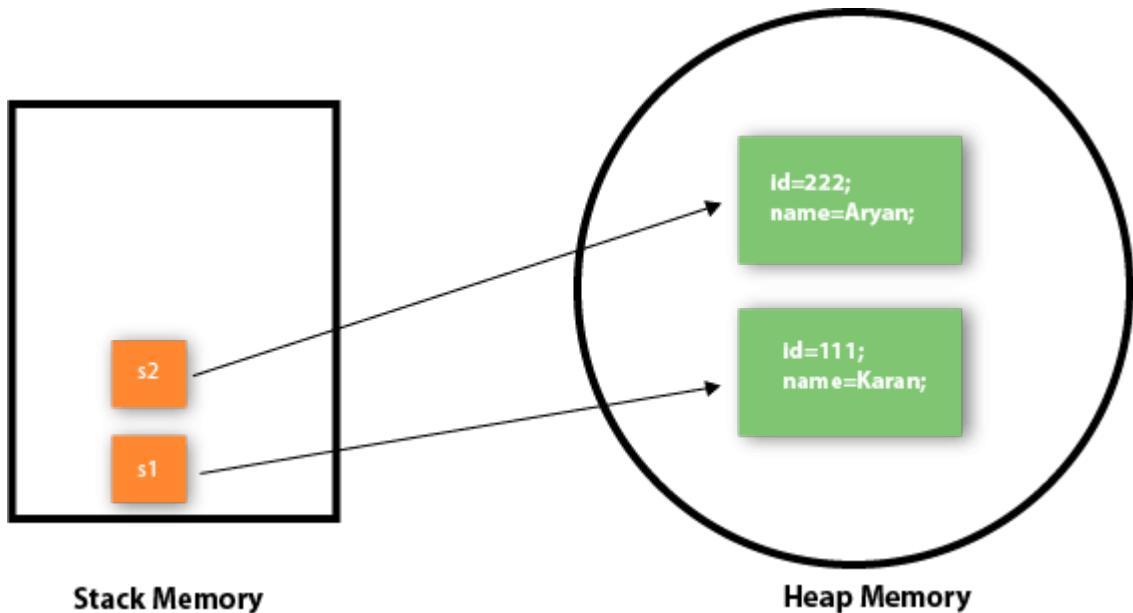
In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

File: TestStudent4.java

```

class Student{
int rollno;
String name;
void insertRecord(int r, String n){
rollno=r;
name=n;
}
void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
public static void main(String args[]){
Student s1=new Student();
Student s2=new Student();
s1.insertRecord(111,"Karan");
s2.insertRecord(222,"Aryan");
s1.displayInformation();
s2.displayInformation();
}
}

```



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

3) Object and Class Example: Initialization through a constructor

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

File: TestEmployee.java

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

```
e2.display();
e3.display();
}
}
```

Object and Class Example: Rectangle

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

File: TestRectangle1.java

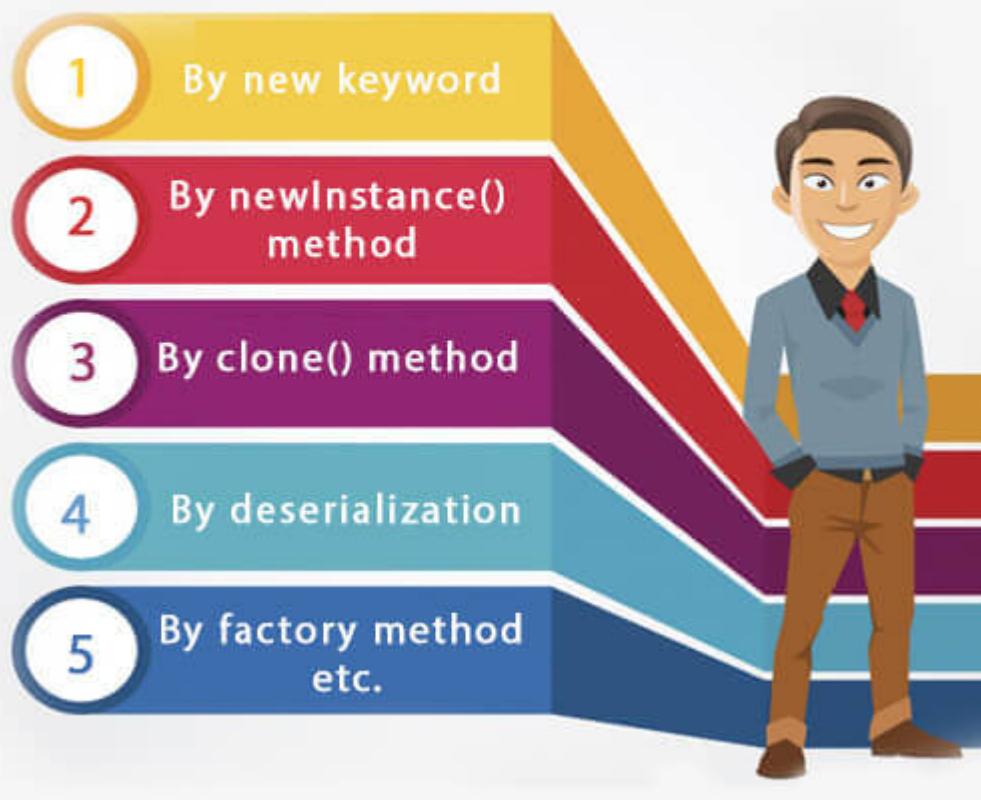
```
class Rectangle{
    int length;
    int width;
    void insert(int l, int w){
        length=l;
        width=w;
    }
    void calculateArea(){System.out.println(length*width);}
}
class TestRectangle1{
    public static void main(String args[]){
        Rectangle r1=new Rectangle();
        Rectangle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

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

Different ways to create an object in Java



Anonymous object

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

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

1. new Calculation();//anonymous object

Real World Example: Account

File: TestAccount.java

```
//Java Program to demonstrate the working of a banking-system  
//where we deposit and withdraw amount from our account.  
//Creating an Account class which has deposit() and withdraw() methods  
class Account{  
    int acc_no;  
    String name;
```

```

float amount;
//Method to initialize object
void insert(int a,String n,float amt){
acc_no=a;
name=n;
amount=amt;
}
//deposit method
void deposit(float amt){
amount=amount+amt;
System.out.println(amt+" deposited");
}
//withdraw method
void withdraw(float amt){
if(amount<amt){
System.out.println("Insufficient Balance");
}else{
amount=amount-amt;
System.out.println(amt+" withdrawn");
}
}
//method to check the balance of the account
void checkBalance(){System.out.println("Balance is: "+amount);}
//method to display the values of an object
void display(){System.out.println(acc_no+" "+name+" "+amount);}
}
//Creating a test class to deposit and withdraw amount
class TestAccount{
public static void main(String[] args){
Account a1=new Account();
a1.insert(832345,"Ankit",1000);
a1.display();
a1.checkBalance();
a1.deposit(40000);
a1.checkBalance();
a1.withdraw(15000);
a1.checkBalance();
}}

```

Method in Java

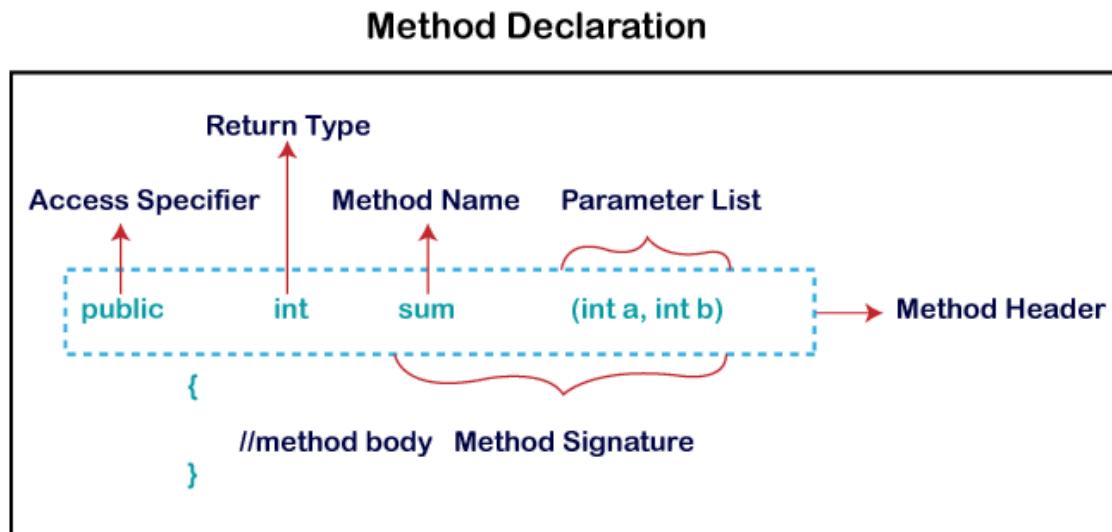
In general, a method is a way to perform some task. Similarly, the method in Java is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using methods. In this section, we will learn what is a method in Java, types of methods, method declaration, and how to call a method in Java.

What is a method in Java?

A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the reusability of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the easy modification and readability of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as method header, as we have shown in the following figure.



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the method name and parameter list.

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides four types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be subtraction(). A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming a Method

While defining a method, remember that the method name must be a verb and start with a lowercase letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in uppercase except the first word. For example:

Single-word method name: sum(), area()

Multi-word method name: areaOfCircle(), stringComparision()

It is also possible that a method has the same name as another method name in the same class, it is known as method overloading.

Types of Method

There are two types of methods in Java:

- Predefined Method
- User-defined Method

Predefined Method

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the standard library method or built-in method. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are length(), equals(), compareTo(), sqrt(), etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

Each and every predefined method is defined inside a class. Such as print() method is defined in the java.io.PrintStream class. It prints the statement that we write inside the method. For example, print("Java"), it prints Java on the console.

Let's see an example of the predefined method.

Demo.java

```
public class Demo
{
    public static void main(String[] args)
    {
        // using the max() method of Math class
        System.out.print("The maximum number is: " + Math.max(9,7));
    }
}
```

In the above example, we have used three predefined methods main(), print(), and max(). We have used these methods directly without declaration because they are predefined. The

`print()` method is a method of `PrintStream` class that prints the result on the console. The `max()` method is a method of the `Math` class that returns the greater of two numbers.

```
max

public static int max(int a,
                      int b)

    Returns the greater of two int values.
    same value.

    Parameters:
        a - an argument.
        b - another argument.

    Returns:
        the larger of a and b.
```

In the above method signature, we see that the method signature has access specifier `public`, non-access modifier `static`, return type `int`, method name `max()`, parameter list (`int a, int b`). In the above example, instead of defining the method, we have just invoked the method. This is the advantage of a predefined method. It makes programming less complicated.

Similarly, we can also see the method signature of the `print()` method.

User-defined Method

The method written by the user or programmer is known as a user-defined method. These methods are modified according to the requirement.

How to Create a User-defined Method

Let's create a user defined method that checks the number is even or odd. First, we will define the method.

```
/user defined method
public static void findEvenOdd(int num)
{
    //method body
    if(num%2==0)
        System.out.println(num+" is even");
    else
        System.out.println(num+" is odd");
}
```

We have defined the above method named `findevenodd()`. It has a parameter `num` of type `int`. The method does not return any value that's why we have used `void`. The method body contains the steps to check the number is even or odd. If the number is even, it prints the number is even, else prints the number is odd.

How to Call or Invoke a User-defined Method

Once we have defined a method, it should be called. The calling of a method in a program is simple. When we call or invoke a user-defined method, the program control transfer to the called method.

```
import java.util.Scanner;
public class EvenOdd
{
    public static void main (String args[])
    {
        //creating Scanner class object
        Scanner scan=new Scanner(System.in);
        System.out.print("Enter the number: ");
        //reading value from the user
        int num=scan.nextInt();
        //method calling
        findEvenOdd(num);
    }
}
```

In the above code snippet, as soon as the compiler reaches at line `findEvenOdd(num)`, the control transfer to the method and gives the output accordingly.

Let's combine both snippets of codes in a single program and execute it.

```
EvenOdd.java
import java.util.Scanner;
public class EvenOdd
{
    public static void main (String args[])
    {
        //creating Scanner class object
        Scanner scan=new Scanner(System.in);
        System.out.print("Enter the number: ");
        //reading value from user
        int num=scan.nextInt();
        //method calling
        findEvenOdd(num);
    }
    //user defined method
    public static void findEvenOdd(int num)
    {
        //method body
        if(num%2==0)
            System.out.println(num+" is even");
        else
            System.out.println(num+" is odd");
    }
}
```

Let's see another program that return a value to the calling method.

In the following program, we have defined a method named `add()` that sum up the two numbers. It has two parameters `n1` and `n2` of integer type. The values of `n1` and `n2`

correspond to the value of a and b, respectively. Therefore, the method adds the value of a and b and store it in the variable s and returns the sum.

Addition.java

```
public class Addition
{
    public static void main(String[] args)
    {
        int a = 19;
        int b = 5;
        //method calling
        int c = add(a, b); //a and b are actual parameters
        System.out.println("The sum of a and b is= " + c);
    }
    //user defined method
    public static int add(int n1, int n2) //n1 and n2 are formal parameters
    {
        int s;
        s=n1+n2;
        return s; //returning the sum
    }
}
```

Static Method

A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword static before the method name.

The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it. It is used to create an instance method. It is invoked by using the class name. The best example of a static method is the main() method.

Example of static method

Display.java

```
public class Display
{
    public static void main(String[] args)
    {
        show();
    }
    static void show()
    {
        System.out.println("It is an example of static method.");
    }
}
```

Instance Method

The method of the class is known as an instance method. It is a non-static method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. Let's see an example of an instance method.

InstanceMethodExample.java

```
public class InstanceMethodExample
{
    public static void main(String [] args)
    {
        //Creating an object of the class
        InstanceMethodExample obj = new InstanceMethodExample();
        //invoking instance method
        System.out.println("The sum is: "+obj.add(12, 13));
    }
    int s;
    //user-defined method because we have not used static keyword
    public int add(int a, int b)
    {
        s = a+b;
        //returning the sum
        return s;
    }
}
```

There are two types of instance method:

- Accessor Method
- Mutator Method

Accessor Method: The method(s) that reads the instance variable(s) is known as the accessor method. We can easily identify it because the method is prefixed with the word get. It is also known as getters. It returns the value of the private field. It is used to get the value of the private field.

Example

1. public int getId()
2. {
3. return Id;
4. }

Mutator Method: The method(s) read the instance variable(s) and also modify the values. We can easily identify it because the method is prefixed with the word set. It is also known as setters or modifiers. It does not return anything. It accepts a parameter of the same data type that depends on the field. It is used to set the value of the private field.

Example

1. public void setRoll(int roll)
2. {
3. this.roll = roll;
4. }

Example of accessor and mutator method

```
Student.java
public class Student
{
    private int roll;
    private String name;
    public int getRoll() //accessor method
    {
        return roll;
    }
    public void setRoll(int roll) //mutator method
    {
        this.roll = roll;
    }
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
    public void display()
    {
        System.out.println("Roll no.: "+roll);
        System.out.println("Student name: "+name);
    }
}
```

Abstract Method

The method that does not have method body is known as abstract method. In other words, without an implementation is known as abstract method. It always declares in the abstract class. It means the class itself must be abstract if it has abstract method. To create an abstract method, we use the keyword `abstract`.

Syntax

1. `abstract void method_name();`

Example of abstract method

```
Demo.java
abstract class Demo //abstract class
{
    //abstract method declaration
    abstract void display();
}
public class MyClass extends Demo
{
```

```

//method implementation
void display()
{
System.out.println("Abstract method?");
}
public static void main(String args[])
{
//creating object of abstract class
Demo obj = new MyClass();
//invoking abstract method
obj.display();
}
}

```

Constructors in Java

In [Java](#), a constructor is a block of codes similar to the method. It is called when an instance of the [class](#) is created. At the time of calling constructor, memory for the object is allocated in the memory.

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

Every time an object is created using the `new()` keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

There are two rules defined for the constructor.

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

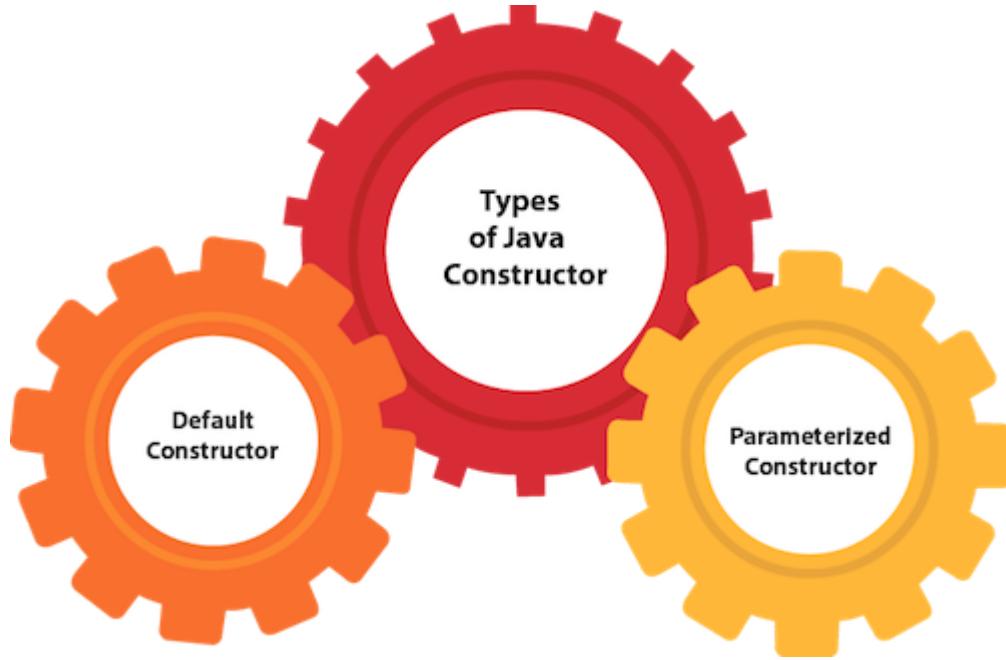
Note: We can use `private` while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)

2. Parameterized constructor



Java Default Constructor

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

Syntax of default constructor:

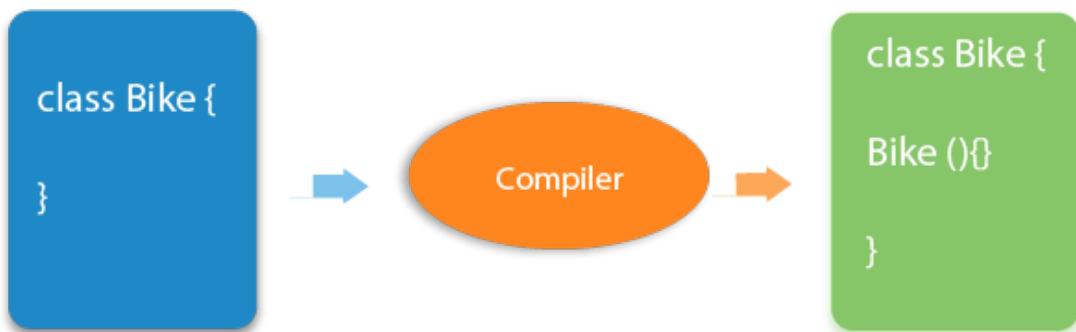
1. <class_name>()

Example of default constructor

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

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

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



Q) What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example of default constructor that displays the default values

```
//Let us see another example of default constructor  
//which displays the default values  
class Student3{  
    int id;  
    String name;  
    //method to display the value of id and name  
    void display(){System.out.println(id+" "+name);}  
  
    public static void main(String args[]){  
        //creating objects  
        Student3 s1=new Student3();  
        Student3 s2=new Student3();  
        //displaying values of the object  
        s1.display();  
        s2.display();  
    }  
}
```

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

Java Parameterized Constructor

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

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

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

//Java Program to demonstrate the use of the parameterized constructor.

```
class Student4{  
    int id;  
    String name;  
    //creating a parameterized constructor  
    Student4(int i, String n){  
        id = i;  
        name = n;  
    }  
    //method to display the values  
    void display(){System.out.println(id+" "+name);}  
  
    public static void main(String args[]){  
        //creating objects and passing values  
        Student4 s1 = new Student4(111, "Karan");  
        Student4 s2 = new Student4(222, "Aryan");  
        //calling method to display the values of object  
        s1.display();  
        s2.display();  
    }  
}
```

Constructor Overloading in Java

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

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

Example of Constructor Overloading

//Java program to overload constructors

```
class Student5{
```

```

int id;
String name;
int age;
//creating two arg constructor
Student5(int i,String n){
id = i;
name = n;
}
//creating three arg constructor
Student5(int i,String n,int a){
id = i;
name = n;
age=a;
}
void display(){System.out.println(id+" "+name+" "+age);}

public static void main(String args[]){
Student5 s1 = new Student5(111,"Karan");
Student5 s2 = new Student5(222,"Aryan",25);
s1.display();
s2.display();
}
}

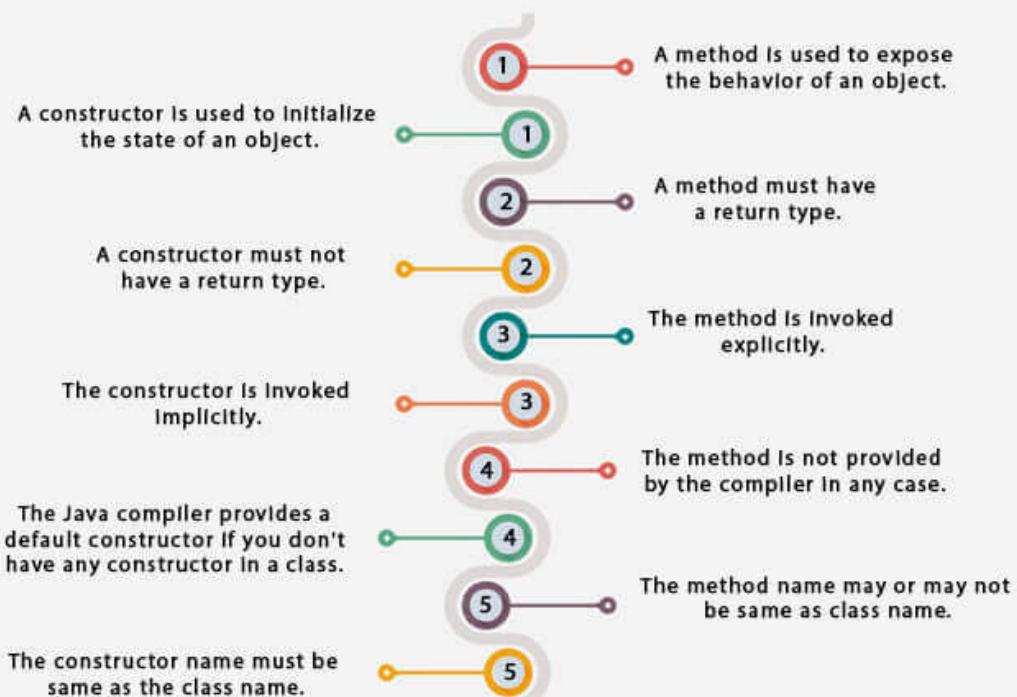
```

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.

Difference between constructor and method in Java



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.

/Java program to initialize the values from one object to another object.

```
class Student6{  
    int id;  
    String name;  
    //constructor to initialize integer and string  
    Student6(int i,String n){  
        id = i;  
        name = n;  
    }  
    //constructor to initialize another object  
    Student6(Student6 s){
```

```

id = s.id;
name = s.name;
}
void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
Student6 s1 = new Student6(111,"Karan");
Student6 s2 = new Student6(s1);
s1.display();
s2.display();
}
}

```

Copying values without constructor

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

```

class Student7{
int id;
String name;
Student7(int i,String n){
id = i;
name = n;
}
Student7(){}
void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
Student7 s1 = new Student7(111,"Karan");
Student7 s2 = new Student7();
s2.id=s1.id;
s2.name=s1.name;
s1.display();
s2.display();
}
}

```

Q) Does constructor return any value?

Yes, it is the current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

Is there Constructor class in Java?

Yes.

What is the purpose of Constructor class?

Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the `java.lang.reflect` package.

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.

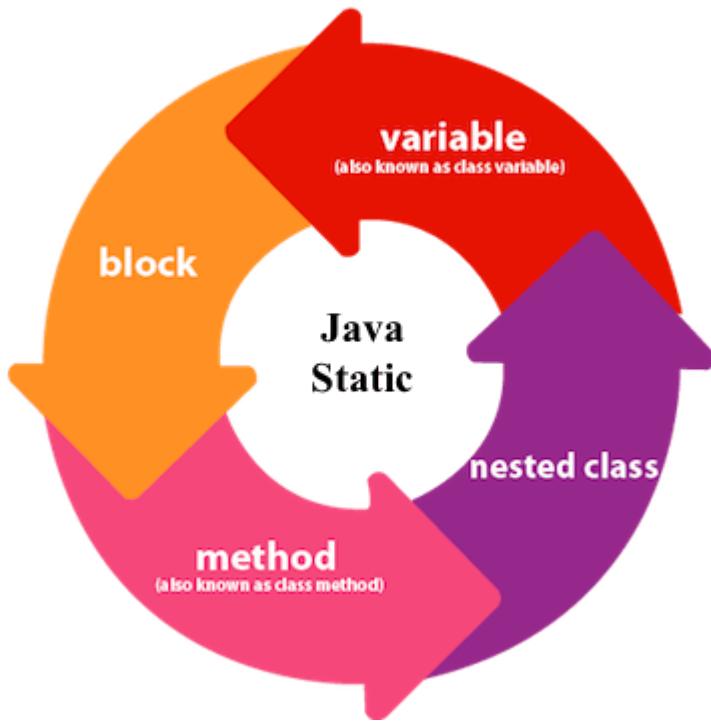
The static can be:

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.

Note: Java static property is shared to all objects.

Example of static variable

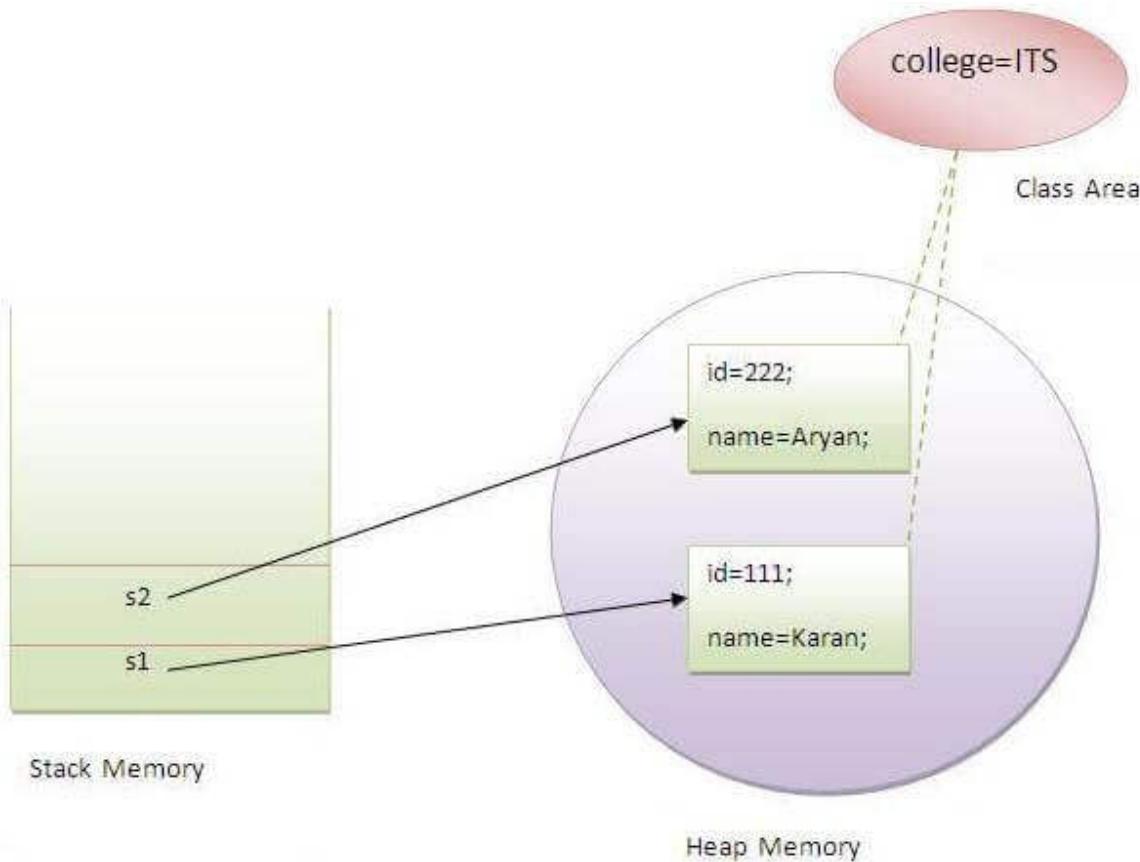
```

//Java Program to demonstrate the use of static variable
class Student{
    int rollno;//instance variable
    String name;
    static String college ="ITS";//static variable
    //constructor
    Student(int r, String n){
```

```

rollno = r;
name = n;
}
//method to display the values
void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
    public static void main(String args[]){
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        //we can change the college of all objects by the single line of code
        //Student.college="BBDIT";
        s1.display();
        s2.display();
    }
}

```



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.

```
//Java Program to demonstrate the use of an instance variable
```

```

//which get memory each time when we create an object of the class.
class Counter{
int count=0;//will get memory each time when the instance is created

Counter(){
count++;//incrementing value
System.out.println(count);
}

public static void main(String args[]){
//Creating objects
Counter c1=new Counter();
Counter c2=new Counter();
Counter c3=new Counter();
}
}

```

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.

```

//Java Program to illustrate the use of static variable which
//is shared with all objects.
class Counter2{
static int count=0;//will get memory only once and retain its value

```

```

Counter2(){
count++;//incrementing the value of static variable
System.out.println(count);
}

```

```

public static void main(String args[]){
//creating objects
Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();
}
}

```

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

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

Another example of a static method that performs a normal calculation

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

```
class Calculate{  
    static int cube(int x){  
        return x*x*x;  
    }  
  
    public static void main(String args[]){
```

```
int result=Calculate.cube(5);
System.out.println(result);
}
}
Output:125
```

Restrictions for the static method

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

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

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

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

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 to 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. }
```

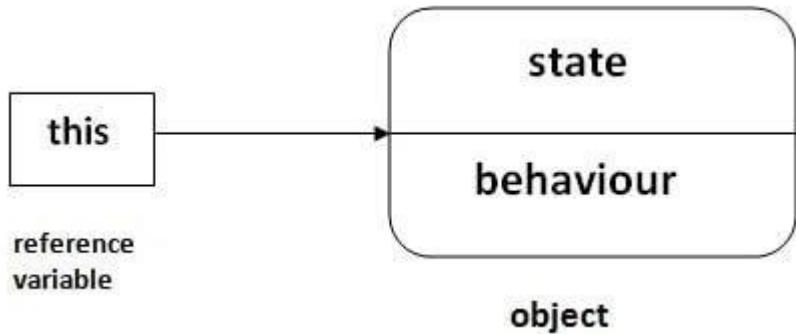
Q) Can we execute a program without main() method?

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

```
1. class A3{
2.     static{
3.         System.out.println("static block is invoked");
4.         System.exit(0);
5.     }
6. }
```

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

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

01

this can be used to refer current class instance variable.

02

this can be used to invoke current class method (implicity)

03

this() can be used to invoke current class Constructor.

04

this can be passed as an argument in the method call.

05

this can be passed as argument in the constructor call.

06

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:

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

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

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

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

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

Program where this keyword is not required

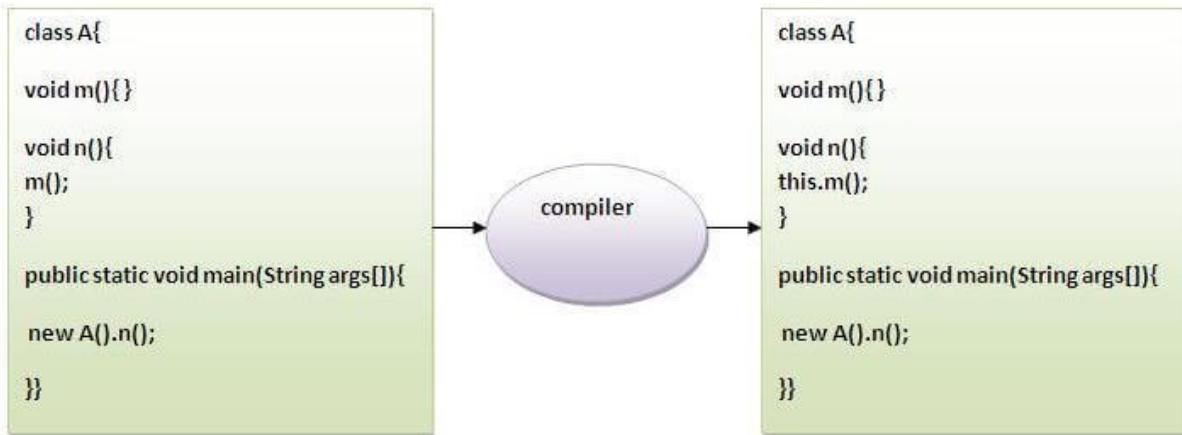
```
class Student{  
    int rollno;  
    String name;  
    float fee;  
    Student(int r, String n, float f){  
        rollno=r;  
        name=n;  
        fee=f;  
    }  
    void display(){System.out.println(rollno+" "+name+" "+fee);}  
}
```

```
class TestThis3{  
    public static void main(String args[]){  
        Student s1=new Student(111,"ankit",5000f);  
        Student s2=new Student(112,"sumit",6000f);  
        s1.display();  
        s2.display();  
    }  
}
```

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

2) this: to invoke current class method

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



```
class A{  
    void m(){System.out.println("hello m");}  
    void n(){  
        System.out.println("hello n");  
        //m();//same as this.m()  
        this.m();  
    }  
}  
class TestThis4{  
    public static void main(String args[]){  
        A a=new A();  
        a.n();  
    }  
}
```

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:

```
class A{  
    A(){System.out.println("hello a");}  
    A(int x){  
        this();  
        System.out.println(x);  
    }  
}  
class TestThis5{
```

```

public static void main(String args[]){
A a=new A(10);
}}
Calling parameterized constructor from default constructor:
class A{
A(){
this(5);
System.out.println("hello a");
}
A(int x){
System.out.println(x);
}
}
class TestThis6{
public static void main(String args[]){
A a=new 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.

```

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

```

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

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.

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

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.

```
class A{  
    private int data=40;  
    private void msg(){System.out.println("Hello java");}  
}  
  
public class Simple{  
    public static void main(String args[]){  
        A obj=new A();  
        System.out.println(obj.data);//Compile Time Error  
        obj.msg();//Compile Time Error  
    }  
}
```

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:

```
class A{  
    private A(){}//private constructor  
    void msg(){System.out.println("Hello java");}  
}  
  
public class Simple{  
    public static void main(String args[]){  
        A obj=new A();//Compile Time Error  
    }  
}
```

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.

```
//save by A.java  
package pack;  
class A{  
    void msg(){System.out.println("Hello");}  
}
```

```
//save by B.java
package mypack;
import pack.*;
class B{
    public static void main(String args[]){
        A obj = new A();//Compile Time Error
        obj.msg();//Compile Time Error
    }
}
```

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 modifer.

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.

```
//save by A.java
package pack;
public class A{
    protected void msg(){System.out.println("Hello");}
}

//save by B.java
package mypack;
import pack.*;

class B extends A{
    public static void main(String args[]){
        B obj = new B();
        obj.msg();
    }
}
```

4) Public

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

Example of public access modifier

//save by A.java

```
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
```

//save by B.java

```
package mypack;
import pack.*;

class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

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.

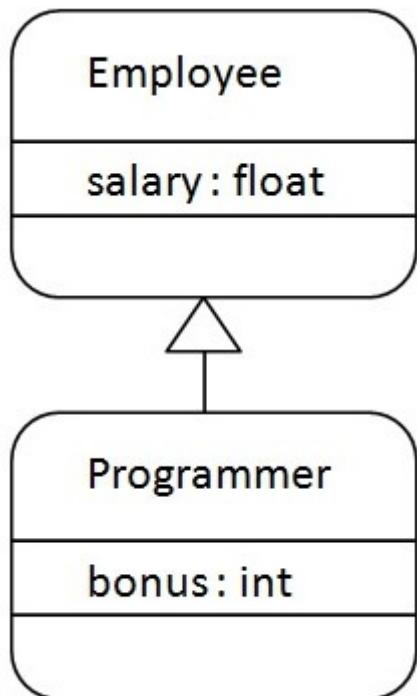
- Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- Reusability: 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.



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.

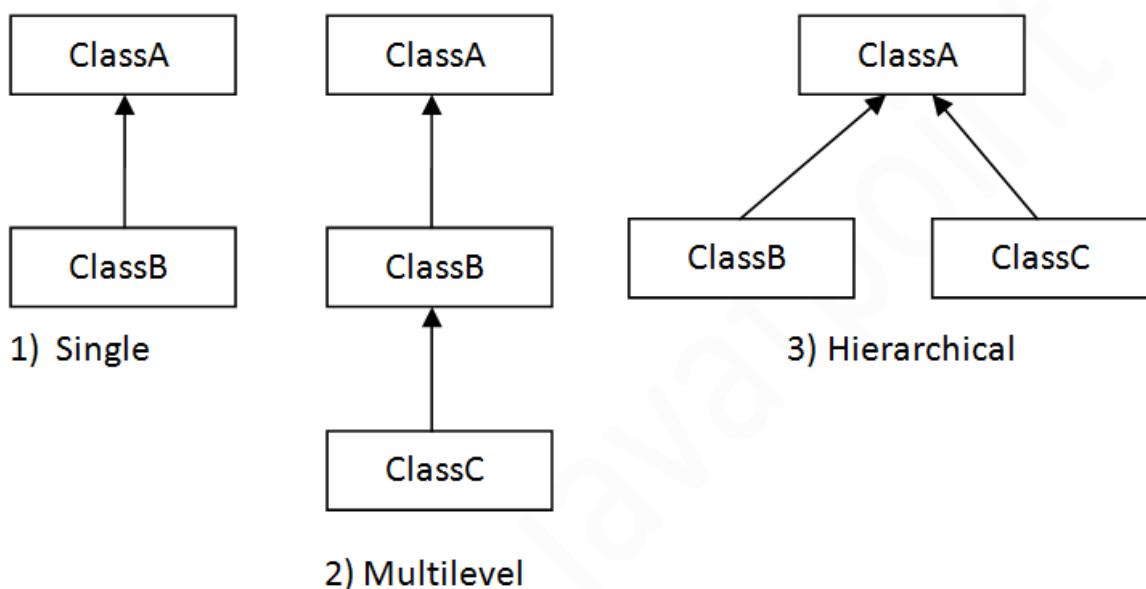
1. class Employee{
2. float salary=40000;
3. }
4. class Programmer extends Employee{
5. int bonus=10000;
6. public static void main(String args[]){
7. Programmer p=new Programmer();
8. System.out.println("Programmer salary is:"+p.salary);

```
9.     System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }
```

Types of inheritance in java

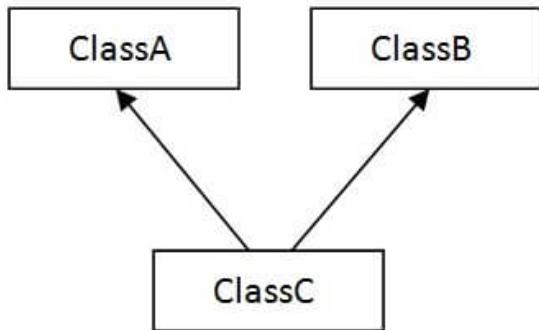
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

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

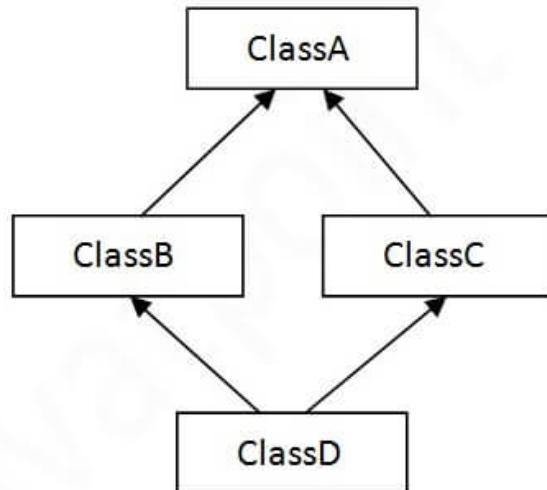


Note: Multiple inheritance is not supported in Java through class.

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



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

```

class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}

```

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

```

class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{

```

```

void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}

```

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

```

class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark(); //C.T.Error
}}}

```

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.

```
class A{  
void msg(){System.out.println("Hello");}  
}  
class B{  
void msg(){System.out.println("Welcome");}  
}  
class C extends A,B{//suppose if it were  
  
public static void main(String args[]){  
C obj=new C();  
obj.msg();//Now which msg() method would be invoked?  
}  
}
```

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.

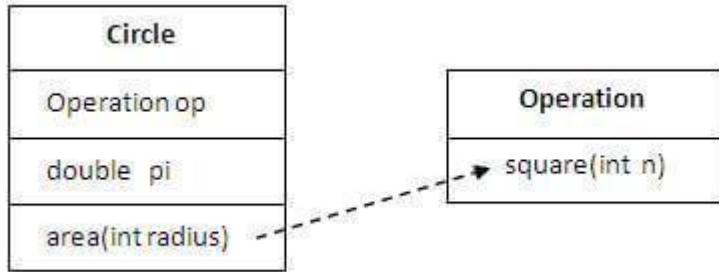
```
class Employee{  
int id;  
String name;  
Address address;//Address is a class  
...  
}
```

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

Why use Aggregation?

- For Code Reusability.
-

Simple Example of Aggregation



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

```
class Operation{
int square(int n){
    return n*n;
}
}
```

```
class Circle{
Operation op;//aggregation
double pi=3.14;

double area(int radius){
    op=new Operation();
    int rsquare=op.square(radius);//code reusability (i.e. delegates the method call).
    return pi*rsquare;
}
}
```

```
public static void main(String args[]){
    Circle c=new Circle();
    double result=c.area(5);
    System.out.println(result);
}
```

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.

Understanding meaningful example of Aggregation

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

```
public class Address {  
    String city,state,country;  
  
    public Address(String city, String state, String country) {  
        this.city = city;  
        this.state = state;  
        this.country = country;  
    }  
  
}
```

Emp.java

```
public class Emp {  
    int id;  
    String name;  
    Address address;  
  
    public Emp(int id, String name,Address address) {  
        this.id = id;  
        this.name = name;  
        this.address=address;  
    }  
  
    void display(){  
        System.out.println(id+" "+name);  
        System.out.println(address.city+" "+address.state+" "+address.country);  
    }  
  
    public static void main(String[] args) {  
        Address address1=new Address("gzb","UP","india");  
        Address address2=new Address("gno","UP","india");  
  
        Emp e=new Emp(111,"varun",address1);  
        Emp e2=new Emp(112,"arun",address2);  
  
        e.display();  
        e2.display();  
  
    }  
}
```

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.

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

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.

```
class Adder{  
    static int add(int a,int b){return a+b;}  
    static int add(int a,int b,int c){return a+b+c;}  
}  
class TestOverloading1{  
    public static void main(String[] args){  
        System.out.println(Adder.add(11,11));  
        System.out.println(Adder.add(11,11,11));  
    }  
}
```

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.

```
class Adder{  
    static int add(int a, int b){return a+b;}  
    static double add(double a, double b){return a+b;}  
}  
class TestOverloading2{  
    public static void main(String[] args){  
        System.out.println(Adder.add(11,11));  
        System.out.println(Adder.add(12.3,12.6));  
    }  
}
```

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:

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

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

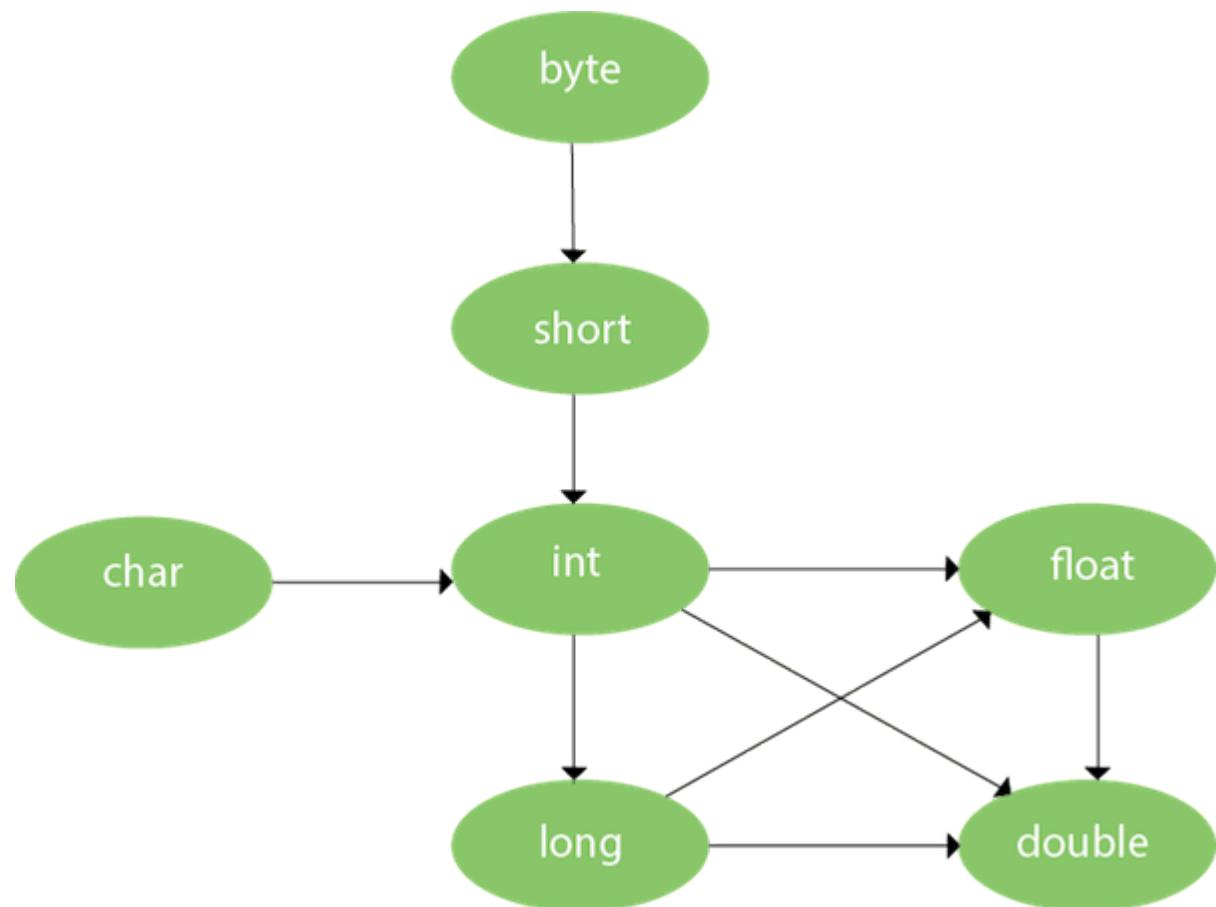
Can we overload java main() method?

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

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

Method Overloading and Type Promotion

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



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

Example of Method Overloading with TypePromotion

```
class OverloadingCalculation1{
    void sum(int a,long b){System.out.println(a+b);}
    void sum(int a,int b,int c){System.out.println(a+b+c);}

    public static void main(String args[]){
        OverloadingCalculation1 obj=new OverloadingCalculation1();
        obj.sum(20,20);//now second int literal will be promoted to long
        obj.sum(20,20,20);

    }
}
```

Example of Method Overloading with Type Promotion if matching found

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

```
class OverloadingCalculation2{
    void sum(int a,int b){System.out.println("int arg method invoked");}
    void sum(long a,long b){System.out.println("long arg method invoked");}

    public static void main(String args[]){
        OverloadingCalculation2 obj=new OverloadingCalculation2();
        obj.sum(20,20);//now int arg sum() method gets invoked
    }
}
```

Example of Method Overloading with Type Promotion in case of ambiguity

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

```
class OverloadingCalculation3{
    void sum(int a,long b){System.out.println("a method invoked");}
    void sum(long a,int b){System.out.println("b method invoked");}

    public static void main(String args[]){
        OverloadingCalculation3 obj=new OverloadingCalculation3();
        obj.sum(20,20);//now ambiguity
    }
}
```

One type is not de-promoted implicitly for example double cannot be depromoted to any type implicitly.

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.

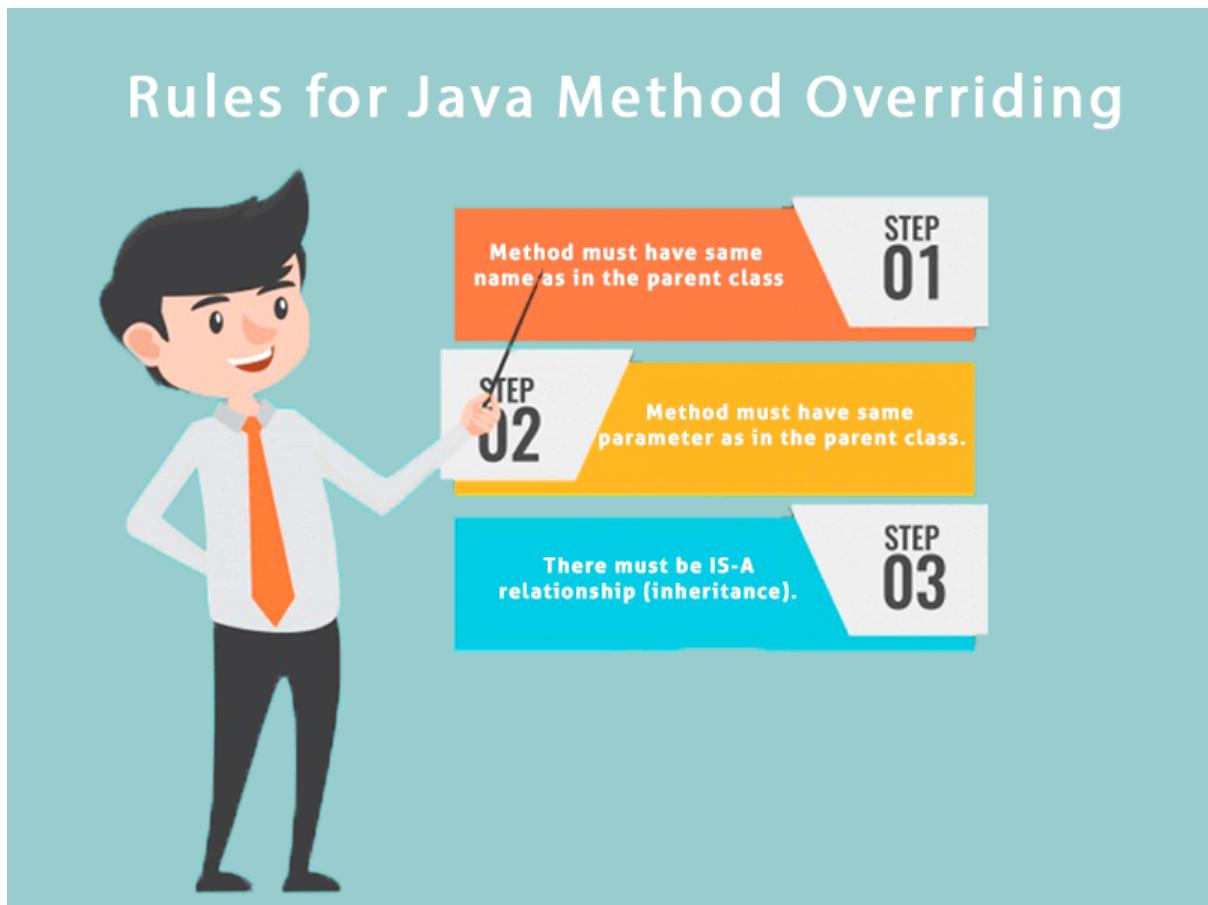
In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

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.

//Java Program to demonstrate why we need method overriding

```

//Here, we are calling the method of parent class with child
//class object.
//Creating a parent class
class Vehicle{
    void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike extends Vehicle{
    public static void main(String args[]){
        //creating an instance of child class
        Bike obj = new Bike();
        //calling the method with child class instance
        obj.run();
    }
}

```

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.

//Java Program to illustrate the use of Java Method Overriding

```

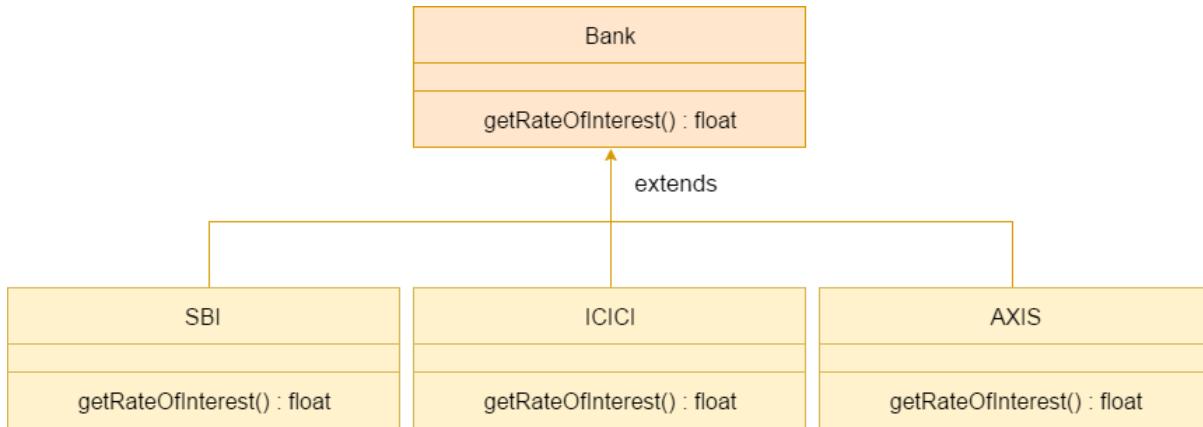
//Creating a parent class.
class Vehicle{
    //defining a method
    void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
    //defining the same method as in the parent class
    void run(){System.out.println("Bike is running safely");}
}

public static void main(String args[]){
    Bike2 obj = new Bike2();//creating object
    obj.run();//calling method
}

```

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 Program to demonstrate the real scenario of Java Method Overriding
//where three classes are overriding the method of a parent class.
//Creating a parent class.
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}

//Test class to create objects and call the methods
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}

```

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

No.	Method Overloading	Method Overriding
1)	Method overloading is used to increase the readability of the program.	Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
2)	Method overloading is performed within class.	Method overriding occurs in two classes that have IS-A (inheritance) relationship.
3)	In case of method overloading, parameter must be different.	In case of method overriding, parameter must be same.
4)	Method overloading is the example of compile time polymorphism.	Method overriding is the example of run time polymorphism.
5)	In java, method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading. But you must have to change the parameter.	Return type must be same or covariant in method overriding.

Java Method Overloading example

```
class OverloadingExample{  
    static int add(int a,int b){return a+b;}  
    static int add(int a,int b,int c){return a+b+c;}  
}
```

Java Method Overriding example

```
class Animal{  
void eat(){System.out.println("eating...");}  
}  
class Dog extends Animal{  
void eat(){System.out.println("eating bread...");}  
}
```

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.

AD

AD

Usage of 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.

```
class Animal{  
String color="white";  
}  
class Dog extends Animal{  
String color="black";  
void printColor(){  
System.out.println(color);//prints color of Dog class  
System.out.println(super.color);//prints color of Animal class  
}  
}  
class TestSuper1{  
public static void main(String args[]){  
Dog d=new Dog();  
d.printColor();  
}}
```

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 methods. It should be used if the subclass contains the same method as the parent class. In other words, it is used if the method is overridden.

```
class Animal{  
void eat(){System.out.println("eating...");}  
}  
class Dog extends Animal{  
void eat(){System.out.println("eating bread...");}  
void bark(){System.out.println("barking...");}  
void work(){  
super.eat();  
bark();  
}  
}  
class TestSuper2{  
public static void main(String args[]){  
Dog d=new Dog();  
d.work();  
}}
```

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.

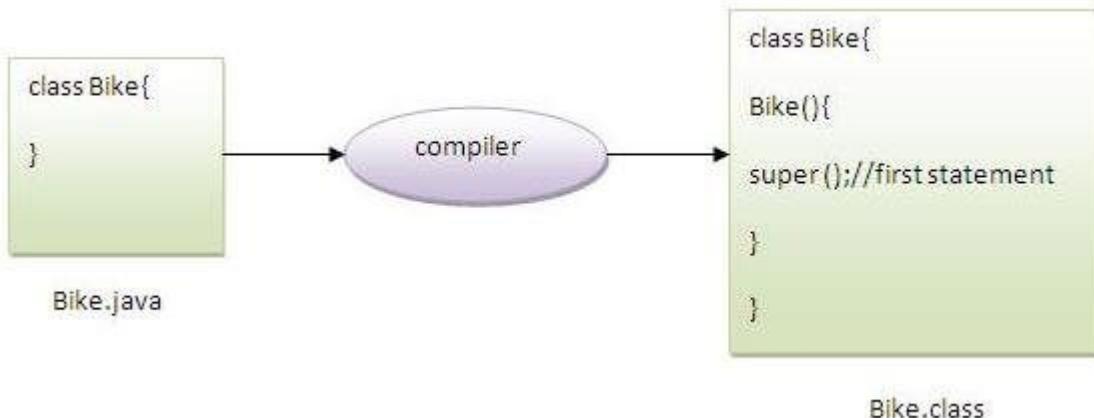
3) super is used to invoke parent class constructor.

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

```
class Animal{  
Animal(){System.out.println("animal is created");}  
}  
class Dog extends Animal{  
Dog(){  
super();  
System.out.println("dog is created");  
}  
}  
class TestSuper3{  
public static void main(String args[]){  
Dog d=new Dog();
```

```
}}
```

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.

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
System.out.println("dog is created");
}
}
class TestSuper4{
public static void main(String args[]){
Dog d=new Dog();
}}}
```

super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```
class Person{
int id;
String name;
Person(int id,String name){
this.id=id;
this.name=name;
}
}
class Emp extends Person{
float salary;
```

```
Emp(int id,String name,float salary){  
super(id,name);//reusing parent constructor  
this.salary=salary;  
}  
void display(){System.out.println(id+" "+name+" "+salary);}  
}  
class TestSuper5{  
public static void main(String[] args){  
Emp e1=new Emp(1,"ankit",45000f);  
e1.display();  
}}
```

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.

Java Final Keyword

- ⇒ Stop Value Change
- ⇒ Stop Method Overriding
- ⇒ Stop Inheritance

javatpoint.com

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.

```
class Bike9{  
    final int speedlimit=90;//final variable  
    void run(){  
        speedlimit=400;  
    }  
    public static void main(String args[]){  
        Bike9 obj=new Bike9();  
        obj.run();  
    }  
}//end of class
```

2) Java final method

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

Example of final method

```
class Bike{  
    final void run(){System.out.println("running");}  
}  
  
class Honda extends Bike{  
    void run(){System.out.println("running safely with 100kmph");}  
  
    public static void main(String args[]){  
        Honda honda= new Honda();  
        honda.run();  
    }  
}
```

3) Java final class

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

Example of final class

```
final class Bike{}  
  
class Honda1 extends Bike{  
    void run(){System.out.println("running safely with 100kmph");}  
  
    public static void main(String args[]){  
        Honda1 honda= new Honda1();  
        honda.run();  
    }  
}
```

Q) Is final method inherited?

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

```
class Bike{  
    final void run(){System.out.println("running...");}  
}  
class Honda2 extends Bike{  
    public static void main(String args[]){  
        new Honda2().run();  
    }  
}
```

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

```
class Student{  
    int id;  
    String name;  
    final String PAN_CARD_NUMBER;  
    ...  
}
```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
class Bike10{  
    final int speedlimit;//blank final variable  
  
    Bike10(){  
        speedlimit=70;  
        System.out.println(speedlimit);  
    }  
  
    public static void main(String args[]){
```

```
    new Bike10();
}
}
```

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

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

Q) What is final parameter?

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

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

Q) Can we declare a constructor final?

No, because constructor is never inherited.

Runtime Polymorphism in Java

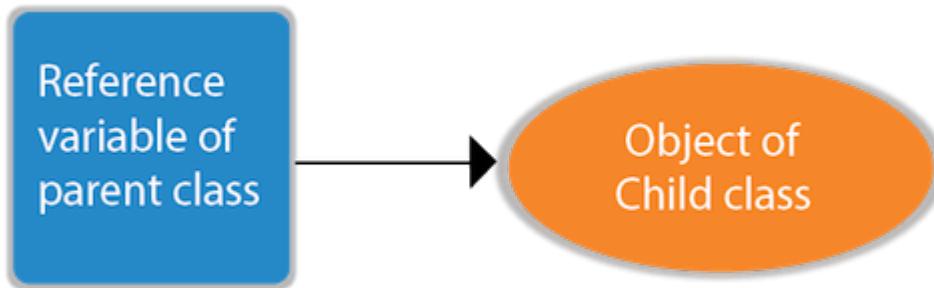
Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

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



```

class A{}
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{}

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 the 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 the compiler, it is known as runtime polymorphism.

```

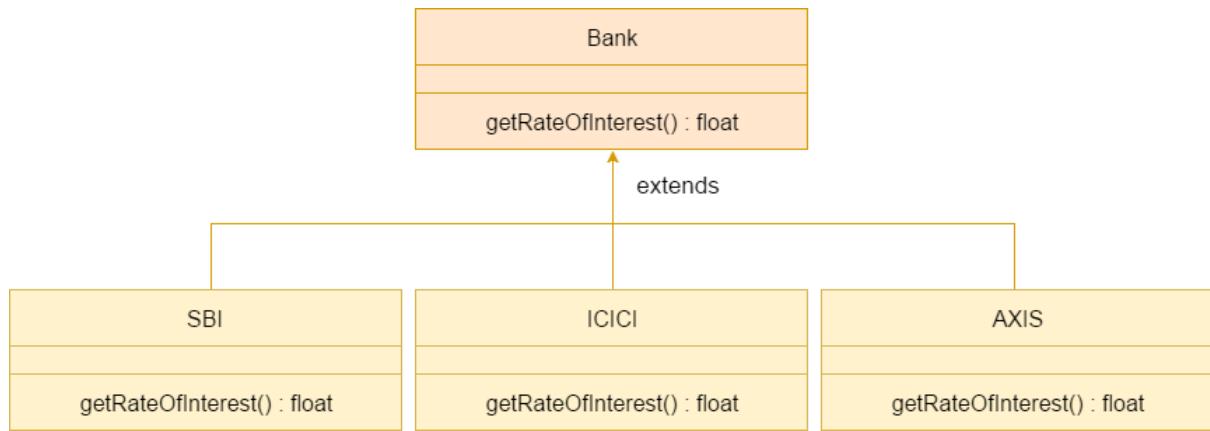
class Bike{
    void run(){System.out.println("running");}
}
class Splendor extends Bike{
    void run(){System.out.println("running safely with 60km");}
}

public static void main(String args[]){
    Bike b = new Splendor(); //upcasting
    b.run();
}
}

```

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.



```

class Bank{
float getRateOfInterest(){return 0;}
}
class SBI extends Bank{
float getRateOfInterest(){return 8.4f;}
}
class ICICI extends Bank{
float getRateOfInterest(){return 7.3f;}
}
class AXIS extends Bank{
float getRateOfInterest(){return 9.7f;}
}
class TestPolymorphism{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
b=new ICICI();
System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
b=new AXIS();
System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
}
}

```

Java Runtime Polymorphism Example: Shape

```

class Shape{
void draw(){System.out.println("drawing...");}
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle...");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing circle...");}
}

```

```

}
class Triangle extends Shape{
void draw(){System.out.println("drawing triangle...");}
}
class TestPolymorphism2{
public static void main(String args[]){
Shape s;
s=new Rectangle();
s.draw();
s=new Circle();
s.draw();
s=new Triangle();
s.draw();
}
}

```

Java Runtime Polymorphism Example: Animal

```

class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
}
class Cat extends Animal{
void eat(){System.out.println("eating rat...");}
}
class Lion extends Animal{
void eat(){System.out.println("eating meat...");}
}
class TestPolymorphism3{
public static void main(String[] args){
Animal a;
a=new Dog();
a.eat();
a=new Cat();
a.eat();
a=new Lion();
a.eat();
}}

```

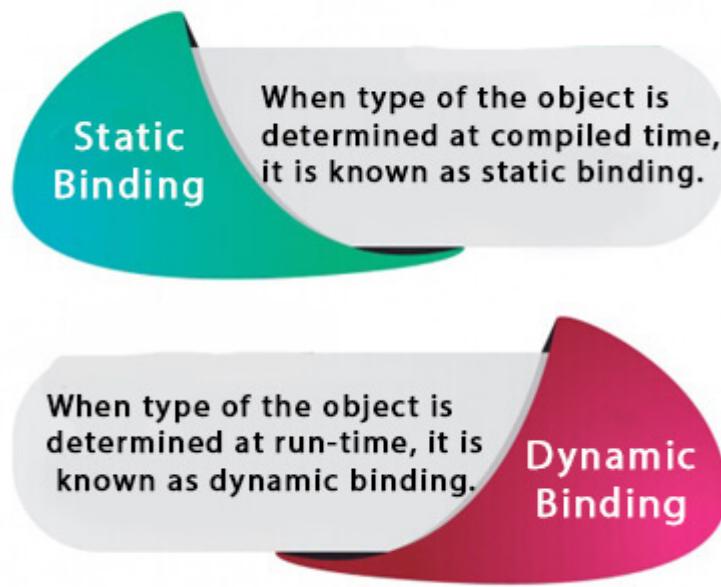
Static Binding and Dynamic Binding

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 vs Dynamic Binding



Understanding Type

Let's understand the type of instance.

1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

1. int data=30;

Here data variable is a type of int.

2) References have a type

```
class Dog{  
    public static void main(String args[]){  
        Dog d1;//Here d1 is a type of Dog  
    }  
}
```

3) Objects have a type

An object is an instance of a particular java class, but it is also an instance of its superclass.

```
class Animal{}

class Dog extends Animal{
    public static void main(String args[]){
        Dog d1=new Dog();
    }
}
```

Here d1 is an instance of Dog class, but it is also an instance of Animal.

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

```
class Dog{
    private void eat(){System.out.println("dog is eating...");}

    public static void main(String args[]){
        Dog d1=new Dog();
        d1.eat();
    }
}
```

Dynamic binding

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

Example of dynamic binding

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

class Dog extends Animal{
    void eat(){System.out.println("dog is eating...");}

    public static void main(String args[]){
        Animal a=new Dog();
    }
}
```

```
a.eat();
}
}
```

Java instanceof

The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type comparison operator because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Simple example of java instanceof

Let's see the simple example of instance operator where it tests the current class.

```
class Simple1{
    public static void main(String args[]){
        Simple1 s=new Simple1();
        System.out.println(s instanceof Simple1);//true
    }
}
```

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

Another example of java instanceof operator

```
class Animal{}
class Dog1 extends Animal{//Dog inherits Animal

    public static void main(String args[]){
        Dog1 d=new Dog1();
        System.out.println(d instanceof Animal);//true
    }
}
```

instanceof in java with a variable that have null value

If we apply instanceof operator with a variable that have null value, it returns false. Let's see the example given below where we apply instanceof operator with the variable that have null value.

```
class Dog2{
    public static void main(String args[]){
        Dog2 d=null;
        System.out.println(d instanceof Dog2);//false
    }
}
```

```
}
```

Downcasting with java instanceof operator

When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

1. Dog d=new Animal();//Compilation error

If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

1. Dog d=(Dog)new Animal();

//Compiles successfully but ClassCastException is thrown at runtime

Possibility of downcasting with instanceof

Let's see the example, where downcasting is possible by instanceof operator.

```
class Animal { }
```

```
class Dog3 extends Animal {
    static void method(Animal a) {
        if(a instanceof Dog3){
            Dog3 d=(Dog3)a;//downcasting
            System.out.println("ok downcasting performed");
        }
    }

    public static void main (String [] args) {
        Animal a=new Dog3();
        Dog3.method(a);
    }
}
```

Downcasting without the use of java instanceof

Downcasting can also be performed without the use of instanceof operator as displayed in the following example:

```
class Animal { }
class Dog4 extends Animal {
    static void method(Animal a) {
        Dog4 d=(Dog4)a;//downcasting
        System.out.println("ok downcasting performed");
    }

    public static void main (String [] args) {
        Animal a=new Dog4();
        Dog4.method(a);
    }
}
```

```
}
```

Let's take closer look at this, actual object that is referred by a, is an object of Dog class. So if we downcast it, it is fine. But what will happen if we write:

1. Animal a=new Animal();
2. Dog.method(a); //Now ClassCastException but not in case of instanceof operator

Understanding Real use of instanceof in java

Let's see the real use of instanceof keyword by the example given below.

```
interface Printable{}

class A implements Printable{
public void a(){System.out.println("a method");}
}

class B implements Printable{
public void b(){System.out.println("b method");}
}

class Call{
void invoke(Printable p){//upcasting
if(p instanceof A){
A a=(A)p;//Downcasting
a.a();
}
if(p instanceof B){
B b=(B)p;//Downcasting
b.b();
}
}
}//end of Call class

class Test4{
public static void main(String args[]){
Printable p=new B();
Call c=new Call();
c.invoke(p);
}
}
```

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%)

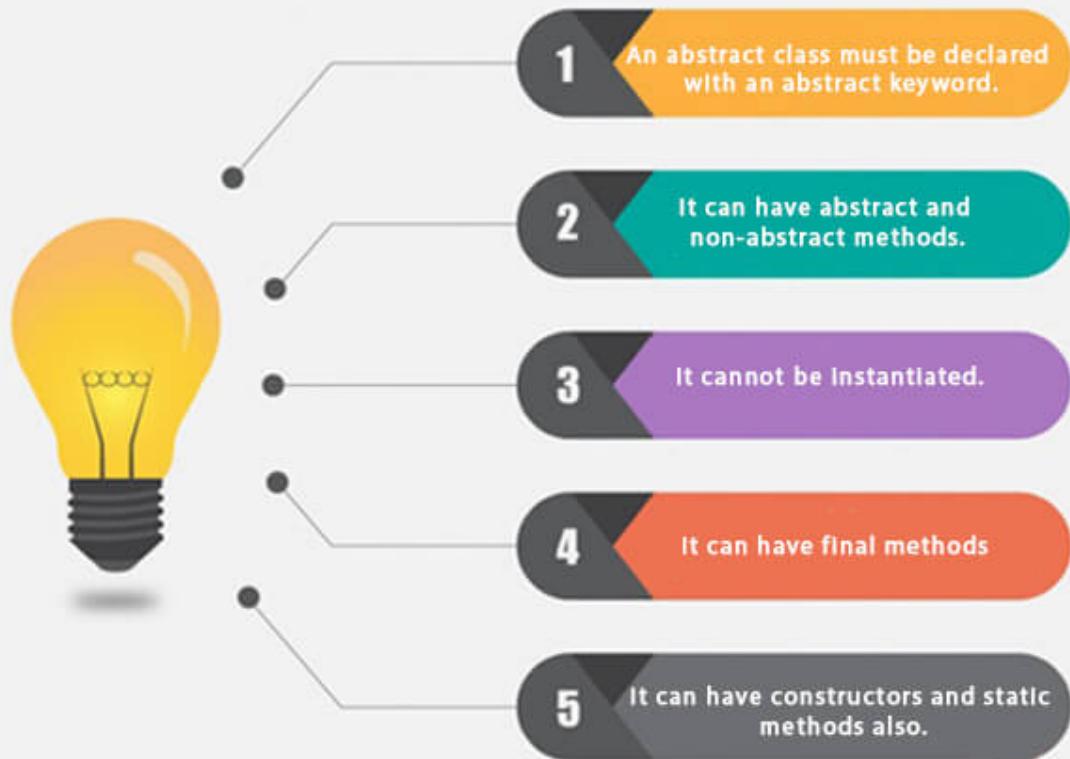
Abstract class in Java

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

1. 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.

```
abstract class Bike{  
    abstract void run();  
}  
class Honda4 extends Bike{  
    void run(){System.out.println("running safely");}  
    public static void main(String args[]){  
        Bike obj = new Honda4();  
        obj.run();  
    }  
}
```

```
}
```

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

```
abstract class Shape{  
    abstract void draw();  
}  
  
//In real scenario, implementation is provided by others i.e. unknown by end user  
class Rectangle extends Shape{  
    void draw(){System.out.println("drawing rectangle");}  
}  
  
class Circle1 extends Shape{  
    void draw(){System.out.println("drawing circle");}  
}  
  
//In real scenario, method is called by programmer or user  
class TestAbstraction1{  
    public static void main(String args[]){  
        Shape s=new Circle1();  
        //In a real scenario, object is provided through method, e.g.,  
        //getShape() method  
        s.draw();  
    }  
}
```

Another example of Abstract class in java

File: TestBank.java

```
abstract class Bank{  
    abstract int getRateOfInterest();  
}  
  
class SBI extends Bank{  
    int getRateOfInterest(){return 7;}  
}  
  
class PNB extends Bank{  
    int getRateOfInterest(){return 8;}  
}  
  
class TestBank{
```

```

public static void main(String args[]){
    Bank b;
    b=new SBI();
    System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
    b=new PNB();
    System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}

```

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

```

//Example of an abstract class that has abstract and non-abstract methods
abstract class Bike{
    Bike(){System.out.println("bike is created");}
    abstract void run();
    void changeGear(){System.out.println("gear changed");}
}

//Creating a Child class which inherits Abstract class
class Honda extends Bike{
    void run(){System.out.println("running safely..");}
}

//Creating a Test class which calls abstract and non-abstract methods
class TestAbstraction2{
    public static void main(String args[]){
        Bike obj = new Honda();
        obj.run();
        obj.changeGear();
    }
}

```

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

```

class Bike12{
    abstract void run();
}

```

Rule: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

Interface in Java

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

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also represents the IS-A relationship.

It cannot be instantiated just like the abstract class.

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

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

Why use Java interface?

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

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



How to declare an interface?

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

Syntax:

```
interface <interface_name>{

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

}
```

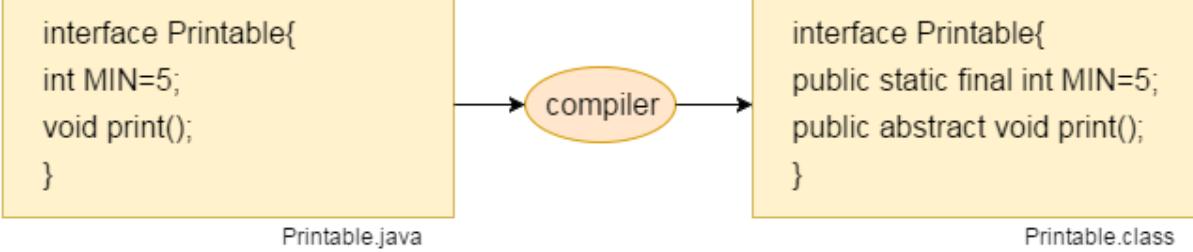
Java 8 Interface Improvement

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

Internal addition by the compiler

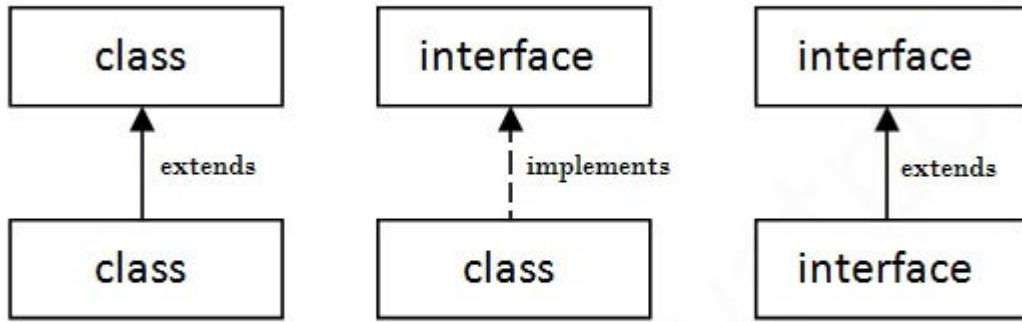
The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

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.

```

interface printable{
void print();
}

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

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

```

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
//Interface declaration: by first user
interface Drawable{
void draw();
}

//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

```

```

}

class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}

//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
d.draw();
}}

```

Java Interface Example: Bank

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

File: TestInterface2.java

```

interface Bank{
float rateOfInterest();
}

class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}

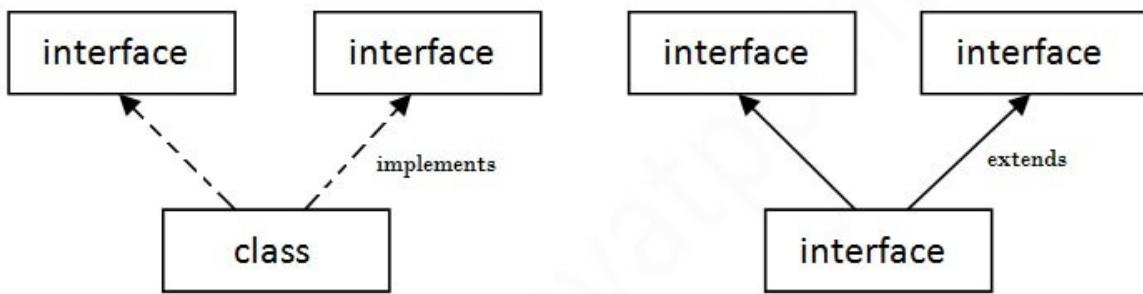
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}

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

```

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

```

interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
}
}

```

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

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

```

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

class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
}

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

```

```
TestInterface3 obj = new TestInterface3();
obj.print();
}
}
```

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.

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

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

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
interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
```

```
d.draw();
d.msg();
}}
```

Java 8 Static Method in Interface

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

File: TestInterfaceStatic.java

```
interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable(cube(3)));
}}
```

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.

//How Serializable interface is written?

```
public interface Serializable{
}
```

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:

```
interface printable{
void print();
interface MessagePrintable{
void msg();
}
}
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

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

Java Package

A java package is a group of similar types of classes, interfaces and sub-packages.

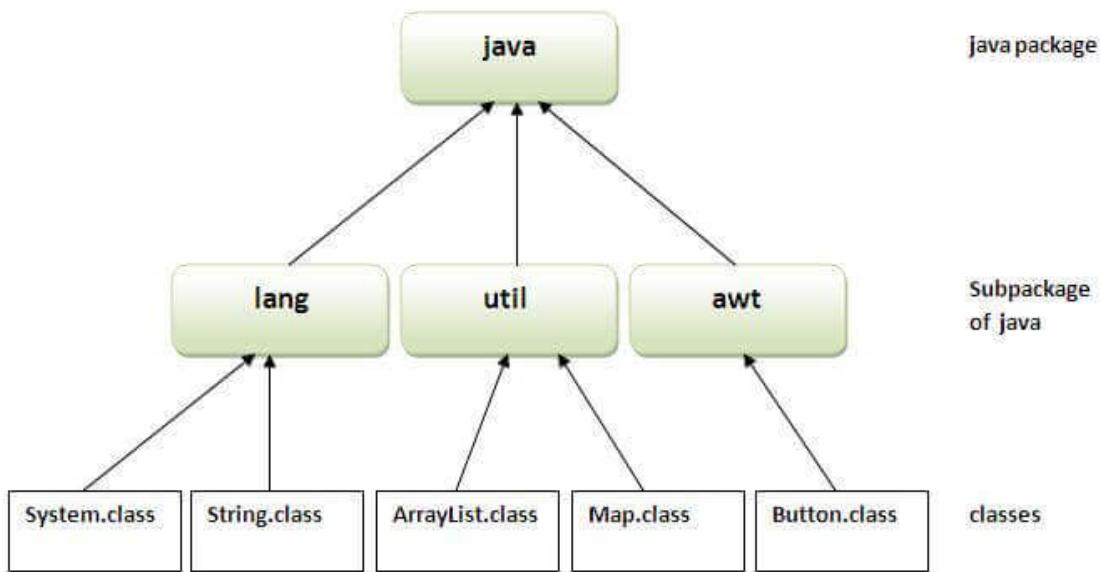
Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

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



Simple example of java package

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

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

How to access package from another package?

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

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

1) Using packagename.*

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

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

Example of package that import the packagename.*

```
//save by A.java
```

```

package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;

class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}

```

2) Using packagename.classname

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

Example of package by import package.classname

```

//save by A.java

package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.A;

class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}

```

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
```

```

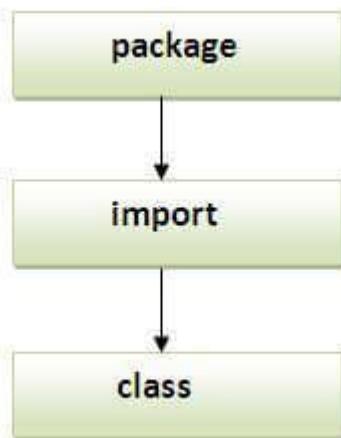
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
    public static void main(String args[]){
        pack.A obj = new pack.A(); //using fully qualified name
        obj.msg();
    }
}

```

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

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

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



Subpackage in java

Package inside the package is called the subpackage. It should be created to categorize the package further.

Let's take an example, Sun Microsystem has defined a package named `java` that contains many classes like `System`, `String`, `Reader`, `Writer`, `Socket` etc. These classes represent a particular group e.g. `Reader` and `Writer` classes are for Input/Output operation, `Socket` and `ServerSocket` classes are for networking etc and so on. So, Sun has subcategorized the `java` package into subpackages such as `lang`, `net`, `io` etc. and put the Input/Output related classes in `io` package, `Server` and `ServerSocket` classes in `net` packages and so on.

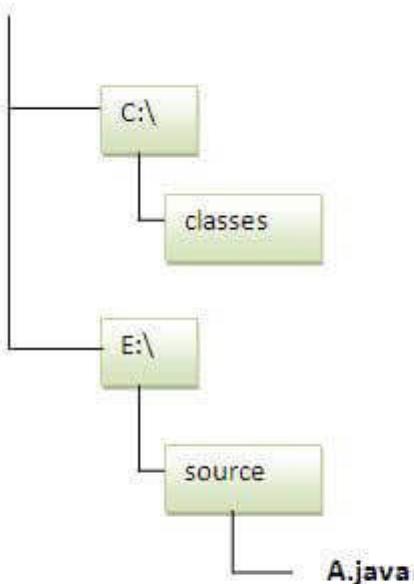
The standard of defining a package is `domain.company.package` e.g. `com.javatpoint.bean` or `org.sssit.dao`.

Example of Subpackage

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

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

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



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

What about package class?

Package class

The package class provides methods to get information about the specification and implementation of a package. It provides methods such as getName(), getImplementationTitle(), getImplementationVendor(), getImplementationVersion() etc.

Example of Package class

In this example, we are printing the details of java.lang package by invoking the methods of package class.

```
class PackageInfo{  
    public static void main(String args[]){  
  
        Package p=Package.getPackage("java.lang");  
  
        System.out.println("package name: "+p.getName());  
  
        System.out.println("Specification Title: "+p.getSpecificationTitle());  
        System.out.println("Specification Vendor: "+p.getSpecificationVendor());  
        System.out.println("Specification Version: "+p.getSpecificationVersion());  
  
        System.out.println("Implementaion Title: "+p.getImplementationTitle());
```

```
System.out.println("Implementation Vendor:  
"+p.getImplementationVendor());  
  
System.out.println("Implementation Version:  
"+p.getImplementationVersion());  
  
System.out.println("Is sealed: "+p.isSealed());  
  
}  
}
```

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

```
//A Java class which is a fully encapsulated class.
```

```
//It has a private data member and getter and setter methods.
```

```
package com.javatpoint;
```

```
public class Student{
```

```
    //private data member
```

```
    private String name;
```

```
    //getter method for name
```

```
    public String getName(){
```

```
        return name;
```

```
}
```

```
//setter method for name

public void setName(String name){

this.name=name

}

}
```

```
File: Test.java
//A Java class to test the encapsulated class.
package com.javatpoint;
class Test{
public static void main(String[] args){
//creating instance of the encapsulated class
Student s=new Student();
//setting value in the name member
s.setName("vijay");
//getting value of the name member
System.out.println(s.getName());
}
}
```

Read-Only class

//A Java class which has only getter methods.

```
public class Student{  
    //private data member  
    private String college="AKG";  
    //getter method for college  
    public String getCollege(){  
        return college;  
    }  
}
```

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

Write-Only class

//A Java class which has only setter methods.

```
public class Student{  
    //private data member  
    private String college;  
    //getter method for college  
    public void setCollege(String college){  
        this.college=college;  
    }  
}
```

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

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

Another Example of Encapsulation in Java

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

File: Account.java

```
//A Account class which is a fully encapsulated class.  
//It has a private data member and getter and setter methods.  
class Account {  
    //private data members  
    private long acc_no;  
    private String name,email;  
    private float amount;  
    //public getter and setter methods  
    public long getAcc_no() {  
        return acc_no;  
    }  
    public void setAcc_no(long acc_no) {  
        this.acc_no = acc_no;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getEmail() {  
        return email;  
    }  
    public void setEmail(String email) {  
        this.email = email;  
    }  
    public float getAmount() {  
        return amount;  
    }  
    public void setAmount(float amount) {  
        this.amount = amount;  
    }  
}
```

Object class in Java

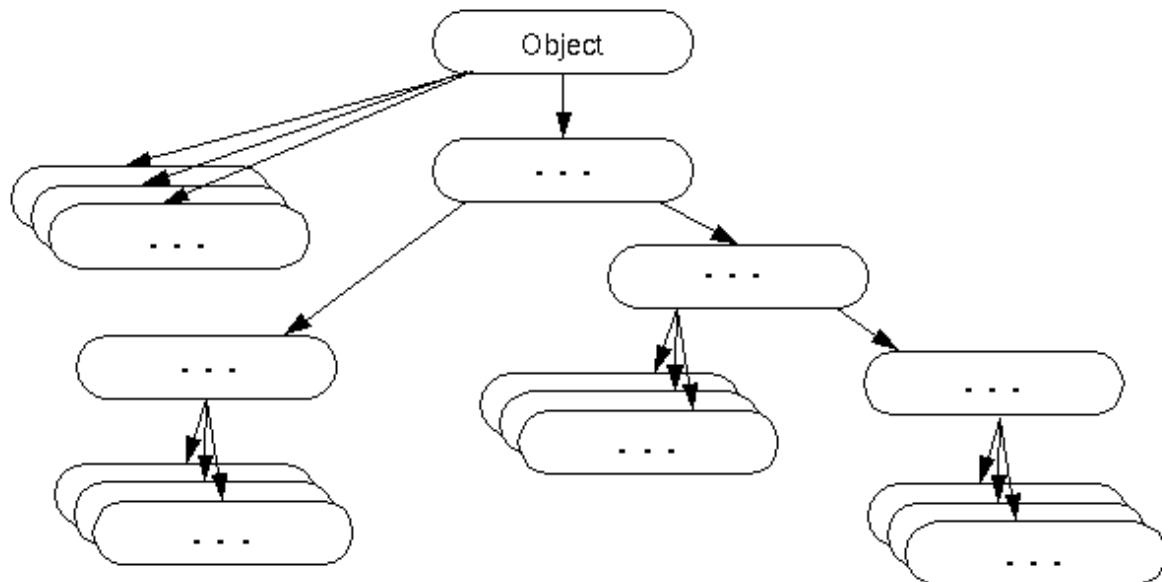
The Object class is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, known as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee, Student etc, we can use Object class reference to refer that object. For example:

1. Object obj=getObject(); //we don't know what object will be returned from this method

The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.



Methods of Object class

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

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.

public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout) throws InterruptedException	causes the current thread to wait for the specified milliseconds, until 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.

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:

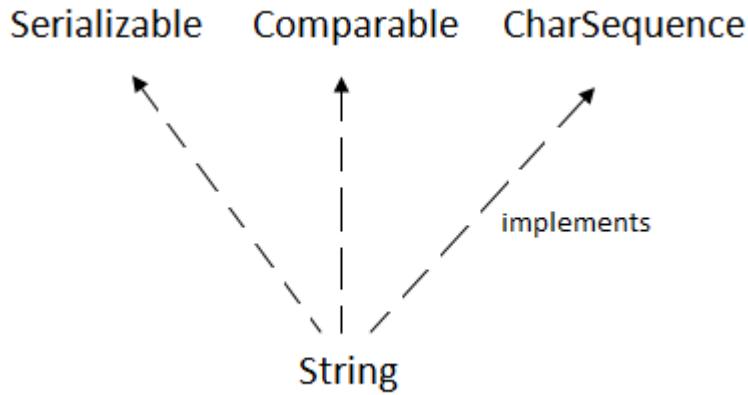
1. `char[] ch={'j','a','v','a','t','p','o','i','n','t'};`
2. `String s=new String(ch);`

is same as:

1. `String s="javatpoint";`

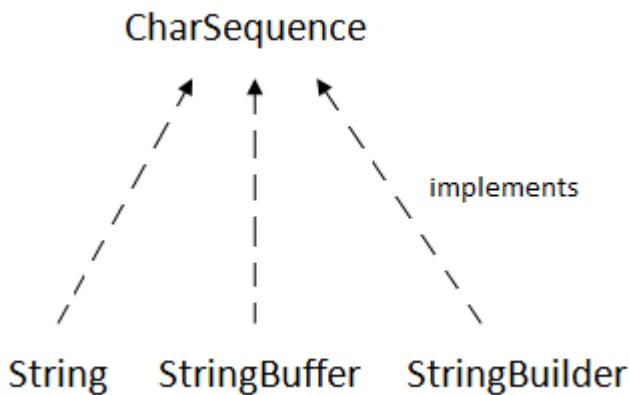
Java String class provides a lot of methods to perform operations on strings such as `compare()`, `concat()`, `equals()`, `split()`, `length()`, `replace()`, `compareTo()`, `intern()`, `substring()` etc.

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 `String` in Java is 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

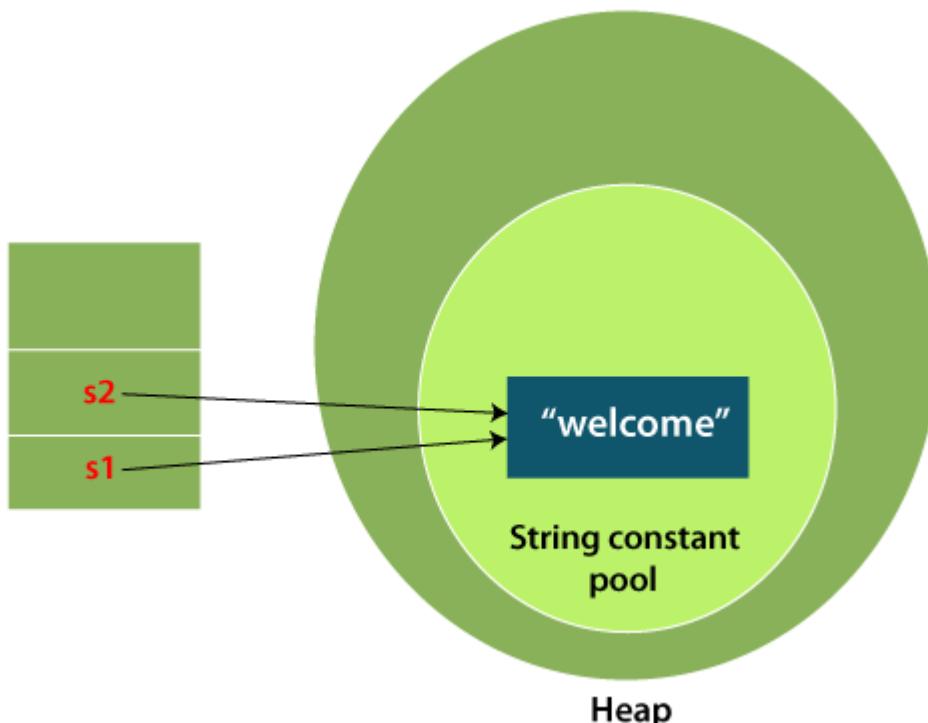
1) String Literal

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

StringExample.java

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

The above code, converts a char array into a String object. And displays the String objects s1, s2, and s3 on console using println() method.

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	char charAt(int index)	It returns char value for the particular index
2	int length()	It returns string length
3	static String format(String format, Object... args)	It returns a formatted string.
4	static String format(Locale l, String format, Object... args)	It returns formatted string with given locale.
5	String substring(int beginIndex)	It returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	It returns substring for given begin index and end index.

7	boolean contains(CharSequence s)	It returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	It returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	It returns a joined string.
10	boolean equals(Object another)	It checks the equality of string with the given object.
11	boolean isEmpty()	It checks if string is empty.
12	String concat(String str)	It concatenates the specified string.
13	String replace(char old, char new)	It replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	It replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	It compares another string. It doesn't check case.
16	String[] split(String regex)	It returns a split string matching regex.
17	String[] split(String regex, int limit)	It returns a split string matching regex and limit.
18	String intern()	It returns an interned string.
19	int indexOf(int ch)	It returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	It returns the specified char value index starting with given index.

21	int indexOf(String substring)	It returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	It returns the specified substring index starting with given index.
23	String toLowerCase()	It returns a string in lowercase.
24	String toLowerCase(Locale l)	It returns a string in lowercase using specified locale.
25	String toUpperCase()	It returns a string in uppercase.
26	String toUpperCase(Locale l)	It returns a string in uppercase using specified locale.
27	String trim()	It removes beginning and ending spaces of this string.
28	static String valueOf(int value)	It converts given type into string. It is an overloaded method.

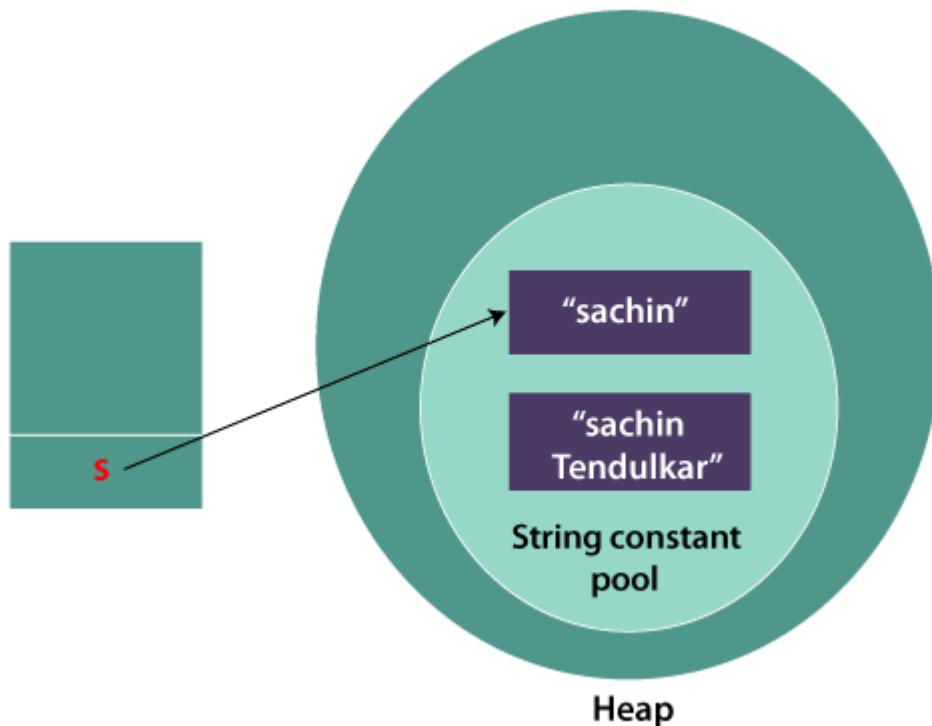
Immutable String in Java

A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc. 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 concept of immutability by the example given below:

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

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with Sachin Tendulkar. 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:

```
Testimmutablestring1.java
class Testimmutablestring1{
    public static void main(String args[]){
        String s="Sachin";
        s=s.concat(" Tendulkar");
        System.out.println(s);
    }
}
```

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

Why String objects are immutable in Java?

As 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 by all the reference variables. That is why String objects are immutable in Java.

Following are some features of String which makes String objects immutable.

1. ClassLoader:

A ClassLoader in Java uses a String object as an argument. Consider, if the String object is modifiable, the value might be changed and the class that is supposed to be loaded might be different.

To avoid this kind of misinterpretation, String is immutable.

2. Thread Safe:

As the String object is immutable we don't have to take care of the synchronization that is required while sharing an object across multiple threads.

3. Security:

As we have seen in class loading, immutable String objects avoid further errors by loading the correct class. This leads to making the application program more secure. Consider an example of banking software. The username and password cannot be modified by any intruder because String objects are immutable. This can make the application program more secure.

4. Heap Space:

AD

The immutability of String helps to minimize the usage in the heap memory. When we try to declare a new String object, the JVM checks whether the value already exists in the String pool or not. If it exists, the same value is assigned to the new object. This feature allows Java to use the heap space efficiently.

Why String class is Final in Java?

The reason behind the String class being final is because no one can override the methods of the String class. So that it can provide the same features to the new String objects as well as to the old ones.

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 Using equals() Method
2. By Using == Operator
3. By compareTo() Method

1) By Using equals() Method

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

Teststringcomparison1.java

```
class Teststringcomparison1{
    public static void main(String args[]){
        String s1="Sachin";
        String s2="Sachin";
        String s3=new String("Sachin");
        String s4="Saurav";
        System.out.println(s1.equals(s2));//true
        System.out.println(s1.equals(s3));//true
    }
}
```

```
        System.out.println(s1.equals(s4));//false
    }
}
```

In the above code, two strings are compared using equals() method of String class. And the result is printed as boolean values, true or false.

Teststringcomparison2.java

```
class Teststringcomparison2{
    public static void main(String args[]){
        String s1="Sachin";
        String s2="SACHIN";

        System.out.println(s1.equals(s2));//false
        System.out.println(s1.equalsIgnoreCase(s2));//true
    }
}
```

In the above program, the methods of String class are used. The equals() method returns true if String objects are matching and both strings are of same case. equalsIgnoreCase() returns true regardless of cases of strings.

2) By Using == operator

The == operator compares references not values.

Teststringcomparison3.java

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

3) String compare by compareTo() method

The above code, demonstrates the use of == operator used for comparing two String objects.

3) By Using compareTo() method

The String class 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 objects. If:

- s1 == s2 : The method returns 0.
- s1 > s2 : The method returns a positive value.
- s1 < s2 : The method returns a negative value.

Teststringcomparison4.java

```

class Teststringcomparison4{
public static void main(String args[]){
    String s1="Sachin";
    String s2="Sachin";
    String s3="Ratan";
    System.out.println(s1.compareTo(s2));//0
    System.out.println(s1.compareTo(s3));//1(because s1>s3)
    System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
}
}

```

String Concatenation in Java

In Java, String concatenation forms a new String that is the combination of multiple strings.
There are two ways to concatenate strings 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:

```

TestStringConcatenation1.java
class TestStringConcatenation1{
public static void main(String args[]){
    String s="Sachin"+" Tendulkar";
    System.out.println(s);//Sachin Tendulkar
}
}

```

The Java compiler transforms above code to this:

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

In Java, String concatenation is implemented through the `StringBuilder` (or `StringBuffer`) class and it's `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 concatenate not only String but primitive values also. For Example:

```

TestStringConcatenation2.java
class TestStringConcatenation2{
public static void main(String args[]){
    String s=50+30+"Sachin"+40+40;
    System.out.println(s);//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.

TestStringConcatenation3.java

```
class TestStringConcatenation3{
    public static void main(String args[]){
        String s1="Sachin ";
        String s2="Tendulkar";
        String s3=s1.concat(s2);
        System.out.println(s3);//Sachin Tendulkar
    }
}
```

The above Java program, concatenates two String objects s1 and s2 using concat() method and stores the result into s3 object.

There are some other possible ways to concatenate Strings in Java,

1. String concatenation using StringBuilder class

StringBuilder is class provides append() method to perform concatenation operation. The append() method accepts arguments of different types like Objects, StringBuilder, int, char, CharSequence, boolean, float, double. StringBuilder is the most popular and fastet way to concatenate strings in Java. It is mutable class which means values stored in StringBuilder objects can be updated or changed.

StrBuilder.java

```
public class StrBuilder
{
    /* Driver Code */
    public static void main(String args[])
    {
        StringBuilder s1 = new StringBuilder("Hello"); //String 1
        StringBuilder s2 = new StringBuilder(" World"); //String 2
        StringBuilder s = s1.append(s2); //String 3 to store the result
        System.out.println(s.toString()); //Displays result
    }
}
```

In the above code snippet, s1, s2 and s are declared as objects of StringBuilder class. s stores the result of concatenation of s1 and s2 using append() method.

2. String concatenation using format() method

String.format() method allows to concatenate multiple strings using format specifier like %s followed by the string values or objects.

StrFormat.java

```
public class StrFormat
```

```
{
```

```
/* Driver Code */
```

```

public static void main(String args[])
{
    String s1 = new String("Hello"); //String 1
    String s2 = new String(" World"); //String 2
    String s = String.format("%s%s",s1,s2); //String 3 to store the result
    System.out.println(s.toString()); //Displays result
}
}

```

Here, the String objects s is assigned the concatenated result of Strings s1 and s2 using String.format() method. format() accepts parameters as format specifier followed by String objects or values.

3. String concatenation using String.join() method (Java Version 8+)

The String.join() method is available in Java version 8 and all the above versions. String.join() method accepts arguments first a separator and an array of String objects.

StrJoin.java:

```

public class StrJoin
{
    /* Driver Code */
    public static void main(String args[])
    {
        String s1 = new String("Hello"); //String 1
        String s2 = new String(" World"); //String 2
        String s = String.join("",s1,s2); //String 3 to store the result
        System.out.println(s.toString()); //Displays result
    }
}

```

In the above code snippet, the String object s stores the result of String.join("",s1,s2) method. A separator is specified inside quotation marks followed by the String objects or array of String objects.

4. String concatenation using StringJoiner class (Java Version 8+)

StringJoiner class has all the functionalities of String.join() method. In advance its constructor can also accept optional arguments, prefix and suffix.

StrJoiner.java

```

public class StrJoiner
{
    /* Driver Code */

```

```

public static void main(String args[])
{
    StringJoiner s = new StringJoiner(", "); //StringJoiner object
    s.add("Hello"); //String 1
    s.add("World"); //String 2
    System.out.println(s.toString()); //Displays result
}
}

```

In the above code snippet, the StringJoiner object s is declared and the constructor StringJoiner() accepts a separator value. A separator is specified inside quotation marks. The add() method appends Strings passed as arguments.

Substring in Java

A part of String is called substring. In other words, substring is a subset of another String. Java String class provides the built-in substring() method that extract a substring from the given string by using the index values passed as an argument. In case of substring() method startIndex is inclusive and endIndex is exclusive.

Suppose the string is "computer", then the substring will be com, compu, ter, etc. 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). The method throws an IndexOutOfBoundsException when the startIndex is larger than the length of String or less than zero.
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. The method throws an IndexOutOfBoundsException when the startIndex is less than zero or startIndex is greater than endIndex or endIndex is greater than length of String.

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)); //returns he as a substring

In the above substring, 0 points the first letter and 2 points the second letter i.e., e (because end index is exclusive).

Example of Java substring() method

```
TestSubstring.java
public class TestSubstring{
    public static void main(String args[]){
        String s="SachinTendulkar";
        System.out.println("Original String: " + s);
        System.out.println("Substring starting from index 6: " +s.substring(6));//Tendulkar
        System.out.println("Substring starting from index 0 to 6: "+s.substring(0,6)); //Sachin
    }
}
```

Using String.split() method:

The split() method of String class can be used to extract a substring from a sentence. It accepts arguments in the form of a regular expression.

```
TestSubstring2.java
```

```
import java.util.*;

public class TestSubstring2
{
    /* Driver Code */
    public static void main(String args[])
    {
        String text= new String("Hello, My name is Sachin");
        /* Splits the sentence by the delimiter passed as an argument */
        String[] sentences = text.split("\\.");
        System.out.println(Arrays.toString(sentences));
    }
}
```

Java String Class Methods

The `java.lang.String` class provides a lot of built-in methods that are used to manipulate string in Java. By the help of these methods, we can perform operations on String objects 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 use some 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.

```
Stringoperation1.java
```

```
public class Stringoperation1
{
    public static void main(String ar[])
}
```

```
{  
String s="Sachin";  
System.out.println(s.toUpperCase());//SACHIN  
System.out.println(s.toLowerCase());//sachin  
System.out.println(s);//Sachin(no change in original)  
}  
}
```

Java String trim() method

The String class trim() method eliminates white spaces before and after the String.

```
Stringoperation2.java  
public class Stringoperation2  
{  
public static void main(String ar[])  
{  
String s=" Sachin ";  
System.out.println(s);// Sachin  
System.out.println(s.trim());//Sachin  
}  
}
```

Java String startsWith() and endsWith() method

The method startsWith() checks whether the String starts with the letters passed as arguments and endsWith() method checks whether the String ends with the letters passed as arguments.

```
Stringoperation3.java  
public class Stringoperation3  
{  
public static void main(String ar[])  
{  
String s="Sachin";  
System.out.println(s.startsWith("Sa"));//true  
System.out.println(s.endsWith("n"));//true  
}  
}
```

Java String charAt() Method

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

```
Stringoperation4.java  
public class Stringoperation4  
{  
public static void main(String ar[])  
{  
String s="Sachin";  
System.out.println(s.charAt(0));//S
```

```
System.out.println(s.charAt(3));//h
}
}
```

Java String length() Method

The String class length() method returns length of the specified String.

```
Stringoperation5.java
public class Stringoperation5
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.length());//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.

```
Stringoperation6.java
public class Stringoperation6
{
public static void main(String ar[])
{
String s=new String("Sachin");
String s2=s.intern();
System.out.println(s2);//Sachin
}
}
```

Java String valueOf() Method

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

```
Stringoperation7.java
public class Stringoperation7
{
public static void main(String ar[])
{
int a=10;
String s=String.valueOf(a);
System.out.println(s+10);
}
}
```

```
}
```

Java String replace() Method

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

Stringoperation8.java

```
public class Stringoperation8
{
    public static void main(String ar[])
    {
        String s1="Java is a programming language. Java is a platform. Java is an Island.";
        String replaceString=s1.replace("Java","Kava");//replaces all occurrences of "Java" to "Kava"
        System.out.println(replaceString);
    }
}
```

Java String charAt()

The Java String class 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 the length of the string. It returns StringIndexOutOfBoundsException, if the given index number is greater than or equal to this string length or a negative number.

Syntax

1. `public char charAt(int index)`

The method accepts index as a parameter. The starting index is 0. It returns a character at a specific index position in a string. It throws StringIndexOutOfBoundsException if the index is a negative value or greater than this string length.

Specified by

CharSequence interface, located inside java.lang package.

Internal implementation

```
public char charAt(int index) {
    if ((index < 0) || (index >= value.length)) {
        throw new StringIndexOutOfBoundsException(index);
    }
    return value[index];
}
```

Java String charAt() Method Examples

Let's see Java program related to string in which we will use charAt() method that perform some operation on the give string.

FileName: CharAtExample.java

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

Let's see the example of the `charAt()` method where we are passing a greater index value. In such a case, it throws `StringIndexOutOfBoundsException` at run time.

FileName: `CharAtExample.java`

```
public class CharAtExample{
public static void main(String args[]){
String name="javatpoint";
char ch=name.charAt(10);//returns the char value at the 10th index
System.out.println(ch);
}}
```

Accessing First and Last Character by Using the `charAt()` Method

Let's see a simple example where we are accessing first and last character from the provided string.

FileName: `CharAtExample3.java`

```
public class CharAtExample3 {
    public static void main(String[] args) {
        String str = "Welcome to Javatpoint portal";
        int strLength = str.length();
        // Fetching first character
        System.out.println("Character at 0 index is: "+ str.charAt(0));
        // The last Character is present at the string length-1 index
        System.out.println("Character at last index is: "+ str.charAt(strLength-1));
    }
}
```

Print Characters Presented at Odd Positions by Using the `charAt()` Method

Let's see an example where we are accessing all the elements present at odd index.

FileName: `CharAtExample4.java`

```
public class CharAtExample4 {
    public static void main(String[] args) {
        String str = "Welcome to Javatpoint portal";
        for (int i=0; i<=str.length()-1; i++) {
            if(i%2!=0) {
                System.out.println("Char at "+i+" place "+str.charAt(i));
            }
        }
    }
}
```

Counting Frequency of a character in a String by Using the charAt() Method

Let's see an example in which we are counting frequency of a character in the given string.

FileName: CharAtExample5.java

```
public class CharAtExample5 {  
    public static void main(String[] args) {  
        String str = "Welcome to Javatpoint portal";  
        int count = 0;  
        for (int i=0; i<=str.length()-1; i++) {  
            if(str.charAt(i) == 't') {  
                count++;  
            }  
        }  
        System.out.println("Frequency of t is: "+count);  
    }  
}
```

Java String compareTo()

The Java String class compareTo() method compares the given string with the current string lexicographically. It returns a positive number, negative number, or 0.

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

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

1. if $s_1 > s_2$, it returns positive number
2. if $s_1 < s_2$, it returns negative number
3. if $s_1 == s_2$, it returns 0

Syntax

1. `public int compareTo(String anotherString)`

The method accepts a parameter of type String that is to be compared with the current string.

It returns an integer value. It throws the following two exceptions:

`ClassCastException`: If this object cannot get compared with the specified object.

`NullPointerException`: If the specified object is null.

Internal implementation

```
int compareTo(String anotherString) {  
    int length1 = value.length;
```

```

int length2 = anotherString.value.length;
int limit = Math.min(length1, length2);
char v1[] = value;
char v2[] = anotherString.value;

int i = 0;
while (i < limit) {
    char ch1 = v1[i];
    char ch2 = v2[i];
    if (ch1 != ch2) {
        return ch1 - ch2;
    }
    i++;
}
return length1 - length2;
}

```

Java String compareTo() Method Example

FileName: CompareToExample.java

```

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

```

Java String compareTo(): empty string

When we compare two strings in which either first or second string is empty, the method returns the length of the string. So, there may be two scenarios:

- If first string is an empty string, the method returns a negative
- If second string is an empty string, the method returns a positive number that is the length of the first string.

FileName: CompareToExample2.java

```

public class CompareToExample2{
public static void main(String args[]){
String s1="hello";
String s2="";
String s3="me";
System.out.println(s1.compareTo(s2));
}

```

```
System.out.println(s2.compareTo(s3));
}}
```

Java String compareTo(): case sensitive

To check whether the compareTo() method considers the case sensitiveness of characters or not, we will make the comparison between two strings that contain the same letters in the same sequence.

Suppose, a string having letters in uppercase, and the second string having the letters in lowercase. On comparing these two string, if the outcome is 0, then the compareTo() method does not consider the case sensitiveness of characters; otherwise, the method considers the case sensitiveness of characters.

FileName: CompareToExample3.java

```
public class CompareToExample3
{
// main method
public static void main(String args[])
{
    // input string in uppercase
    String st1 = new String("INDIA IS MY COUNTRY");

    // input string in lowercase
    String st2 = new String("india is my country");

    System.out.println(st1.compareTo(st2));
}
}
```

Java StringBuffer Class

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the 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
-------------	-------------

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

Important methods of StringBuffer class

Modifier and Type	Method	Description
<code>public synchronized StringBuffer</code>	<code>append(String s)</code>	It 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 synchronized StringBuffer</code>	<code>insert(int offset, String s)</code>	It 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 synchronized StringBuffer</code>	<code>replace(int startIndex, int endIndex, String str)</code>	It is used to replace the string from specified <code>startIndex</code> and <code>endIndex</code> .
<code>public synchronized StringBuffer</code>	<code>delete(int startIndex, int endIndex)</code>	It is used to delete the string from specified <code>startIndex</code> and <code>endIndex</code> .
<code>public synchronized StringBuffer</code>	<code>reverse()</code>	is used to reverse the string.
<code>public int</code>	<code>capacity()</code>	It is used to return the current capacity.
<code>public void</code>	<code>ensureCapacity(int minimumCapacity)</code>	It is used to ensure the capacity at least equal to the given minimum.

public char	charAt(int index)	It is used to return the character at the specified position.
public int	length()	It is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

What is a mutable String?

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

1) StringBuffer Class append() Method

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

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

2) StringBuffer insert() Method

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

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

3) StringBuffer replace() Method

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

```
StringBufferExample3.java
class StringBufferExample3{
public static void main(String args[]){
}
```

```
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJava
}
}
```

4) StringBuffer delete() Method

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

```
StringBufferExample4.java
class StringBufferExample4{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
}
}
```

5) StringBuffer reverse() Method

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

```
StringBufferExample5.java
class StringBufferExample5{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);//prints olleH
}
}
```

6) StringBuffer capacity() Method

The capacity() method of the StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by $(oldCapacity * 2) + 2$. For example if your current capacity is 16, it will be $(16 * 2) + 2 = 34$.

```
StringBufferExample6.java
class StringBufferExample6{
public static void main(String args[]){
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now  $(16 * 2) + 2 = 34$  i.e  $(oldCapacity * 2) + 2$ 
}
}
```

7) StringBuffer ensureCapacity() method

The ensureCapacity() method of the StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by $(oldCapacity * 2) + 2$. For example if your current capacity is 16, it will be $(16 * 2) + 2 = 34$.

StringBufferExample7.java

AD

```
class StringBufferExample7{
public static void main(String args[]){
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
sb.ensureCapacity(10);//now no change
System.out.println(sb.capacity());//now 34
sb.ensureCapacity(50);//now (34*2)+2
System.out.println(sb.capacity());//now 70
}
}
```

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

Important methods of StringBuilder class

Method	Description
public StringBuilder append(String s)	It 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 StringBuilder insert(int offset, String s)	It 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 StringBuilder replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.
public StringBuilder delete(int startIndex, int endIndex)	It is used to delete the string from specified startIndex and endIndex.
public StringBuilder reverse()	It is used to reverse the string.
public int capacity()	It is used to return the current capacity.
public void ensureCapacity(int minimumCapacity)	It is used to ensure the capacity at least equal to the given minimum.
public char charAt(int index)	It is used to return the character at the specified position.
public int length()	It is used to return the length of the string i.e. total number of characters.
public String substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
public String substring(int beginIndex, int endIndex)	It 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.

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

2) StringBuilder insert() method

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

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

3) StringBuilder replace() method

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

```
StringBuilderExample3.java
class StringBuilderExample3{
public static void main(String args[]){
StringBuilder sb=new StringBuilder("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJava
}
}
```

4) StringBuilder delete() method

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

```
StringBuilderExample4.java
class StringBuilderExample4{
public static void main(String args[]){}
```

```
StringBuilder sb=new StringBuilder("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
}
}
```

5) StringBuilder reverse() method

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

StringBuilderExample5.java

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

6) StringBuilder capacity() method

The capacity() method of StringBuilder class returns the current capacity of the Builder. The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by $(oldCapacity * 2) + 2$. For example if your current capacity is 16, it will be $(16 * 2) + 2 = 34$.

StringBuilderExample6.java

```
class StringBuilderExample6{
public static void main(String args[]){
StringBuilder sb=new StringBuilder();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("Java is my favourite language");
System.out.println(sb.capacity());//now  $(16 * 2) + 2 = 34$  i.e  $(oldCapacity * 2) + 2$ 
}
}
```

7) StringBuilder ensureCapacity() method

The ensureCapacity() method of StringBuilder class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by $(oldCapacity * 2) + 2$. For example if your current capacity is 16, it will be $(16 * 2) + 2 = 34$.

StringBuilderExample7.java

```
class StringBuilderExample7{
public static void main(String args[]){
StringBuilder sb=new StringBuilder();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
}
```

```

sb.append("Java is my favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
sb.ensureCapacity(10);//now no change
System.out.println(sb.capacity());//now 34
sb.ensureCapacity(50);//now (34*2)+2
System.out.println(sb.capacity());//now 70
}
}

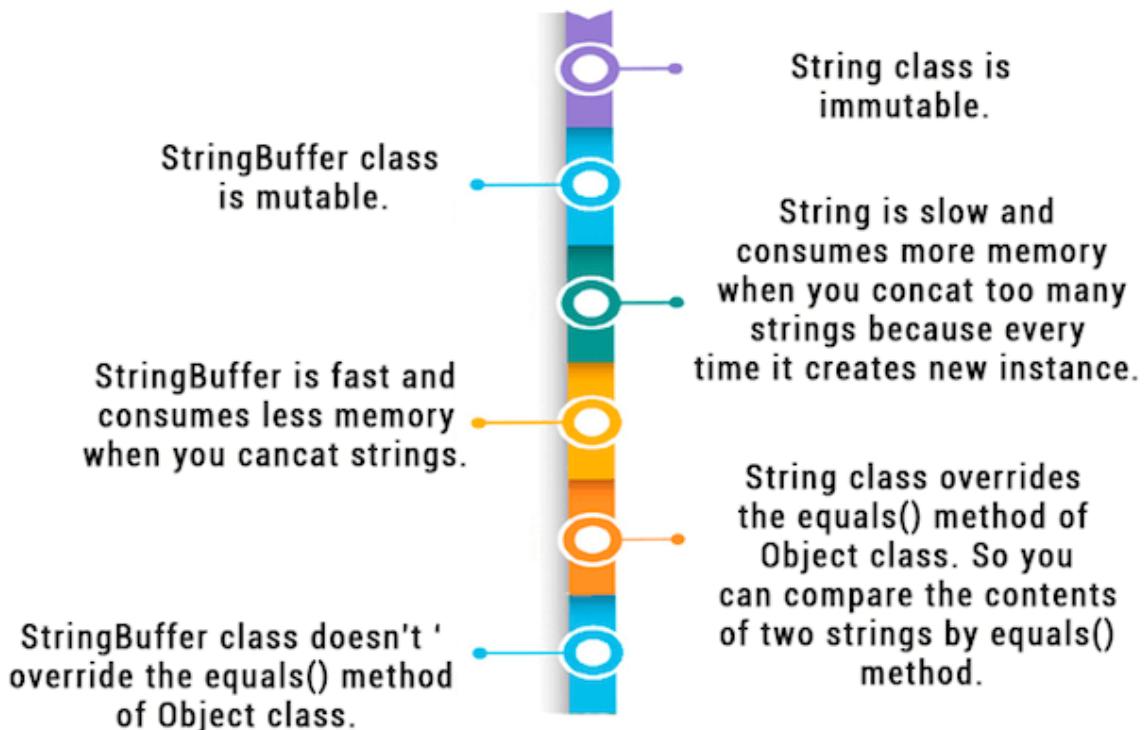
```

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)	The String class is immutable.	The StringBuffer class is mutable.
2)	String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when we concatenate t 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.
4)	String class is slower while performing concatenation operation.	StringBuffer class is faster while performing concatenation operation.
5)	String class uses String constant pool.	StringBuffer uses Heap memory

StringBuffer vs String



Performance Test of String and StringBuffer

```
ConcatTest.java
public class ConcatTest{
    public static String concatWithString() {
        String t = "Java";
        for (int i=0; i<10000; i++){
            t = t + "Tpoint";
        }
        return t;
    }
    public static String concatWithStringBuffer(){
        StringBuffer sb = new StringBuffer("Java");
        for (int i=0; i<10000; i++){
            sb.append("Tpoint");
        }
        return sb.toString();
    }
    public static void main(String[] args){
        long startTime = System.currentTimeMillis();
        concatWithString();
        System.out.println("Time taken by Concating with String:
"+(System.currentTimeMillis()-startTime)+"ms");
        startTime = System.currentTimeMillis();
    }
}
```

```

        concatWithStringBuffer();
        System.out.println("Time taken by Concating with StringBuffer:
"+(System.currentTimeMillis()-startTime)+"ms");
    }
}

```

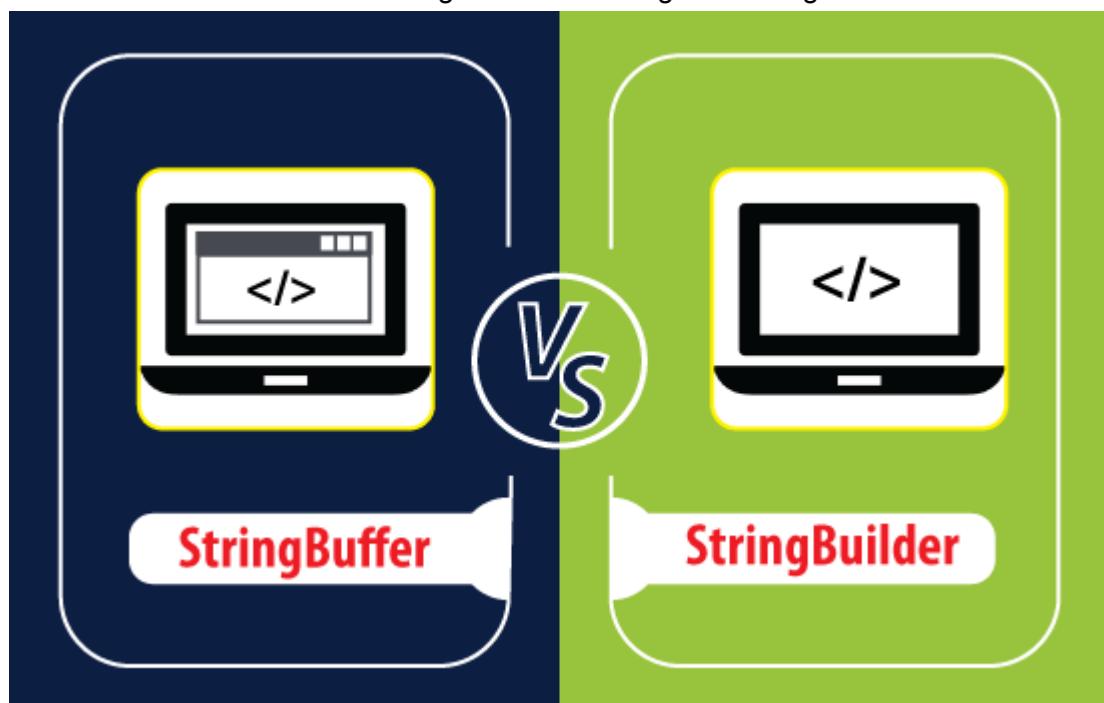
Output:

Time taken by Concating with String: 578ms
 Time taken by Concating with StringBuffer: 0ms

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 is given below:



No.	StringBuffer	StringBuilder
1)	StringBuffer is synchronized i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.

2)	StringBuffer is less efficient than StringBuilder.	StringBuilder is more efficient than StringBuffer.
3)	StringBuffer was introduced in Java 1.0	StringBuilder was introduced in Java 1.5

Exception Handling in Java

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

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 need to handle exceptions. Let's consider a scenario:

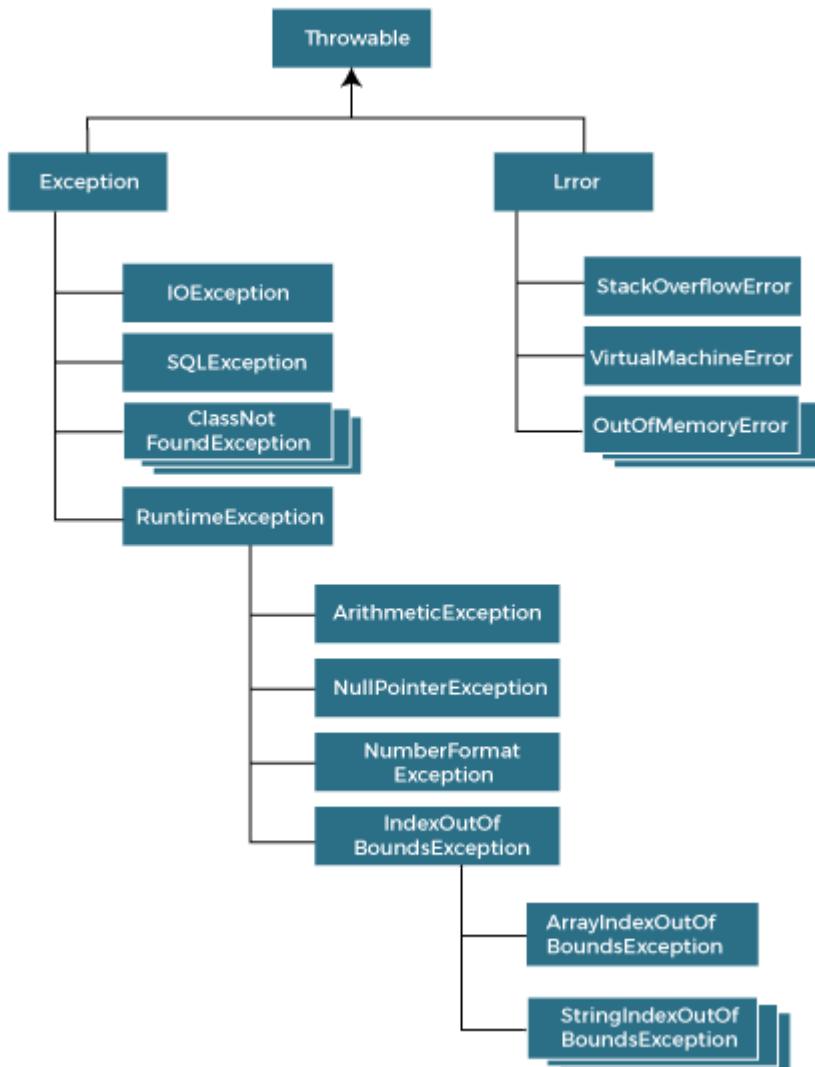
```

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

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

Hierarchy of Java Exception classes

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



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error



Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
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 a finally block later.
finally	The "finally" block is used to execute the necessary 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 specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signatures.

Java Exception Handling Example

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

JavaExceptionExample.java

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(ArithmaticException e){System.out.println(e);}
7. //rest code of the program
8. System.out.println("rest of the code... ");
9. }
10.}
```

Output:

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

In the above example, 100/0 raises an ArithmaticException 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 ArithmaticException occurs

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

```
1. int a=50/0;//ArithmaticException
```

2) A scenario where NullPointerException occurs

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

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

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

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to its size, the ArrayIndexOutOfBoundsException occurs. There may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

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 in the 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

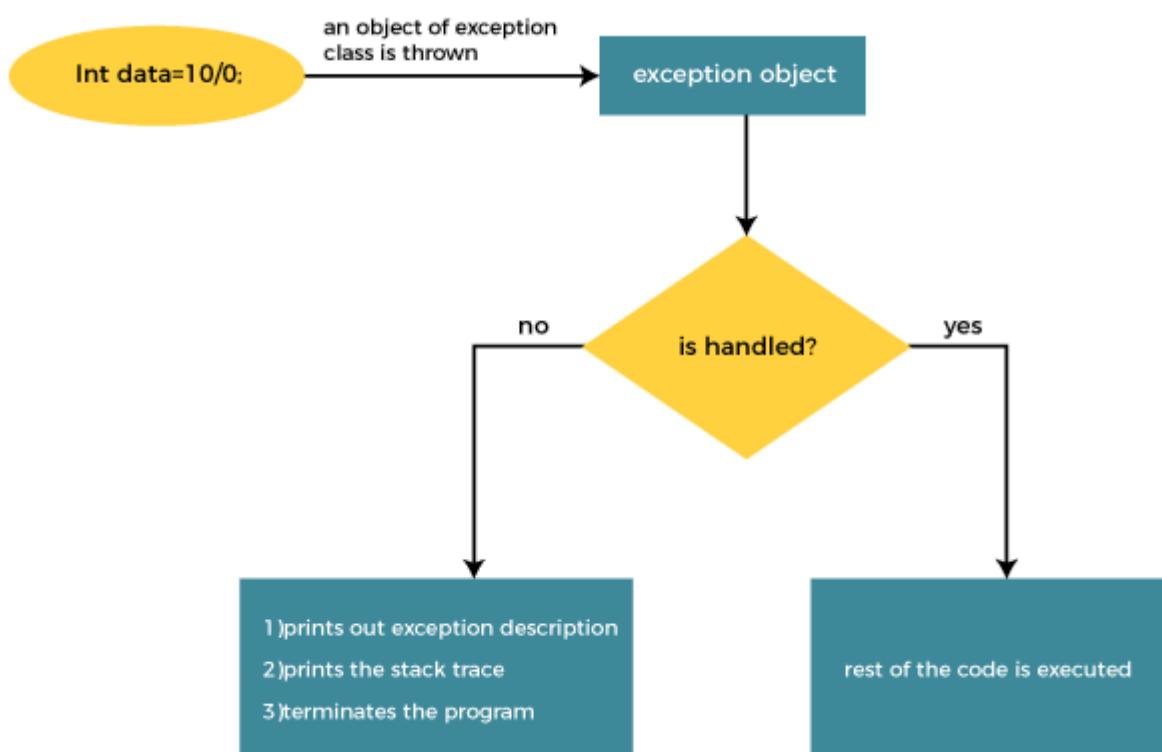
1. try{
2. //code that may throw an exception
3. }finally{}

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.

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 the application programmer handles the exception, the normal flow of the application is maintained, i.e., the rest of the code is executed.

Problem without exception handling

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

Example 1

TryCatchExample1.java

```
public class TryCatchExample1 {  
  
    public static void main(String[] args) {  
  
        int data=50/0; //may throw exception  
  
        System.out.println("rest of the code");  
  
    }  
  
}
```

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 might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

Solution by exception handling

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

Example 2

TryCatchExample2.java

```
public class TryCatchExample2 {  
  
    public static void main(String[] args) {  
  
        try  
  
        {  
  
            int data=50/0; //may throw exception  
  
        }  
  
        //handling the exception  
  
        catch(ArithmetricException e)  
  
        {  
  
            System.out.println(e);  
  
        }  
  
        System.out.println("rest of the code");  
  
    }  
  
}
```

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.

TryCatchExample3.java

```
public class TryCatchExample3 {  
  
    public static void main(String[] args) {  
  
        try  
  
        {  
  
            int data=50/0; //may throw exception  
  
            // if exception occurs, the remaining statement will not execute  
  
            System.out.println("rest of the code");  
  
        }  
  
        // handling the exception  
  
        catch(ArithmeticException e)  
  
        {  
  
            System.out.println(e);  
  
        }  
  
    }  
  
}
```

Output:

```
java.lang.ArithmetricException: / 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.

TryCatchExample4.java

```
public class TryCatchExample4 {  
  
    public static void main(String[] args) {  
  
        try  
  
        {  
  
            int data=50/0; //may throw exception  
  
        }  
  
        // handling the exception by using Exception class  
  
        catch(Exception e)  
  
        {  
  
            System.out.println(e);  
  
        }  
  
        System.out.println("rest of the code");  
  
    }  
}
```

```
}
```

Output:

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

rest of the code

Example 5

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

TryCatchExample5.java

```
public class TryCatchExample5 {  
  
    public static void main(String[] args) {  
  
        try {  
  
            int data=50/0; //may throw exception  
  
        }  
  
        // handling the exception  
  
        catch(Exception e) {  
  
            // displaying the custom message  
  
            System.out.println("Can't divided by zero");  
  
        }  
    }  
}
```

```
}
```

Output:

Can't divided by zero

Java Catch Multiple Exceptions

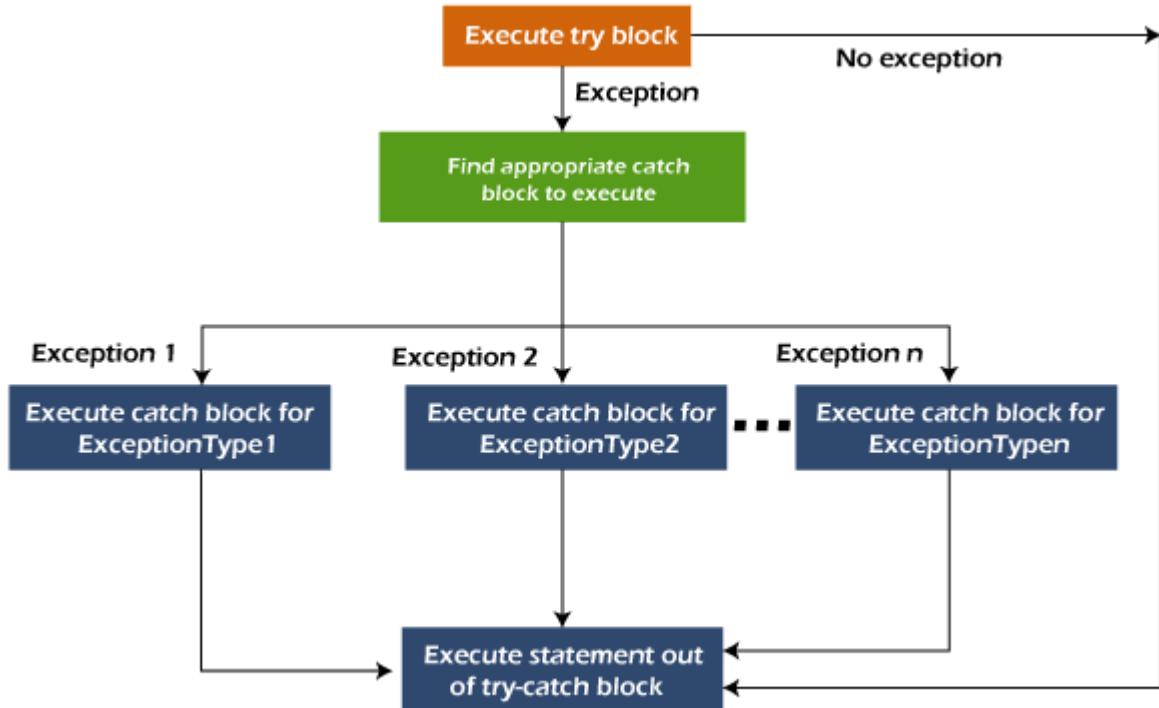
Java Multi-catch block

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 `ArithmaticException` must come before catch for `Exception`.

Flowchart of Multi-catch Block



Example 1

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

MultipleCatchBlock1.java

```

public class MultipleCatchBlock1 {

    public static void main(String[] args) {

        try{
            int a[] = new int[5];
            a[5] = 30/0;
        }

        catch(ArithmeticException e)
    }
}
  
```

```

{
    System.out.println("Arithmetic Exception occurs");

}

catch(ArrayIndexOutOfBoundsException e)

{
    System.out.println("ArrayIndexOutOfBoundsException Exception occurs");

}

catch(Exception e)

{
    System.out.println("Parent Exception occurs");

}

System.out.println("rest of the code");

}
}


```

Output:

Arithmetic Exception occurs
rest of the code

Example 2

MultipleCatchBlock2.java

```
public class MultipleCatchBlock2 {
```

```
public static void main(String[] args) {  
  
    try{  
  
        int a[]={};  
  
        System.out.println(a[10]);  
  
    }  
  
    catch(ArithmetricException e)  
  
    {  
  
        System.out.println("Arithmetric Exception occurs");  
  
    }  
  
    catch(ArrayIndexOutOfBoundsException e)  
  
    {  
  
        System.out.println("ArrayIndexOutOfBoundsException occurs");  
  
    }  
  
    catch(Exception e)  
  
    {  
  
        System.out.println("Parent Exception occurs");  
  
    }  
  
    System.out.println("rest of the code");  
  
}
```

Output:

ArrayIndexOutOfBoundsException occurs

rest of the code

Java Nested try block

In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter in a statement in the try block, context of that exception is pushed onto the stack.

For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

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:

```
....  
//main try block  
try  
{  
    statement 1;  
    statement 2;  
    //try catch block within another try block  
try  
{
```

```
statement 3;  
statement 4;  
//try catch block within nested try block  
try  
{  
    statement 5;  
    statement 6;  
}  
catch(Exception e2)  
{  
    //exception message  
}  
  
}  
  
catch(Exception e1)  
{  
    //exception message  
}  
  
}  
  
//catch block of parent (outer) try block  
catch(Exception e3)  
{
```

```
//exception message
```

```
}
```

```
....
```

Java Nested try Example

Example 1

Let's see an example where we place a try block within another try block for two different exceptions.

NestedTryBlock.java

```
public class NestedTryBlock{  
  
    public static void main(String args[]){  
  
        //outer try block  
  
        try{  
  
            //inner try block 1  
  
            try{  
  
                System.out.println("going to divide by 0");  
  
                int b =39/0;  
  
            }  
  
            //catch block of inner try block 1  
  
            catch(ArithmeticException e)  
  
            {  
  
                System.out.println(e);  
  
            }  
    }  
}
```

```
//inner try block 2

try{
    int a[]=new int[5];
}

//assigning the value out of array bounds
a[5]=4;
}

//catch block of inner try block 2
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println(e);
}

System.out.println("other statement");
}

//catch block of outer try block
catch(Exception e)
{
```

```

        System.out.println("handled the exception (outer catch)");

    }

System.out.println("normal flow..");

}

}

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock.java
C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock
going to divide by 0
java.lang.ArithmetricException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
other statement
normal flow..

```

When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

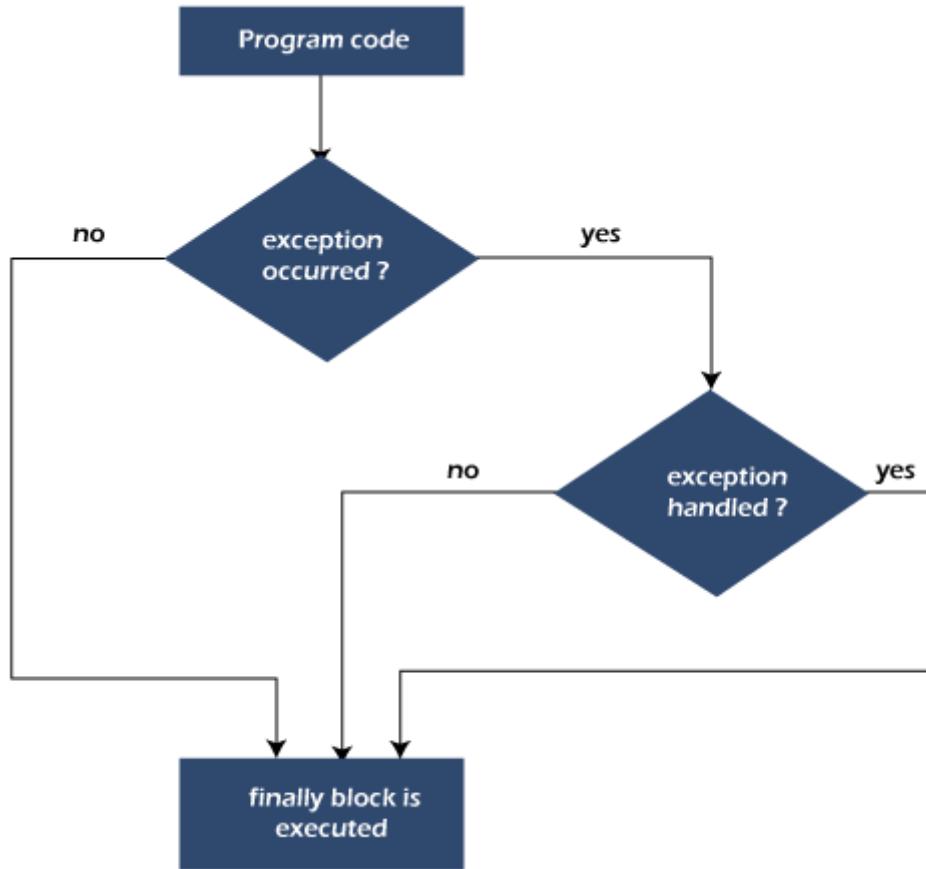
Java finally block

Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

Flowchart of finally block



Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

Usage of Java finally

Let's see the different cases where Java finally block can be used.

Case 1: When an exception does not occur

Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

TestFinallyBlock.java

```
class TestFinallyBlock {  
  
    public static void main(String args[]){  
  
        try{  
  
            //below code do not throw any exception  
  
            int data=25/5;  
  
            System.out.println(data);  
  
        }  
  
        //catch won't be executed  
  
        catch(NullPointerException e){  
  
            System.out.println(e);  
  
        }  
  
        //executed regardless of exception occurred or not  
  
        finally {  
  
            System.out.println("finally block is always executed");  
  
        }  
  
        System.out.println("rest of phe code...");  
  
    }  
  
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock.java
C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock
5
finally block is always executed
rest of the code...
```

Case 2: When an exception occur but not handled by the catch block

Let's see the following example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

TestFinallyBlock1.java

```
public class TestFinallyBlock1{

    public static void main(String args[]){
        try {
            System.out.println("Inside the try block");

            //below code throws divide by zero exception
            int data=25/0;
            System.out.println(data);
        }

        //cannot handle Arithmetic type exception
        //can only accept Null Pointer type exception
    }
}
```

```

catch(NullPointerException e){

    System.out.println(e);

}

//executes regardless of exception occurred or not

finally {

    System.out.println("finally block is always executed");

}

System.out.println("rest of the code...");

}

}

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock1.java
C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock1
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmetricException: / by zero
        at TestFinallyBlock1.main(TestFinallyBlock1.java:9)

```

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 the program exits (either by calling `System.exit()` or by causing a fatal error that causes the process to abort).

Java throw Exception

In Java, exceptions allow us to write good quality code where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

Java throw keyword

The Java throw keyword is used to throw an exception explicitly.

We specify the exception object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throwing a keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw an ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw an exception using the throw keyword.

The syntax of the Java throw keyword is given below.

throw Instance i.e.,

1. `throw new exception_class("error message");`

Let's see the example of throw IOException.

1. `throw new IOException("sorry device error");`

Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

Java throw keyword Example

Example 1: Throwing Unchecked Exception

In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

TestThrow1.java

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.

```
public class TestThrow1 {  
    //function to check if person is eligible to vote or not  
    public static void validate(int age) {  
        if(age<18) {  
            //throw Arithmetic exception if not eligible to vote  
            throw new ArithmeticException("Person is not eligible to vote");  
        }  
        else {  
            System.out.println("Person is eligible to vote!!");  
        }  
    }  
    //main method  
    public static void main(String args[]){  
        //calling the function  
        validate(13);  
        System.out.println("rest of the code...");  
    }  
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java  
C:\Users\Anurati\Desktop\abcDemo>java TestThrow1  
Exception in thread "main" java.lang.ArithmetricException: Person is not eligible to  
vote  
    at TestThrow1.validate(TestThrow1.java:8)  
    at TestThrow1.main(TestThrow1.java:18)
```

The above code throw an unchecked exception. Similarly, we can also throw unchecked and user defined exceptions.

Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

Example 2: Throwing Checked Exception

Note: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.

TestThrow2.java

```
import java.io.*;  
  
public class TestThrow2 {  
  
    //function to check if person is eligible to vote or not  
  
    public static void method() throws FileNotFoundException {  
  
        FileReader file = new  
FileReader("C:\\\\Users\\\\Anurati\\\\Desktop\\\\abc.txt");  
  
        BufferedReader fileInput = new BufferedReader(file);  
  
        throw new FileNotFoundException();  
    }  
}
```

```

}

//main method

public static void main(String args[]){
    try
    {
        method();
    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }
    System.out.println("rest of the code...");
}
}

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac TestThrow2.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrow2
java.io.FileNotFoundException
        at TestThrow2.method(TestThrow2.java:12)
        at TestThrow2.main(TestThrow2.java:22)
rest of the code...

```

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.

Note: By default Unchecked Exceptions are forwarded in calling chain (propagated).

Exception Propagation Example

TestExceptionPropagation1.java

```
class TestExceptionPropagation1{

    void m(){
        int data=50/0;

    }

    void n(){
        m();
    }

    void p(){
        try{
            n();
        }catch(Exception e){System.out.println("exception handled");}
    }

    public static void main(String args[]){
        TestExceptionPropagation1 obj=new TestExceptionPropagation1();
        obj.p();
    }
}
```

```
        System.out.println("normal flow...");  
    }  
}
```

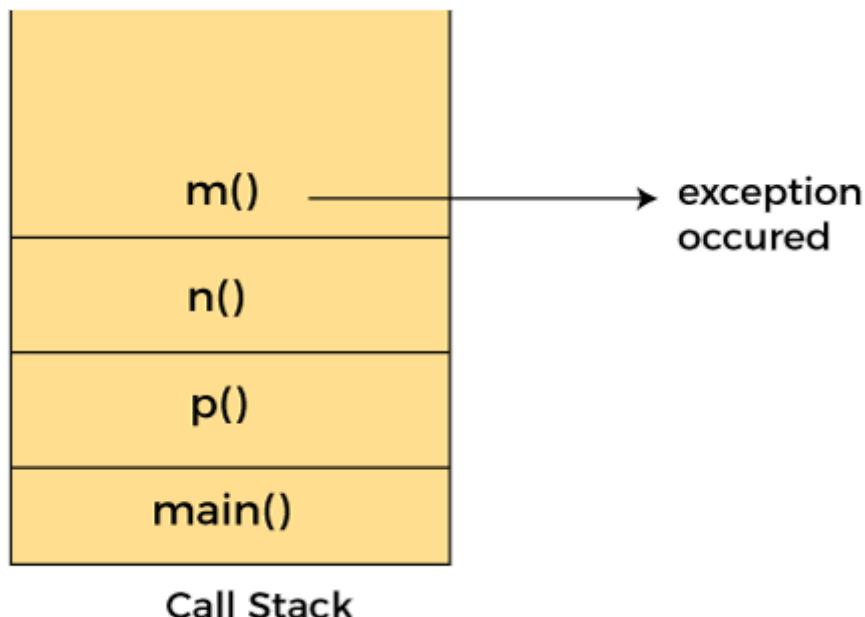
Output:

exception handled

normal flow...

In the above example exception occurs in the m() method where it is not handled, so it is propagated to the previous n() method where it is not handled, again it is propagated to the p() method where exception is handled.

Exception can be handled in any method in call stack either in the main() method, p() method, n() method or m() method.



Note: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Exception Propagation Example

TestExceptionPropagation1.java

```
class TestExceptionPropagation2{

void m(){

    throw new java.io.IOException("device error");//checked exception

}

void n(){

    m();

}

void p(){

try{

    n();

}catch(Exception e){System.out.println("exception handled");}

}

public static void main(String args[]){

TestExceptionPropagation2 obj=new TestExceptionPropagation2();

obj.p();

System.out.println("normal flow");

}

}
```

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 the normal flow of the program 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 checking the code before it 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 our control so we can correct our code.
- **error:** beyond our control. For example, we 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.

Testthrows1.java

```
import java.io.IOException;
```

```

class Testthrows1{

    void m()throws IOException{
        throw new IOException("device error");//checked exception
    }

    void n()throws IOException{
        m();
    }

    void p(){
        try{
            n();
        }catch(Exception e){System.out.println("exception handled");}
    }

    public static void main(String args[]){
        Testthrows1 obj=new Testthrows1();
        obj.p();
        System.out.println("normal flow...");
    }
}

```

Output:

exception handled
normal flow...

Rule: If we are calling a method that declares an exception, we must either caught or declare the exception.

here are two cases:

1. Case 1: We have caught the exception i.e. we have handled the exception using try/catch block.
2. Case 2: We have declared the exception i.e. specified throws keyword with the method.

Case 1: Handle Exception Using try-catch block

In case we handle the exception, the code will be executed fine whether exception occurs during the program or not.

Testthrows2.java

```
import java.io.*;  
  
class M{  
  
    void method()throws IOException{  
  
        throw new IOException("device error");  
  
    }  
  
}  
  
public class Testthrows2{  
  
    public static void main(String args[]){  
  
        try{  
  
            M m=new M();  
  
            m.method();  
  
        }catch(Exception e){System.out.println("exception handled");}  
  
        System.out.println("normal flow...");  
    }  
}
```

```
}
```

```
}
```

Output:

exception handled
normal flow...

Case 2: Declare Exception

- In case we declare the exception, if exception does not occur, the code will be executed fine.
- In case we declare the exception and the exception occurs, it will be thrown at runtime because throws does not handle the exception.

Let's see examples for both the scenario.

A) If exception does not occur

Testthrows3.java

```
import java.io.*;  
  
class M{  
  
    void method()throws IOException{  
  
        System.out.println("device operation performed");  
  
    }  
  
}  
  
class Testthrows3{  
  
    public static void main(String args[])throws IOException{//declare  
exception  
  
    M m=new M();  
  
    m.method();
```

```
    System.out.println("normal flow...");  
}  
}
```

Output:

device operation performed
normal flow..

B) If exception occurs

Testthrows4.java

```
import java.io.*;  
  
class M{  
  
    void method() throws IOException{  
  
        throw new IOException("device error");  
    }  
}  
  
class Testthrows4{  
  
    public static void main(String args[]) throws IOException{//declare  
exception  
  
    M m=new M();  
  
    m.method();  
  
    System.out.println("normal flow...");  
}  
}
```

Output:

```
Exception in thread "main" java.io.IOException: device error
at M.method(Testthrows4.java:4)
at Testthrows4.main(Testthrows4.java:10)
```

Que) Can we rethrow an exception?

Yes, by throwing same exception in catch block.

Java Custom Exception

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Consider the example 1 in which InvalidAgeException class extends the Exception class.

Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e. Exception class that can be obtained using getMessage() method on the object we have created.

```
public class WrongFileNameException extends Exception {

    public WrongFileNameException(String errorMessage) {
        super(errorMessage);
    }
}
```

Note: We need to write the constructor that takes the String as the error message and it is called parent class constructor.

Example 1:

Let's see a simple example of Java custom exception. In the following code, constructor of InvalidAgeException takes a string as an argument. This string is passed to constructor of parent class Exception using the super() method. Also the constructor of Exception class can be called without using a parameter and calling super() method is not mandatory.

TestCustomException1.java

```
// class representing custom exception
class InvalidAgeException extends Exception
{
    public InvalidAgeException (String str)
    {
        // calling the constructor of parent Exception
        super(str);
    }
}

// class that uses custom exception InvalidAgeException
public class TestCustomException1
{

    // method to check the age
    static void validate (int age) throws InvalidAgeException{
        if(age < 18){

            // throw an object of user defined exception
            throw new InvalidAgeException("age is not valid to vote");
        }
        else {
            System.out.println("welcome to vote");
        }
    }

    // main method
    public static void main(String args[])
    {
        try
        {
```

```
// calling the method  
validate(13);  
}  
catch (InvalidAgeException ex)  
{  
    System.out.println("Caught the exception");  
  
    // printing the message from InvalidAgeException object  
    System.out.println("Exception occurred: " + ex);  
}  
  
System.out.println("rest of the code...");  
}  
}
```

Multithreading in Java

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

- 1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
- 2) You **can perform many operations together, so it saves time.**

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

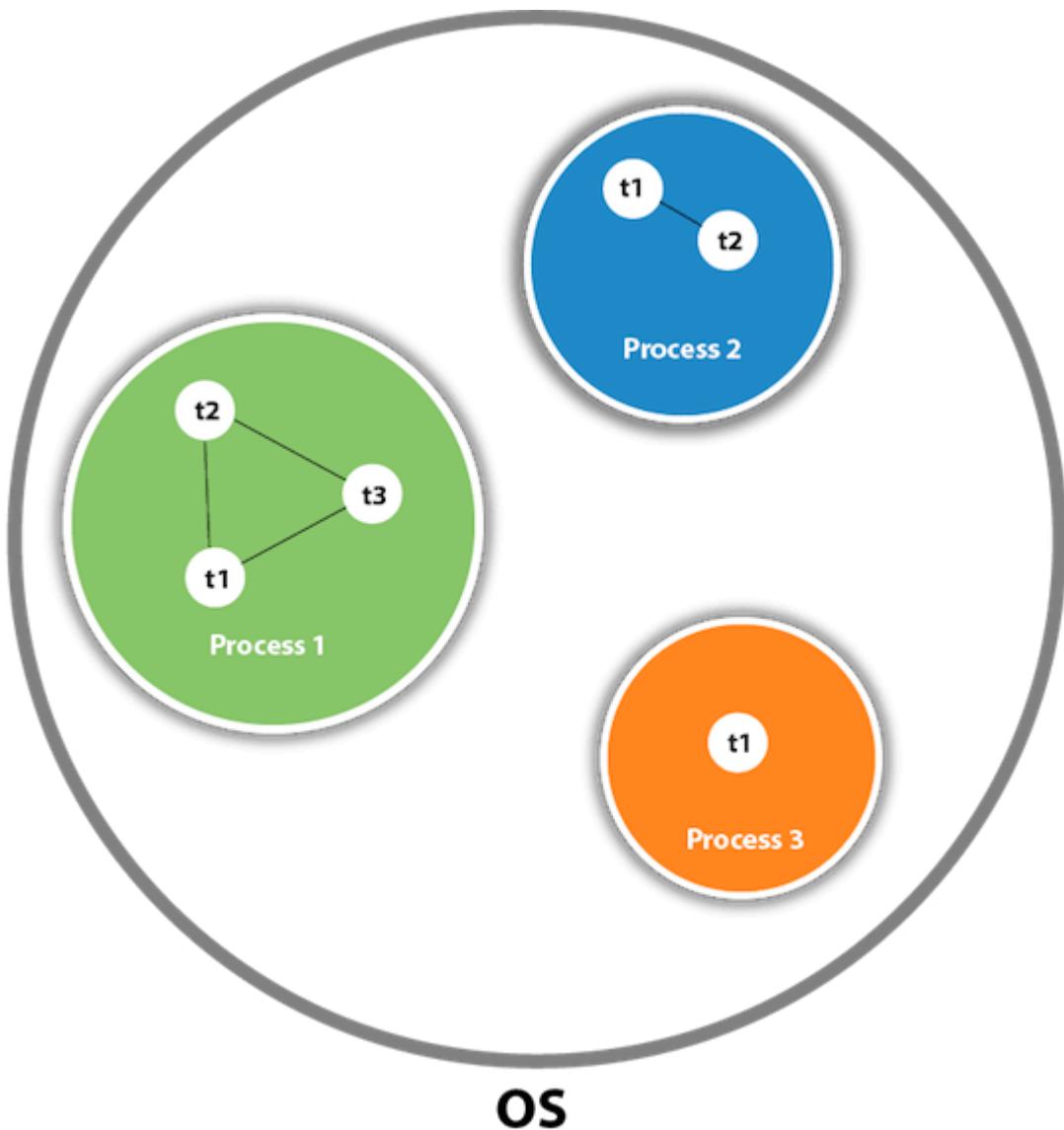
- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Note: At least one process is required for each thread.

What is Thread in java

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

Java Thread class

Java provides **Thread class** to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

Explanation of Different Thread States

New: Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

Active: When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **Runnable**, and the other is **running**.

- **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.
A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.
- **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

Blocked or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

For example, a thread (let's say its name is A) may want to print some data from the printer. However, at the same time, the other thread (let's say its name is B) is using the printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus, thread A is in the blocked state. A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU). Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state.

When the main thread invokes the `join()` method then, it is said that the main thread is in the waiting state. The main thread then waits for the child threads to complete their tasks. When the child threads complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.

If there are a lot of threads in the waiting or blocked state, then it is the duty of the thread scheduler to determine which thread to choose and which one to reject, and the chosen thread is then given the opportunity to run.

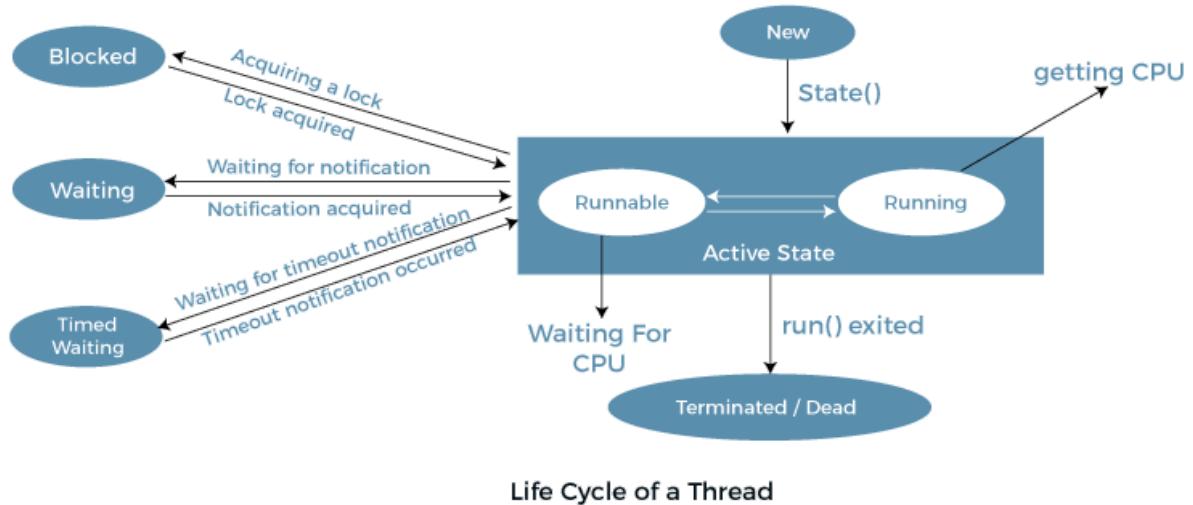
Timed Waiting: Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the `sleep()` method on a specific thread. The `sleep()` method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

Terminated: A thread reaches the termination state because of the following reasons:

- When a thread has finished its job, then it exists or terminates normally.
- **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

The following diagram shows the different states involved in the life cycle of a thread.



Implementation of Thread States

In Java, one can get the current state of a thread using the **Thread.getState()** method. The **java.lang.Thread.State** class of Java provides the constants ENUM to represent the state of a thread. These constants are:

1. **public static final Thread.State NEW**

It represents the first state of a thread that is the NEW state.

1. **public static final Thread.State RUNNABLE**

It represents the runnable state. It means a thread is waiting in the queue to run.

1. **public static final Thread.State BLOCKED**

It represents the blocked state. In this state, the thread is waiting to acquire a lock.

1. **public static final Thread.State WAITING**

It represents the waiting state. A thread will go to this state when it invokes the `Object.wait()` method, or `Thread.join()` method with no timeout. A thread in the waiting state is waiting for another thread to complete its task.

1. **public static final Thread.State TIMED_WAITING**

It represents the timed waiting state. The main difference between waiting and timed waiting is the time constraint. Waiting has no time constraint, whereas timed waiting has the time constraint. A thread invoking the following method reaches the timed waiting state.

- sleep
- join with timeout
- wait with timeout
- parkUntil
- parkNanos

1. **public static final** Thread.State TERMINATED

It represents the final state of a thread that is terminated or dead. A terminated thread means it has completed its execution.

Java Program for Demonstrating Thread States

The following Java program shows some of the states of a thread defined above.

FileName: ThreadState.java

```
// ABC class implements the interface Runnable

class ABC implements Runnable

{

    public void run()

    {

        // try-catch block

        try

        {

            // moving thread t2 to the state timed waiting
        }
    }
}
```

```
    Thread.sleep(100);

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}
```

System.out.println("The state of thread t1 while it invoked the method join()
on thread t2 -"+ ThreadState.t1.getState());

```
// try-catch block

try

{

    Thread.sleep(200);

}

catch (InterruptedException ie)

{

    ie.printStackTrace();

}

}
```

```
// ThreadState class implements the interface Runnable

public class ThreadState implements Runnable

{

    public static Thread t1;

    public static ThreadState obj;

    // main method

    public static void main(String args[])
    {

        // creating an object of the class ThreadState

        obj = new ThreadState();

        t1 = new Thread(obj);

        // thread t1 is spawned

        // The thread t1 is currently in the NEW state.

        System.out.println("The state of thread t1 after spawning it - " + t1.getState());

        // invoking the start() method on

        // the thread t1

        t1.start();
    }
}
```

```
// thread t1 is moved to the Runnable state  
  
System.out.println("The state of thread t1 after invoking the method start() on  
it - " + t1.getState());  
  
}
```

```
public void run()  
{  
  
ABC myObj = new ABC();  
  
Thread t2 = new Thread(myObj);
```

// thread t2 is created and is currently in the NEW state.

```
System.out.println("The state of thread t2 after spawning it - " + t2.getState());  
  
t2.start();
```

// thread t2 is moved to the runnable state

```
System.out.println("the state of thread t2 after calling the method start() on it -  
" + t2.getState());
```

// try-catch block for the smooth flow of the program

```
try  
{  
  
// moving the thread t1 to the state timed waiting  
  
Thread.sleep(200);
```

```
}

catch (InterruptedException ie)

{

ie.printStackTrace();

}
```

```
System.out.println("The state of thread t2 after invoking the method sleep()
on it - "+ t2.getState() );
```

```
// try-catch block for the smooth flow of the program
```

```
try

{

// waiting for thread t2 to complete its execution

t2.join();

}
```

```
catch (InterruptedException ie)
```

```
{

ie.printStackTrace();

}
```

```
System.out.println("The state of thread t2 when it has completed it's
execution - " + t2.getState());
```

```
}
```

```
}
```

The state of thread t1 after spawning it - NEW

The state of thread t1 after invoking the method start() on it - RUNNABLE

The state of thread t2 after spawning it - NEW

the state of thread t2 after calling the method start() on it - RUNNABLE

The state of thread t1 while it invoked the method join() on thread t2

-TIMED_WAITING

The state of thread t2 after invoking the method sleep() on it - TIMED_WAITING

The state of thread t2 when it has completed its execution - TERMINATED

Explanation: Whenever we spawn a new thread, that thread attains the new state. When the method start() is invoked on a thread, the thread scheduler moves that thread to the runnable state. Whenever the join() method is invoked on any thread instance, the current thread executing that statement has to wait for this thread to finish its execution, i.e., move that thread to the terminated state. Therefore, before the final print statement is printed on the console, the program invokes the method join() on thread t2, making the thread t1 wait while the thread t2 finishes its execution and thus, the thread t2 gets to the terminated or dead state. Thread t1 goes to the waiting state because it is waiting for thread t2 to finish its execution as it has invoked the method join() on thread t2.

Java Threads | How to create a thread in Java

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- **Thread()**
- **Thread(String name)**
- **Thread(Runnable r)**
- **Thread(Runnable r, String name)**

Commonly used methods of Thread class:

- 1. public void run():** is used to perform action for a thread.
- 2. public void start():** starts the execution of the thread. JVM calls the run() method on the thread.
- 3. public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- 4. public void join():** waits for a thread to die.
- 5. public void join(long miliseconds):** waits for a thread to die for the specified milliseconds.
- 6. public int getPriority():** returns the priority of the thread.
- 7. public int setPriority(int priority):** changes the priority of the thread.
- 8. public String getName():** returns the name of the thread.
- 9. public void setName(String name):** changes the name of the thread.
- 10. public Thread currentThread():** returns the reference of currently executing thread.
- 11. public int getId():** returns the id of the thread.
- 12. public Thread.State getState():** returns the state of the thread.
- 13. public boolean isAlive():** tests if the thread is alive.
- 14. public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
- 15. public void suspend():** is used to suspend the thread(deprecated).

- 16. public void resume():** is used to resume the suspended thread(deprecated).
- 17. public void stop():** is used to stop the thread(deprecated).
- 18. public boolean isDaemon():** tests if the thread is a daemon thread.
- 19. public void setDaemon(boolean b):** marks the thread as daemon or user thread.
- 20. public void interrupt():** interrupts the thread.
- 21. public boolean isInterrupted():** tests if the thread has been interrupted.
- 22. public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

- 1. public void run():** is used to perform action for a thread.

Starting a thread:

The start() method of Thread class is used to start a newly created thread. It performs the following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

1) Java Thread Example by extending Thread class

FileName: Multi.java

```
class Multi extends Thread{  
    public void run(){  
}
```

```
System.out.println("thread is running...");  
}  
  
public static void main(String args[]){  
    Multi t1=new Multi();  
    t1.start();  
}
```

Output:

thread is running...

2) Java Thread Example by implementing Runnable interface

FileName: Multi3.java

```
class Multi3 implements Runnable{  
    public void run(){  
        System.out.println("thread is running...");  
    }  
  
    public static void main(String args[]){  
        Multi3 m1=new Multi3();  
        Thread t1 =new Thread(m1); // Using the constructor Thread(Runnable r)  
        t1.start();  
    }  
}
```

3) Using the Thread Class: Thread(String Name)

We can directly use the Thread class to spawn new threads using the constructors defined above.

FileName: MyThread1.java

```
public class MyThread1

{

    // Main method

    public static void main(String args[])
    {

        // creating an object of the Thread class using the constructor Thread(String
        name)

        Thread t= new Thread("My first thread");



        // the start() method moves the thread to the active state

        t.start();

        // getting the thread name by invoking the getName() method

        String str = t.getName();

        System.out.println(str);

    }

}
```

Thread Scheduler in Java

A component of Java that decides which thread to run or execute and which thread to wait is called a thread scheduler in Java. In Java, a thread is only chosen by a thread scheduler if it is in the runnable state. However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones. There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. Priority and Time of arrival.

Priority: Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

Time of Arrival: Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, arrival time of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

Thread Scheduler Algorithms

On the basis of the above-mentioned factors, the scheduling algorithm is followed by a Java thread scheduler.

First Come First Serve Scheduling:

In this scheduling algorithm, the scheduler picks the threads that arrive first in the runnable queue. Observe the following table:

Threads	Time of Arrival
t1	0
t2	1
t3	2
t4	3

In the above table, we can see that Thread t1 has arrived first, then Thread t2, then t3, and at last t4, and the order in which the threads will be processed is according to the time of arrival of threads.

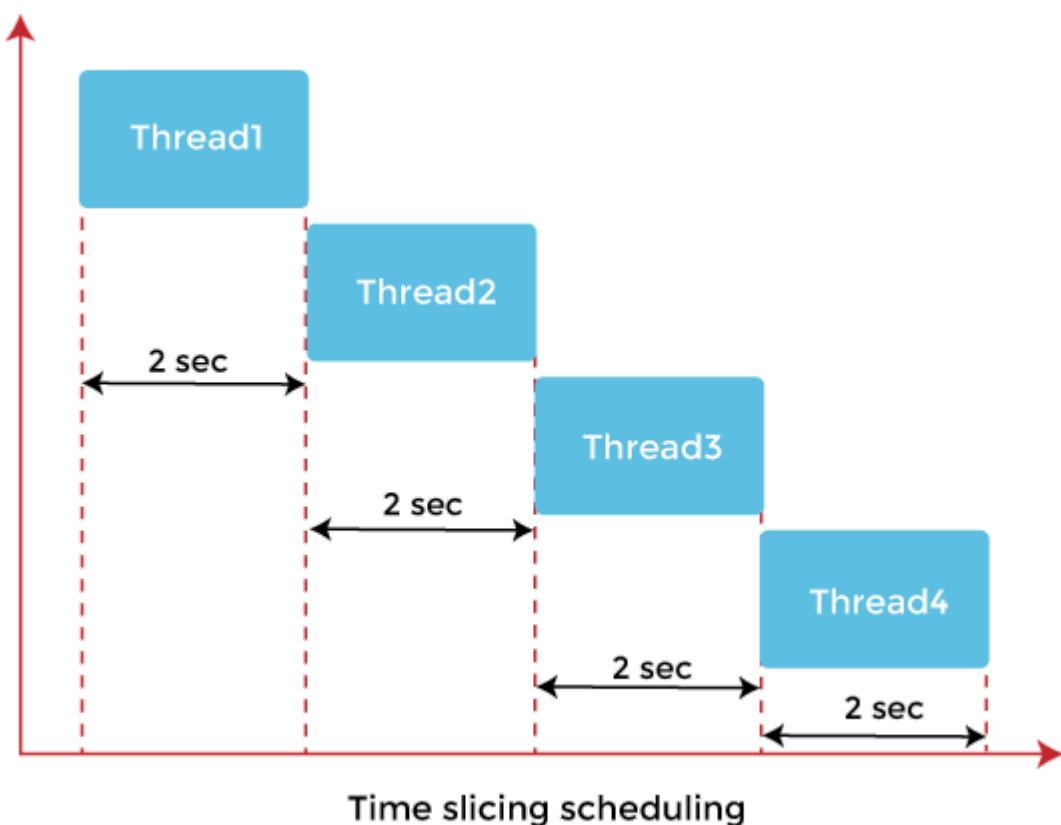


First Come First Serve Scheduling

Hence, Thread t_1 will be processed first, and Thread t_4 will be processed last.

Time-slicing scheduling:

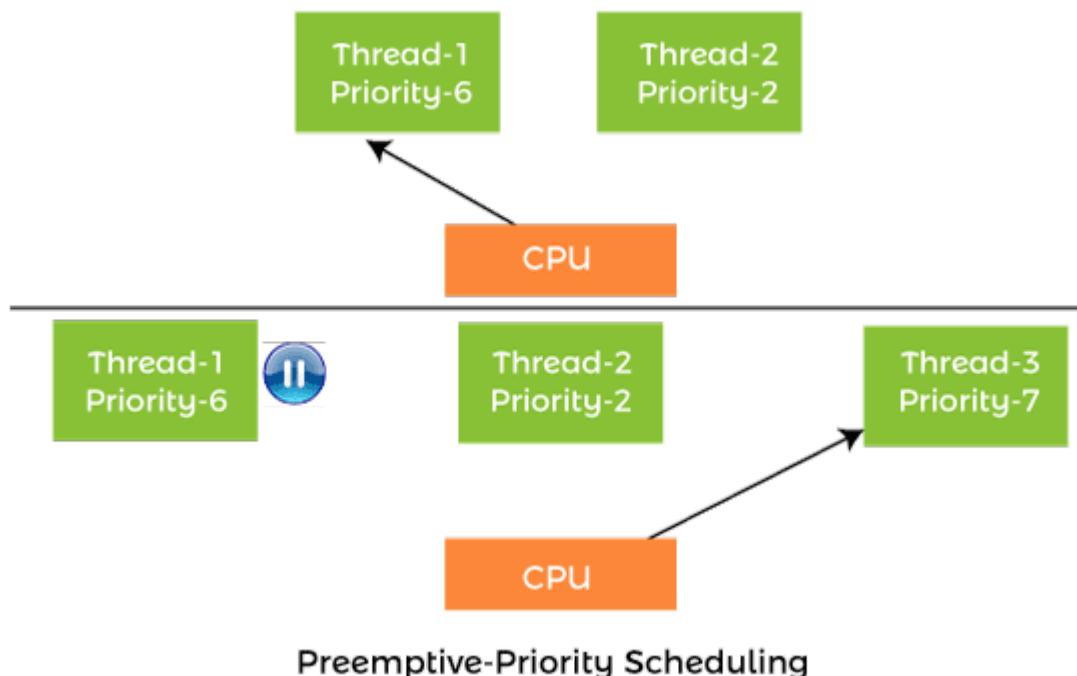
Usually, the First Come First Serve algorithm is non-preemptive, which is bad as it may lead to infinite blocking (also known as starvation). To avoid that, some time-slices are provided to the threads so that after some time, the running thread has to give up the CPU. Thus, the other waiting threads also get time to run their job.



In the above diagram, each thread is given a time slice of 2 seconds. Thus, after 2 seconds, the first thread leaves the CPU, and the CPU is then captured by Thread2. The same process repeats for the other threads too.

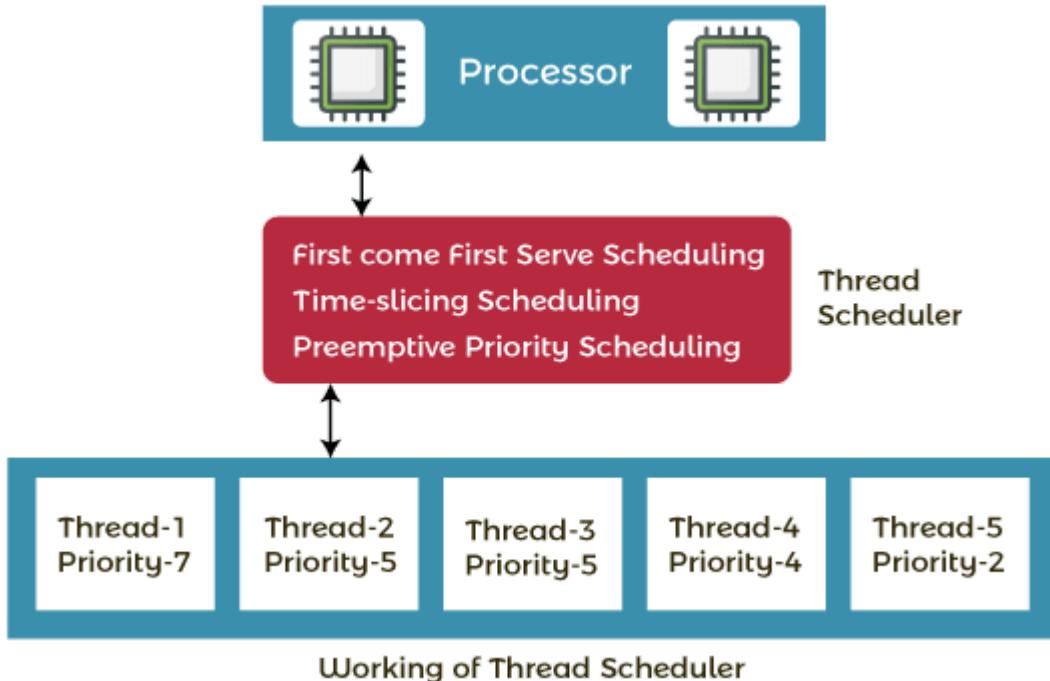
Preemptive-Priority Scheduling:

The name of the scheduling algorithm denotes that the algorithm is related to the priority of the threads.



Suppose there are multiple threads available in the runnable state. The thread scheduler picks that thread that has the highest priority. Since the algorithm is also preemptive, therefore, time slices are also provided to the threads to avoid starvation. Thus, after some time, even if the highest priority thread has not completed its job, it has to release the CPU because of preemption.

Working of the Java Thread Scheduler



Let's understand the working of the Java thread scheduler. Suppose, there are five threads that have different arrival times and different priorities. Now, it is the responsibility of the thread scheduler to decide which thread will get the CPU first.

The thread scheduler selects the thread that has the highest priority, and the thread begins the execution of the job. If a thread is already in runnable state and another thread (that has higher priority) reaches in the runnable state, then the current thread is pre-empted from the processor, and the arrived thread with higher priority gets the CPU time.

When two threads (Thread 2 and Thread 3) having the same priorities and arrival time, the scheduling will be decided on the basis of FCFS algorithm. Thus, the thread that arrives first gets the opportunity to execute first.

Thread.sleep() in Java with Examples

The Java Thread class provides the two variant of the sleep() method. First one accepts only an arguments, whereas the other variant accepts two arguments. The method sleep() is being used to halt the working of a thread for a given amount of time. The time up to which the thread remains in the sleeping state is known as the sleeping time of the thread. After the sleeping time is over, the thread starts its execution from where it has left.

The sleep() Method Syntax:

Following are the syntax of the sleep() method.

1. **public static void sleep(`long` mls) throws InterruptedException**
2. **public static void sleep(`long` mls, `int` n) throws InterruptedException**

The method sleep() with the one parameter is the native method, and the implementation of the native method is accomplished in another programming language. The other methods having the two parameters are not the native method. That is, its implementation is accomplished in Java. We can access the sleep() methods with the help of the Thread class, as the signature of the sleep() methods contain the static keyword. The native, as well as the non-native method, throw a checked Exception. Therefore, either try-catch block or the throws keyword can work here.

The Thread.sleep() method can be used with any thread. It means any other thread or the main thread can invoke the sleep() method.

Parameters:

The following are the parameters used in the sleep() method.

mls: The time in milliseconds is represented by the parameter mls. The duration for which the thread will sleep is given by the method sleep().

n: It shows the additional time up to which the programmer or developer wants the thread to be in the sleeping state. The range of n is from 0 to 999999.

The method does not return anything.

Important Points to Remember About the Sleep() Method

Whenever the Thread.sleep() methods execute, it always halts the execution of the current thread.

Whenever another thread does interruption while the current thread is already in the sleep mode, then the InterruptedException is thrown.

If the system that is executing the threads is busy, then the actual sleeping time of the thread is generally more as compared to the time passed in arguments. However, if the system executing the sleep() method has less load, then the actual sleeping time of the thread is almost equal to the time passed in the argument.

Example of the sleep() method in Java : on the custom thread

The following example shows how one can use the sleep() method on the custom thread.

FileName: TestSleepMethod1.java

```
class TestSleepMethod1 extends Thread{  
  
    public void run(){  
  
        for(int i=1;i<5;i++){  
  
            // the thread will sleep for the 500 milli seconds  
  
            try{Thread.sleep(500);}catch(InterruptedException  
e){System.out.println(e);}  
  
            System.out.println(i);  
  
        }  
  
    }  
  
    public static void main(String args[]){  
  
        TestSleepMethod1 t1=new TestSleepMethod1();  
  
        TestSleepMethod1 t2=new TestSleepMethod1();  
  
  
  
        t1.start();  
  
        t2.start();  
  
    }  
}
```

Can we start a thread twice

No. After starting a thread, it can never be started again. If you do so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

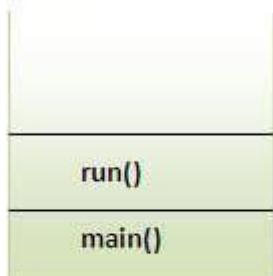
```
public class TestThreadTwice1 extends Thread{  
  
    public void run(){  
  
        System.out.println("running...");  
  
    }  
  
    public static void main(String args[]){  
  
        TestThreadTwice1 t1=new TestThreadTwice1();  
  
        t1.start();  
  
        t1.start();  
  
    }  
}
```

What if we call Java run() method directly instead start() method?

- Each thread starts in a separate call stack.
- Invoking the run() method from the main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

FileName: TestCallRun1.java

```
class TestCallRun1 extends Thread{  
  
    public void run(){  
  
        System.out.println("running...");  
  
    }  
  
    public static void main(String args[]){  
  
        TestCallRun1 t1=new TestCallRun1();  
  
        t1.run();//fine, but does not start a separate call stack  
  
    }  
  
}  
  
running...
```



Stack
(main thread)

Problem if you direct call run() method

FileName: TestCallRun2.java

```
class TestCallRun2 extends Thread{  
    public void run(){  
        for(int i=1;i<5;i++){  
            try{Thread.sleep(500);}catch(InterruptedException  
e){System.out.println(e);}  
            System.out.println(i);  
        }  
    }  
  
    public static void main(String args[]){  
        TestCallRun2 t1=new TestCallRun2();  
        TestCallRun2 t2=new TestCallRun2();  
  
        t1.run();  
        t2.run();  
    }  
}
```

Java join() method

The `join()` method in Java is provided by the `java.lang.Thread` class that permits one thread to wait until the other thread to finish its execution. Suppose `th` be the object the class `Thread` whose thread is doing its execution currently, then the `th.join();` statement ensures that `th` is finished before the program does the execution of the next statement. When there are more than one thread invoking the `join()` method, then it leads to overloading on the `join()` method that permits the developer or programmer to mention the waiting period. However, similar to the `sleep()` method in Java, the `join()` method is also dependent on the operating system for the timing, so we should not assume that the `join()` method waits equal to the time we mention in the parameters. The following are the three overloaded `join()` methods.

Description of The Overloaded join() Method

`join():` When the `join()` method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the `join()` method is invoked has achieved its dead state. If interruption of the thread occurs, then it throws the `InterruptedException`.

Syntax:

1. `public final void join() throws InterruptedException`

`join(long mls):` When the `join()` method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the `join()` method is invoked called is dead or the wait for the specified time frame(in milliseconds) is over.

Syntax:

1. `public final synchronized void join(long mls) throws InterruptedException,`
`where mls is in milliseconds`

join(long mls, int nanos): When the join() method is invoked, the current thread stops its execution and go into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked called is dead or the wait for the specified time frame(in milliseconds + nanos) is over.

Syntax:

1. **public final synchronized void join(long mls, int nanos) throws InterruptedException,** where mls is in milliseconds.

Example of join() Method in Java

The following program shows the usage of the join() method.

FileName: ThreadJoinExample.java

```
// A Java program for understanding  
// the joining of threads  
  
// import statement  
import java.io.*;  
  
  
// The ThreadJoin class is the child class of the class Thread  
class ThreadJoin extends Thread  
{  
    // overriding the run method  
    public void run()  
    {
```

```
for (int j = 0; j < 2; j++)  
{  
    try  
    {  
        // sleeping the thread for 300 milli seconds  
        Thread.sleep(300);  
  
        System.out.println("The current thread name is: " +  
                           Thread.currentThread().getName());  
    }  
  
    // catch block for catching the raised exception  
  
    catch(Exception e)  
    {  
        System.out.println("The exception has been caught: " + e);  
    }  
  
    System.out.println(j);  
}  
}  
}
```

```
public class ThreadJoinExample  
{  
    // main method
```

```
public static void main (String args[])
{
    // creating 3 threads
    ThreadJoin th1 = new ThreadJoin();
    ThreadJoin th2 = new ThreadJoin();
    ThreadJoin th3 = new ThreadJoin();

    // thread th1 starts
    th1.start();

    // starting the second thread after when
    // the first thread th1 has ended or died.
    try
    {
        System.out.println("The current thread name is: "+
        Thread.currentThread().getName());
    }

    // invoking the join() method
    th1.join();
}
```

```
// catch block for catching the raised exception  
  
catch(Exception e)  
  
{  
  
    System.out.println("The exception has been caught " + e);  
  
}
```

```
// thread th2 starts
```

```
th2.start();
```

```
// starting the th3 thread after when the thread th2 has ended or died.
```

```
try  
  
{  
  
    System.out.println("The current thread name is: " +  
        Thread.currentThread().getName());  
  
    th2.join();  
  
}
```

```
// catch block for catching the raised exception
```

```
catch(Exception e)  
  
{  
  
    System.out.println("The exception has been caught " + e);  
  
}
```

```
// thread th3 starts  
  
th3.start();  
  
}  
  
}
```

Naming Thread and Current Thread

Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name, i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using the `setName()` method. The syntax of `setName()` and `getName()` methods are given below:

1. `public String getName():` is used to **return the name of a thread.**
2. `public void setName(String name):` is used to **change the name of a thread.**

We can also set the name of a thread directly when we create a new thread using the constructor of the class.

Example of naming a thread : Using `setName()` Method

FileName: **TestMultiNaming1.java**

```
class TestMultiNaming1 extends Thread{  
  
    public void run(){  
  
        System.out.println("running...");  
  
    }  
  
    public static void main(String args[]){  
  
        TestMultiNaming1 t1=new TestMultiNaming1();  
    }  
}
```

```

TestMultiNaming1 t2=new TestMultiNaming1();

System.out.println("Name of t1:"+t1.getName());

System.out.println("Name of t2:"+t2.getName());

t1.start();

t2.start();

t1.setName("Sonoo Jaiswal");

System.out.println("After changing name of t1:"+t1.getName());

}

}

```

Example of naming a thread : Without Using setName() Method

One can also set the name of a thread at the time of the creation of a thread, without using the setName() method. Observe the following code.

FileName: ThreadNamingExample.java

```

// A Java program that shows how one can

// set the name of a thread at the time

// of creation of the thread

// import statement

import java.io.*;



// The ThreadNameClass is the child class of the class Thread

```

```
class ThreadName extends Thread

{

// constructor of the class

ThreadName(String threadName)

{

// invoking the constructor of

// the superclass, which is Thread class.

super(threadName);

}

}
```

```
// overriding the method run()

public void run()

{

System.out.println(" The thread is executing....");

}

}
```

```
public class ThreadNamingExample

{

// main method

public static void main (String args[])
}
```

```
{  
  
    // creating two threads and setting their name  
  
    // using the constructor of the class  
  
    ThreadName th1 = new ThreadName("JavaTpoint1");  
  
    ThreadName th2 = new ThreadName("JavaTpoint2");  
  
  
  
    // invoking the getName() method to get the names  
  
    // of the thread created above  
  
    System.out.println("Thread - 1: " + th1.getName());  
  
    System.out.println("Thread - 2: " + th2.getName());  
  
  
  
  
    // invoking the start() method on both the threads  
  
    th1.start();  
  
    th2.start();  
  
}  
  
}
```

Current Thread

The `currentThread()` method returns a reference of the currently executing thread.

1. `public static Thread currentThread()`

Example of `currentThread()` method

FileName: TestMultiNaming2.java

```
class TestMultiNaming2 extends Thread{  
  
    public void run(){  
  
        System.out.println(Thread.currentThread().getName());  
  
    }  
  
    public static void main(String args[]){  
  
        TestMultiNaming2 t1=new TestMultiNaming2();  
  
        TestMultiNaming2 t2=new TestMultiNaming2();  
  
  
  
        t1.start();  
  
        t2.start();  
  
    }  
}
```

Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

public final int getPriority(): The `java.lang.Thread.getPriority()` method returns the priority of the given thread.

`public final void setPriority(int newPriority):` The `java.lang.Thread.setPriority()` method updates or assign the priority of the thread to `newPriority`. The method throws `IllegalArgumentException` if the value `newPriority` goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

1. `public static int MIN_PRIORITY`
2. `public static int NORM_PRIORITY`
3. `public static int MAX_PRIORITY`

Default priority of a thread is 5 (`NORM_PRIORITY`). The value of `MIN_PRIORITY` is 1 and the value of `MAX_PRIORITY` is 10.

Example of priority of a Thread:

FileName: `ThreadPriorityExample.java`

```
// Importing the required classes

import java.lang.*;

public class ThreadPriorityExample extends Thread

{

    // Method 1

    // Whenever the start() method is called by a thread

    // the run() method is invoked

    public void run()

    {

        // the print statement
    }
}
```

```
System.out.println("Inside the run() method");

}

// the main method

public static void main(String args[])
{
    // Creating threads with the help of ThreadPriorityExample class

    ThreadPriorityExample th1 = new ThreadPriorityExample();

    ThreadPriorityExample th2 = new ThreadPriorityExample();

    ThreadPriorityExample th3 = new ThreadPriorityExample();

    // We did not mention the priority of the thread.

    // Therefore, the priorities of the thread is 5, the default value

    // 1st Thread

    // Displaying the priority of the thread

    // using the getPriority() method

    System.out.println("Priority of the thread th1 is : " + th1.getPriority());

    // 2nd Thread

    // Display the priority of the thread

    System.out.println("Priority of the thread th2 is : " + th2.getPriority());
```

```
// 3rd Thread

// // Display the priority of the thread

System.out.println("Priority of the thread th2 is : " + th2.getPriority());


// Setting priorities of above threads by

// passing integer arguments

th1.setPriority(6);

th2.setPriority(3);

th3.setPriority(9);

// 6

System.out.println("Priority of the thread th1 is : " + th1.getPriority());

// 3

System.out.println("Priority of the thread th2 is : " + th2.getPriority());


// 9

System.out.println("Priority of the thread th3 is : " + th3.getPriority());


// Main thread
```

```

// Displaying name of the currently executing thread

System.out.println("Currently Executing The Thread : " +
Thread.currentThread().getName());

System.out.println("Priority of the main thread is : " +
Thread.currentThread().getPriority());

// Priority of the main thread is 10 now

Thread.currentThread().setPriority(10);

System.out.println("Priority of the main thread is : " +
Thread.currentThread().getPriority());

}

}

```

Daemon Thread in Java

Daemon thread in Java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

Points to remember for Daemon Thread in Java

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.

- Its life depends on user threads.
- It is a low priority thread.

Why JVM terminates the daemon thread if there is no user thread?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

Methods for Java Daemon thread by Thread class

The `java.lang.Thread` class provides two methods for java daemon thread.

No.	Method	Description
1)	<code>public void setDaemon(boolean status)</code>	is used to mark the current thread as daemon thread or user thread.
2)	<code>public boolean isDaemon()</code>	is used to check that current is daemon.

Simple example of Daemon thread in java

File: `MyThread.java`

```
public class TestDaemonThread1 extends Thread{

    public void run(){

        if(Thread.currentThread().isDaemon()){//checking for daemon thread
            System.out.println("daemon thread work");
        }
    }
}
```

```

        System.out.println("user thread work");

    }

}

public static void main(String[] args){

    TestDaemonThread1 t1=new TestDaemonThread1(); //creating thread

    TestDaemonThread1 t2=new TestDaemonThread1();

    TestDaemonThread1 t3=new TestDaemonThread1();

    t1.setDaemon(true); //now t1 is daemon thread

    t1.start(); //starting threads

    t2.start();

    t3.start();

}
}

```

Java Thread Pool

Java Thread pool represents a group of worker threads that are waiting for the job and reused many times.

In the case of a thread pool, a group of fixed-size threads is created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, the thread is contained in the thread pool again.

Thread Pool Methods

`newFixedThreadPool(int s)`: The method creates a thread pool of the fixed size s.

newCachedThreadPool(): The method creates a new thread pool that creates the new threads when needed but will still use the previously created thread whenever they are available to use.

newSingleThreadExecutor(): The method creates a new thread.

Advantage of Java Thread Pool

Better performance It saves time because there is no need to create a new thread.

Real time usage

It is used in Servlet and JSP where the container creates a thread pool to process the request.

Example of Java Thread Pool

Let's see a simple example of the Java thread pool using ExecutorService and Executors.

File: WorkerThread.java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
class WorkerThread implements Runnable {
    private String message;
    public WorkerThread(String s){
        this.message=s;
    }
    public void run() {
        System.out.println(Thread.currentThread().getName()+" (Start)
message = "+message);
```

```

processmessage();//call processmessage method that sleeps the thread
for 2 seconds

    System.out.println(Thread.currentThread().getName()+" (End)");//prints
thread name

}

private void processmessage() {

    try { Thread.sleep(2000); } catch (InterruptedException e) {
e.printStackTrace(); }

}

}

```

File: TestThreadPool.java

```

public class TestThreadPool {

    public static void main(String[] args) {

        ExecutorService executor = Executors.newFixedThreadPool(5);//creating
a pool of 5 threads

        for (int i = 0; i < 10; i++) {

            Runnable worker = new WorkerThread(" " + i);

            executor.execute(worker);//calling execute method of ExecutorService

        }

        executor.shutdown();

        while (!executor.isTerminated()) { }

System.out.println("Finished all threads");
    }
}

```

```
}
```

```
}
```

Output:

```
pool-1-thread-1 (Start) message = 0
pool-1-thread-2 (Start) message = 1
pool-1-thread-3 (Start) message = 2
pool-1-thread-5 (Start) message = 4
pool-1-thread-4 (Start) message = 3
pool-1-thread-2 (End)
pool-1-thread-2 (Start) message = 5
pool-1-thread-1 (End)
pool-1-thread-1 (Start) message = 6
pool-1-thread-3 (End)
pool-1-thread-3 (Start) message = 7
pool-1-thread-4 (End)
pool-1-thread-4 (Start) message = 8
pool-1-thread-5 (End)
pool-1-thread-5 (Start) message = 9
pool-1-thread-2 (End)
pool-1-thread-1 (End)
pool-1-thread-4 (End)
pool-1-thread-3 (End)
pool-1-thread-5 (End)
```

Finished all threads

ThreadGroup in Java

Java provides a convenient way to group multiple threads in a single object. In such a way, we can suspend, resume or interrupt a group of threads by a single method call.

Note: Now `suspend()`, `resume()` and `stop()` methods are deprecated.

Java thread group is implemented by `java.lang.ThreadGroup` class.

A `ThreadGroup` represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.

A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

No.	Constructor	Description
1)	<code>ThreadGroup(String name)</code>	creates a thread group with given name.
2)	<code>ThreadGroup(ThreadGroup parent, String name)</code>	creates a thread group with a given parent group and name.

Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using `free()` function in C language and `delete()` in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.

How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.

How can an object be unreferenced?



1) By nulling a reference:

1. Employee e=**new Employee()**;
2. e=**null**;

2) By assigning a reference to another:

1. Employee e1=**new Employee()**;
2. Employee e2=**new Employee()**;
3. e1=e2;**//now the first object referred by e1 is available for garbage collection**

3) By anonymous object:

1. **new Employee();**

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

1. **protected void finalize(){}**

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

1. **public static void gc(){}**

Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Simple Example of garbage collection in java

```
public class TestGarbage1{  
    public void finalize(){System.out.println("object is garbage collected");}  
    public static void main(String args[]){  
        TestGarbage1 s1=new TestGarbage1();  
        TestGarbage1 s2=new TestGarbage1();  
        s1=null;  
        s2=null;  
        System.gc();
```

```

    }
}

object is garbage collected
object is garbage collected

```

Note: Neither finalization nor garbage collection is guaranteed.

Java Runtime class

Java Runtime class is used to *interact with java runtime environment*. Java Runtime class provides methods to execute a process, invoke GC, get total and free memory etc. There is only one instance of `java.lang.Runtime` class is available for one java application.

The `Runtime.getRuntime()` method returns the singleton instance of Runtime class.

Important methods of Java Runtime class

No.	Method	Description
1)	<code>public static Runtime getRuntime()</code>	returns the instance of Runtime class.
2)	<code>public void exit(int status)</code>	terminates the current virtual machine.
3)	<code>public void addShutdownHook(Thread hook)</code>	registers new hook thread.
4)	<code>public Process exec(String command) throws IOException</code>	executes given command in a separate process.
5)	<code>public int availableProcessors()</code>	returns no. of available processors.
6)	<code>public long freeMemory()</code>	returns amount of free memory in JVM.
7)	<code>public long totalMemory()</code>	returns amount of total memory in JVM.

Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
 1. Synchronized method.
 2. Synchronized block.
 3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block

3. By Using Static Synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package `java.util.concurrent.locks` contains several lock implementations.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

TestSynchronization1.java

```
class Table{  
  
    void printTable(int n){//method not synchronized  
  
        for(int i=1;i<=5;i++){  
  
            System.out.println(n*i);  
  
            try{  
  
                Thread.sleep(400);  
  
            }catch(Exception e){System.out.println(e);}  
  
        }  
  
    }  
  
}
```

```
class MyThread1 extends Thread{  
  
Table t;  
  
MyThread1(Table t){  
  
this.t=t;  
  
}  
  
public void run(){  
  
t.printTable(5);  
  
}  
  
}  
  
class MyThread2 extends Thread{  
  
Table t;  
  
MyThread2(Table t){  
  
this.t=t;  
  
}  
  
public void run(){  
  
t.printTable(100);  
  
}  
  
}  
  
class TestSynchronization1{  
  
public static void main(String args[]){
```

```

Table obj = new Table();//only one object

MyThread1 t1=new MyThread1(obj);

MyThread2 t2=new MyThread2(obj);

t1.start();

t2.start();

}

}

```

Java Synchronized Method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

TestSynchronization2.java

```

//example of java synchronized method

class Table{

    synchronized void printTable(int n){//synchronized method

        for(int i=1;i<=5;i++){

            System.out.println(n*i);

            try{

                Thread.sleep(400);

            }catch(Exception e){System.out.println(e);}

        }
    }
}
```

```
    }

}

class MyThread1 extends Thread{

Table t;

MyThread1(Table t){

this.t=t;

}

public void run(){

t.printTable(5);

}

}

class MyThread2 extends Thread{

Table t;

MyThread2(Table t){

this.t=t;

}

public void run(){

t.printTable(100);

}

}
```

```
public class TestSynchronization2{  
    public static void main(String args[]){  
        Table obj = new Table(); //only one object  
        MyThread1 t1=new MyThread1(obj);  
        MyThread2 t2=new MyThread2(obj);  
        t1.start();  
        t2.start();  
    }  
}
```

Synchronized Block in Java

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.

If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to Remember

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.
- A Java synchronized block doesn't allow more than one JVM, to provide access control to a shared resource.
- The system performance may degrade because of the slower working of synchronized keyword.

- Java synchronized block is more efficient than Java synchronized method.

Syntax

```
synchronized (object reference expression) {  
    //code block  
}
```

Example of Synchronized Block

Let's see the simple example of synchronized block.

TestSynchronizedBlock1.java

```
class Table  
{  
    void printTable(int n){  
        synchronized(this){//synchronized block  
            for(int i=1;i<=5;i++){  
                System.out.println(n*i);  
            try{  
                Thread.sleep(400);  
            }catch(Exception e){System.out.println(e);}  
        }  
    }  
}//end of the method
```

```
class MyThread1 extends Thread{  
  
Table t;  
  
MyThread1(Table t){  
  
this.t=t;  
  
}  
  
public void run(){  
  
t.printTable(5);  
  
}  
  
}  
  
class MyThread2 extends Thread{  
  
Table t;  
  
MyThread2(Table t){  
  
this.t=t;  
  
}  
  
public void run(){  
  
t.printTable(100);  
  
}  
  
}  
  
public class TestSynchronizedBlock1{  
  
public static void main(String args[]){
```

```

Table obj = new Table(); //only one object

MyThread1 t1=new MyThread1(obj);

MyThread2 t2=new MyThread2(obj);

t1.start();

t2.start();

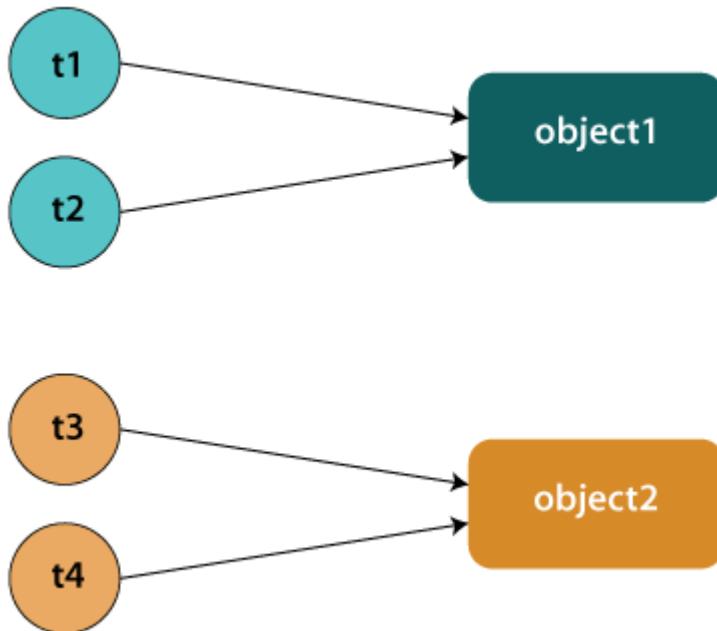
}

}

```

Static Synchronization

If you make any static method as synchronized, the lock will be on the class not on object.



Problem without static synchronization

Suppose there are two objects of a shared class (e.g. Table) named object1 and object2. In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.

We don't want interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

Example of Static Synchronization

In this example we have used synchronized keyword on the static method to perform static synchronization.

TestSynchronization4.java

```
class Table
{
    synchronized static void printTable(int n){
        for(int i=1;i<=10;i++){
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }catch(Exception e){}
        }
    }

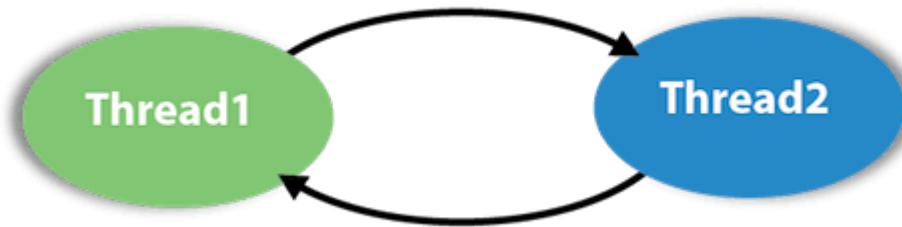
    class MyThread1 extends Thread{
        public void run(){
            Table.printTable(1);
        }
    }

    class MyThread2 extends Thread{
        public void run(){
            Table.printTable(10);
        }
    }
}
```

```
class MyThread3 extends Thread{  
    public void run(){  
        Table.printTable(100);  
    }  
}  
  
class MyThread4 extends Thread{  
    public void run(){  
        Table.printTable(1000);  
    }  
}  
  
public class TestSynchronization4{  
    public static void main(String t[]){  
        MyThread1 t1=new MyThread1();  
        MyThread2 t2=new MyThread2();  
        MyThread3 t3=new MyThread3();  
        MyThread4 t4=new MyThread4();  
        t1.start();  
        t2.start();  
        t3.start();  
        t4.start();  
    }  
}
```

Deadlock in Java

Deadlock in Java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Example of Deadlock in Java

TestDeadlockExample1.java

```
public class TestDeadlockExample1 {

    public static void main(String[] args) {

        final String resource1 = "ratan jaiswal";

        final String resource2 = "vimal jaiswal";

        // t1 tries to lock resource1 then resource2

        Thread t1 = new Thread() {

            public void run() {

                synchronized (resource1) {

                    System.out.println("Thread 1: locked resource 1");

                    try { Thread.sleep(100); } catch (Exception e) {}

                    synchronized (resource2) {

                        System.out.println("Thread 1: locked resource 2");

                    }

                }

            }

        }

    }

}
```

```
    }

};

// t2 tries to lock resource2 then resource1

Thread t2 = new Thread() {

    public void run() {

        synchronized (resource2) {

            System.out.println("Thread 2: locked resource 2");



        try { Thread.sleep(100);} catch (Exception e) {}



        synchronized (resource1) {

            System.out.println("Thread 2: locked resource 1");



        }

    }

}

};

t1.start();

t2.start();

}
```

```
}
```

How to avoid deadlock?

A solution for a problem is found at its roots. In deadlock it is the pattern of accessing the resources A and B, is the main issue. To solve the issue we will have to simply re-order the statements where the code is accessing shared resources.

DeadlockSolved.java

```
public class DeadlockSolved {  
  
    public static void main(String ar[]) {  
  
        DeadlockSolved test = new DeadlockSolved();  
  
        final resource1 a = test.new resource1();  
        final resource2 b = test.new resource2();  
  
        // Thread-1  
        Runnable b1 = new Runnable() {  
  
            public void run() {  
                synchronized (b) {  
                    try {  
                        /* Adding delay so that both threads can start trying to lock  
                        resources */  
                        Thread.sleep(100);  
                    } catch (InterruptedException e) {  
                    }  
                }  
            }  
        };  
        b1.run();  
        a.method();  
    }  
}
```

```
e.printStackTrace();  
}  
  
// Thread-1 have resource1 but need resource2 also  
  
synchronized (a) {  
  
    System.out.println("In block 1");  
  
}  
  
}  
  
}  
  
};  
  
  
// Thread-2  
  
Runnable b2 = new Runnable() {  
  
    public void run() {  
  
        synchronized (b) {  
  
            // Thread-2 have resource2 but need resource1 also  
  
            synchronized (a) {  
  
                System.out.println("In block 2");  
  
            }  
  
        }  
  
    }  
  
};
```

```
    new Thread(b1).start();

    new Thread(b2).start();

}
```

```
// resource1

private class resource1 {

    private int i = 10;
```

```
    public int getI() {

        return i;

    }
```

```
    public void setI(int i) {

        this.i = i;

    }

}
```

```
// resource2

private class resource2 {

    private int i = 20;
```

```
public int getI() {  
    return i;  
}  
  
public void setI(int i) {  
    this.i = i;  
}  
}
```

How to Avoid Deadlock in Java?

Deadlocks cannot be completely resolved. But we can avoid them by following basic rules mentioned below:

- 1. Avoid Nested Locks:** We must avoid giving locks to multiple threads, this is the main reason for a deadlock condition. It normally happens when you give locks to multiple threads.
- 2. Avoid Unnecessary Locks:** The locks should be given to the important threads. Giving locks to the unnecessary threads that cause the deadlock condition.
- 3. Using Thread Join:** A deadlock usually happens when one thread is waiting for the other to finish. In this case, we can use join with a maximum time that a thread will take.

Inter-thread Communication in Java

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:

- **wait()**
- **notify()**
- **notifyAll()**

1) wait() method

The **wait()** method causes current thread to release the lock and wait until either another thread invokes the **notify()** method or the **notifyAll()** method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

2) notify() method

The **notify()** method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

1. **public final void notify()**

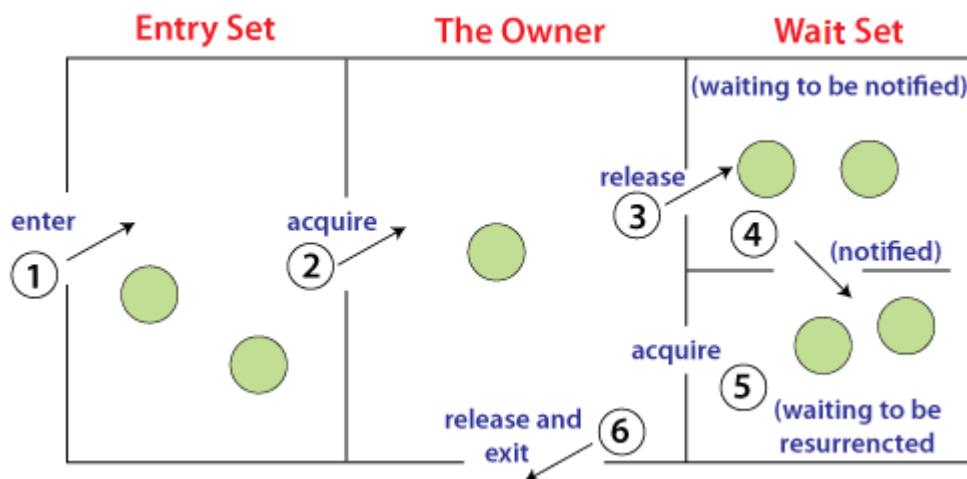
3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

Syntax:

1. `public final void notifyAll()`

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by one thread.
3. Now thread goes to waiting state if you call `wait()` method on the object.
Otherwise it releases the lock and exits.
4. If you call `notify()` or `notifyAll()` method, thread moves to the notified state (Runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Why `wait()`, `notify()` and `notifyAll()` methods are defined in `Object` class not `Thread` class?

It is because they are related to lock and object has a lock.

Difference between `wait` and `sleep`?

Let's see the important differences between `wait` and `sleep` methods.

<code>wait()</code>	<code>sleep()</code>
The <code>wait()</code> method releases the lock.	The <code>sleep()</code> method doesn't release the lock.
It is a method of <code>Object</code> class	It is a method of <code>Thread</code> class
It is the non-static method	It is the static method
It should be notified by <code>notify()</code> or <code>notifyAll()</code> methods	After the specified amount of time, sleep is completed.

Example of Inter Thread Communication in Java

Let's see the simple example of inter thread communication.

Test.java

```
class Customer{  
    int amount=10000;  
  
    synchronized void withdraw(int amount){  
        System.out.println("going to withdraw...");  
  
        if(this.amount<amount){  
            System.out.println("Less balance; waiting for deposit...");  
            try{wait();}catch(Exception e){}  
        }  
    }  
}
```

```
}

this.amount-=amount;

System.out.println("withdraw completed...");

}
```

```
synchronized void deposit(int amount){

System.out.println("going to deposit...");

this.amount+=amount;

System.out.println("deposit completed... ");

notify();

}
```

```
class Test{

public static void main(String args[]){

final Customer c=new Customer();

new Thread(){

public void run(){c.withdraw(15000);}

}.start();

new Thread(){

public void run(){c.deposit(10000);}

}.start();
```

```
}}
```

Interrupting a Thread:

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption.

The 3 methods provided by the Thread class for interrupting a thread

- **public void interrupt()**
- **public static boolean interrupted()**
- **public boolean isInterrupted()**

Example of interrupting a thread that stops working

In this example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where sleep() or wait() method is invoked. Let's first see the example where we are propagating the exception.

TestInterruptingThread1.java

```
class TestInterruptingThread1 extends Thread{
    public void run(){
        try{
            Thread.sleep(1000);
        }
    }
}
```

```

System.out.println("task");

}catch(InterruptedException e){
    throw new RuntimeException("Thread interrupted..."+e);
}

}

public static void main(String args[]){
    TestInterruptingThread1 t1=new TestInterruptingThread1();
    t1.start();
    try{
        t1.interrupt();
    }catch(Exception e){System.out.println("Exception handled "+e);}
}

```

Example of interrupting a thread that doesn't stop working

In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working.

TestInterruptingThread2.java

```

class TestInterruptingThread2 extends Thread{

    public void run(){
        try{

```

```

Thread.sleep(1000);

System.out.println("task");

}catch(InterruptedException e){

System.out.println("Exception handled "+e);

}

System.out.println("thread is running...");

}

public static void main(String args[]){

TestInterruptingThread2 t1=new TestInterruptingThread2();

t1.start();

t1.interrupt();

}

}

```

Example of interrupting thread that behaves normally

If thread is not in sleeping or waiting state, calling the `interrupt()` method sets the `interrupted` flag to true that can be used to stop the thread by the java programmer later.

TestInterruptingThread3.java

```
class TestInterruptingThread3 extends Thread{
```

```

public void run(){

for(int i=1;i<=5;i++){

System.out.println(i);

}

}

public static void main(String args[]){

TestInterruptingThread3 t1=new TestInterruptingThread3();

t1.start();

t1.interrupt();

}

}

```

What about isInterrupted and interrupted method?

The `isInterrupted()` method returns the interrupted flag either true or false. The static `interrupted()` method returns the interrupted flag after that it sets the flag to false if it is true.

TestInterruptingThread4.java

```

public class TestInterruptingThread4 extends Thread{

public void run(){

```

```
for(int i=1;i<=2;i++){
    if(Thread.interrupted()){
        System.out.println("code for interrupted thread");
    }
    else{
        System.out.println("code for normal thread");
    }
}//end of for loop
```

```
public static void main(String args[]){
    TestInterruptingThread4 t1=new TestInterruptingThread4();
    TestInterruptingThread4 t2=new TestInterruptingThread4();
    t1.start();
    t1.interrupt();
    t2.start();
}
```

}

COLLECTIONS IN JAVA

Collections in Java is a framework that stores and manipulates a group of objects. It is a hierarchy of interfaces and classes that provides an easy management of a group of objects. Java Collection framework provides many interfaces (List, Queue, Deque, Set) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

What is a Collection in Java?

Consider the example of a piggy bank. We all had it during our childhood where we used to store our coins. This piggy bank is called a Collection and the coins are nothing but objects. Technically, a collection is an object or a container that stores a group of other objects.

A Collection in Java is an object which represents a group of objects, known as its elements. It is a single unit. They are used to standardize the way in which objects are handled in the class.

Now that we know what Collection in Java is, let us try to understand the real-life use cases:

- Linked list emulates your browsing history, train coaches which are connected to each other etc.
- Stacks are like a stack of plates or trays in which the top-most one gets picked first.
- Queue is same as the real-life queues, the one who enters the queue first, it leaves it first too.

What is Java Collections Framework?

The Collections Framework is defined as a unified architecture for representing and manipulating collections. In Java, the Collections Framework is a hierarchy of interfaces and classes that provides an easy management of a group of objects.

The `java.util` package contains the powerful tool of Collections Framework. It was defined in JDK 1.2 version which is one of the most used frameworks till date.

It provides a ready-made architecture for interfaces and classes and used for storing and manipulating a group of objects. All collections framework contain interfaces, classes and algorithms.

Now you might be wondering what is its need. Collections in Java were not a part of the original Java release. But prior to the release, Vectors, Stacks and Arrays were there. They had one major disadvantage, and that was lesser similarities. They didn't have a common interface and interconnection with each other. In this case, it becomes tedious for the user to remember all the functions and syntax. Plus, the conventional ways like arrays and stacks weren't providing the desired performance and flexibility.

This is why the Collections Framework and collection in Java was introduced in order to overcome the drawbacks mentioned above.

Differences between the Collection and Collections Framework in Java

The following are the differences between Collection in Java and Collections Framework

Collection in Java	Collections Framework
Collection in Java is a class.	Collections Framework is a framework.
It is a single unit which contains and manipulates a group of objects.	They are used to manipulate collections.

Advantages of the Collections Framework

As mentioned above, the Collections Framework in Java carries a lot of advantages.

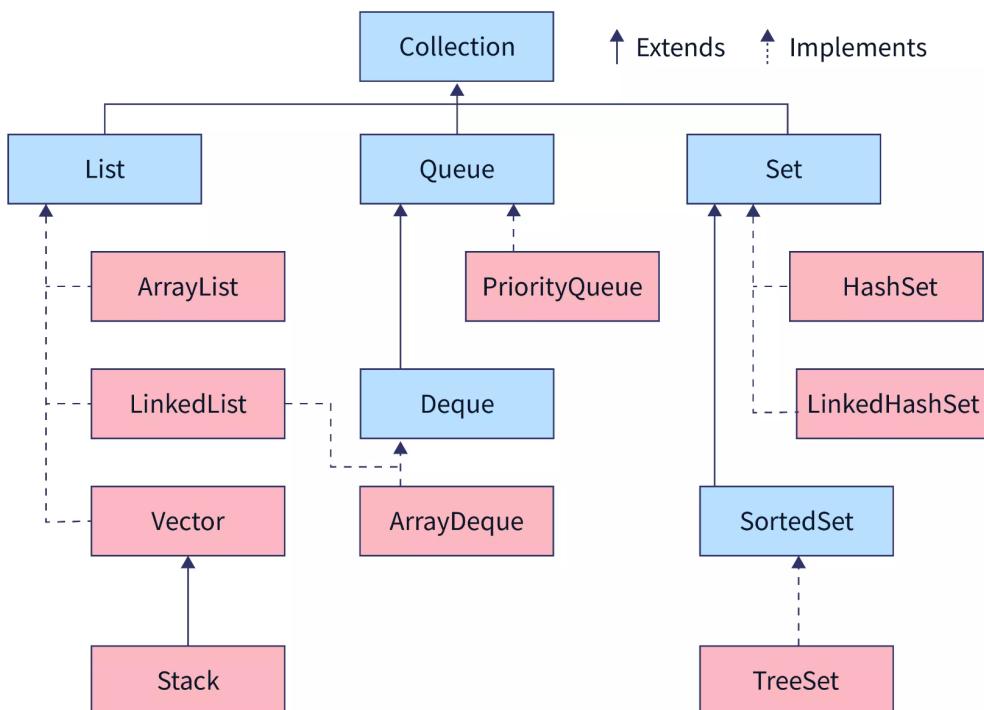
- Provides high-performance and high efficiency. This is due to the fact that various implementations of each interface are interchangeable, so programs can be written by switching implementations.
- Some methods of each interface of the Collections Framework have a uniform implementation. The Collections API has basic interfaces like Set, Map and List, so the classes that implement them have a few common set of methods.
- Reduces the programming effort by providing useful data structures and algorithms. We don't have to write our own data structure as it has already been provided to us.
- Facilitates code reusability. Collections Framework provides common classes and interfaces that can be used with different types of collections.

Hierarchy of Collections Framework

The Collections Framework in Java follows a certain hierarchy. All the classes and interfaces in the Collections Framework come under the `java.util` package.

But before getting acquainted with the hierarchy of the Collections Framework in Java, Let us understand these terms.

- Class: A class is a collection of similar types of objects. It is an implementation of the collection interface.
- Interface: An interface is the abstract data type. It is at the topmost position in the framework hierarchy. In Layman's terms, it is a group of related method with empty bodies. Abstraction is a process of hiding the implementation details from the user by only providing the functionality. Using interfaces, we can achieve (complete) abstraction.
- extends: It is a keyword which is used to develop inheritance between two classes and two interfaces. Inheritance in Java refers to the concept that one can create new classes that are built upon existing classes.
- implements: implements is also a keyword used to develop inheritance between class and interface. A class implements an interface.



Collection Interface

The Collection Interface is the root or the foundation on which the Collections Framework is built. It is a general interface that has the declaration:

```
interface Collection<E>
```

Here, E is the type of object that the collection will hold.

It provides the basic operations like adding, removing, clearing the elements in a collection, checking whether the collection is empty etc.

List, Queue and Set are the components that extend the Collection Interface.

Methods of the Collection Interface

The Collection Interface has the following methods. The methods declared in an interface are by default abstract (only method signature, no body of the method).

Method	Description
boolean add(E obj)	It is used to add an object obj to the collection. Returns true if obj was added, else returns false if the element obj was already a member and the collection does not allow duplicates.
boolean addAll(Collection<? extends E> c)	Adds all the elements of c to the collection. Returns true if the object was added, else returns false.
void clear()	Removes all the elements from the collection.
int size()	Returns the number of elements present in the collection.

Iterator iterator()	Returns an iterator for the collection. It is an object that can be used to loop through collections.
boolean contains(Object obj)	Checks if the object is present in the collection and returns true if found. Else, it returns false.
boolean containsAll(Collection<?> c)	Returns true if the collection contains all the elements of c, else returns false.
int hashCode()	It returns the hash code for the collection i.e., returns an integer or a 4 byte value which is generated by the hashing algorithm.
boolean equals(Object obj)	It returns true if the collection and obj are equal, else returns false.
boolean isEmpty()	Returns true if the collection is empty, else returns false.
boolean remove(Object obj)	Removes one instance of obj from the collection. It returns true if the element was removed, else returns false.
boolean removeAll(Collection<?> c)	Removes all elements of c from the collection. Returns true if the elements were removed, else returns false.
boolean retainAll(Collection<?> c)	It retains only those elements which are in c and removes the other elements from the collection. It returns true if the elements were removed, else returns false.
default Spliterator spliterator()	Returns a spliterator to the collection. A spliterator can be used to iterate over a collection and split it into **multiple sets.

Iterator Interface

Iterator is an object that can be used to loop through collections.

As the name suggests, it is used to iterate over the elements. It is used to modify and iterate over the elements in a collection.

There are 3 methods in the Iterator interface:

- public Object next()- It returns the next element in the collection. It throws the exception of NoSuchElementException if there is no next element.
- public void remove()- It removes the current element from the collection.
- public boolean hasNext()- It returns true if there are more elements in the collection. Else, returns false.

Iterable Interface

The Iterable Interface allows the collection to be iterated over. The root interface for the entire collection framework is termed as the Iterable Interface. The collection interface extends the iterable interface, hence the sub-classes of the collection interface also implement the iterable interface, i.e, it automatically becomes a part of the iterable interface.

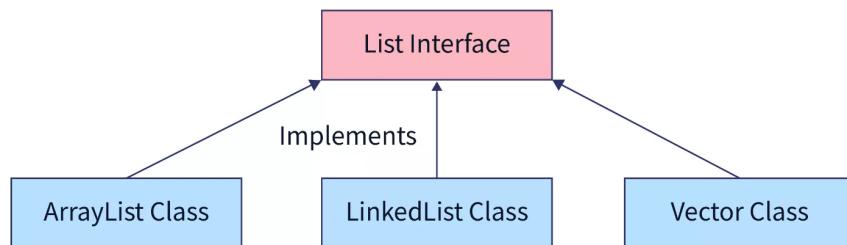
Syntax:

```
Iterator <E> iterator()
```

Returns an iterator of type E for the collection. It can be used to iterate over the elements of the collection.

List Interface

The list interface extends the collection interface. A list is used to store ordered collection of data and it may contain duplicates. Ordered collection means the order in which the elements are being inserted and they contain a specific value. The elements present, can be accessed or inserted by their position in the list using zero-based indexing. The list interface is implemented by LinkedList, ArrayList, Vectors and Stack classes. They are an important part of collections in Java.



SCALER
Topics

There are three classes implemented by the list interface and they are given below.

Syntax

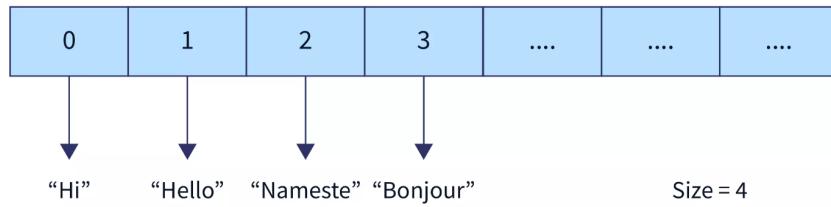
```
List<Data-Type>linkedlist= new LinkedList<Data-Type>();
```

```
List<Data-Type>arraylist= new ArrayList<Data-Type>();
```

```
List<Data-Type>vector= new Vector<Data-Type>();
```

1. ArrayList

- So, you must've figured it out from the name itself, that ArrayList is similar to Arrays. They are also called as dynamic arrays. That means, it does not have a fixed size. Its size can be increased or decreased if elements are added or removed.
- It implements the List Interface.
- It is similar to Vectors in C++.
- Since the ArrayList cannot be used for primitive data types like int,char etc. , we need to use a wrapper class.



SCALER
Topics

Important Features of ArrayList:

1. ArrayList inherits AbstractList class and implements the List interface.
2. ArrayList is initialized by size. However, the size is increased automatically if the collection grows or shrinks if the objects are removed from the collection.
3. Java ArrayList allows us to randomly access the list.
4. ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases.
5. ArrayList in Java can be seen as a vector in C++.
6. ArrayList is not Synchronized. Its equivalent synchronized class in Java is Vector.

Constructors in the ArrayList:

In order to create an ArrayList, we need to create an object of the ArrayList class. The ArrayList class consists of various constructors which allow the possible creation of the array list. The following are the constructors available in this class:

1. `ArrayList()`: This constructor is used to build an empty array list. If we wish to create an empty ArrayList with the name arr, then, it can be created as:

```
ArrayList arr = new ArrayList();
```

2. `ArrayList(Collection c)`: This constructor is used to build an array list initialized with the elements from the collection c. Suppose, we wish to create an ArrayList arr which contains the elements present in the collection c, then, it can be created as:

```
ArrayList arr = new ArrayList(c);
```

3. `ArrayList(int capacity)`: This constructor is used to build an array list with initial capacity being specified. Suppose we wish to create an ArrayList with the initial size being N, then, it can be created as:

```
ArrayList arr = new ArrayList(N);
```

In order to initialize an ArrayList, there are 3 ways.

1. Using the add Keyword

```
ArrayList<Data-Type> str = new ArrayList<Data-Type>();
str.add(<content>);
```

```
str.add(<content>);  
str.add(<content>);
```

Example

```
//initialising ArrayList using add keyword  
import java.util.*;  
  
public class ScalerTopics {  
    public static void main(String args[])  
    {  
        // create an ArrayList of String type  
        ArrayList<String> str = new ArrayList<String>();  
  
        // Initialize an ArrayList with add()  
        str.add("Scaler");  
        str.add("Topics");  
        str.add("Rocks");  
  
        // to print the ArrayList  
        System.out.println("ArrayList is" + str);  
    }  
}
```

Output

ArrayList is [Scaler, Topics, Rocks]

2. Using asList() AsList() method in Java is used to return a fixed-size list backed by the given array.

```
ArrayList<Data-Type> obj =  
    new ArrayList<Data-Type>(Arrays.asList(Obj A, Obj B, Obj C, ....));
```

Example

```
// initialise ArrayList using asList method  
  
import java.util.*;
```

```

public class ScalerTopics {
    public static void main(String args[])
    {
        // create an ArrayList of String type
        // and Initialize an ArrayList with asList()
        ArrayList<String> scaler = new ArrayList<String>(
            Arrays.asList("I",
                "love",
                "Java"));

        // to print the ArrayList
        System.out.println("ArrayList is " + scaler);
    }
}

```

Output

ArrayList is [I, Love, Java]

3. Using list.of() Method It is used to return immutable lists containing the specified elements.

```
List<Data-Type> obj = new ArrayList<>(
    List.of(Obj 1, Obj 1, Obj 1, ....));
```

Example

```
// initialise ArrayList using List.of() method
```

```
import java.util.*;
```

```

public class ScalerTopics {
    public static void main(String args[])
    {
        // create an ArrayList of String type
        // and Initialize an ArrayList with List.of()
        List<String> str = new ArrayList<>(
            List.of("Coding",
                "is",
                "Fun"));


```

```
// to print the ArrayList
```

```
        System.out.println("ArrayList is " + str);
    }
}
Output
```

ArrayList is [Coding, is, Fun]

Now that we are acquainted to the methods to initialise an ArrayList, let us look at an example.

```
import java.util.*;

public class ScalerTopics{
    public static void main(String args[]){
        //creating an ArrayList
        ArrayList<String> str= new ArrayList<String>();

        //displaying the initial size
        System.out.println("Size at the beginning "+str.size());

        //add elements
        str.add("Hello");
        str.add("Hi");
        str.add("Namaste");
        str.add("Bonjour");

        //displaying the ArrayList
        System.out.println(str);

        //displaying the size
        System.out.println("Size after addition "+str.size());

        //remove element at index 0
        str.remove(0);

        //display the new ArrayList
        System.out.println(str);

        //display the new size
        System.out.println("Size after removal "+str.size());
```

```
}
```

```
}
```

Output

```
Size at the beginning 0  
[Hello, Hi, Namaste, Bonjour]  
Size after addition 4  
[Hi, Namaste, Bonjour]  
Size after removal 3
```

Explanation: We saw the ArrayList methods like add(), remove() and size() which perform manipulations on ArrayLists.

Internal implementation of ArrayList

An ArrayList stores data in a resizable array. Before Java 8, when an ArrayList was created, an array of default size 10 was created internally. Now, when an ArrayList is created, an array of size zero is created. Only when the first element is inserted does the array size change to ten. This is called lazy initialization, and it saves a lot of memory.

Before adding an element in ArrayList, its capacity is checked. If the internal array is full. The elements from the old array will be copied to the new array. This increases the capacity of an ArrayList, which is a time-consuming process.

The capacity is the total number of cells. The size is the number of cells that have data in them.

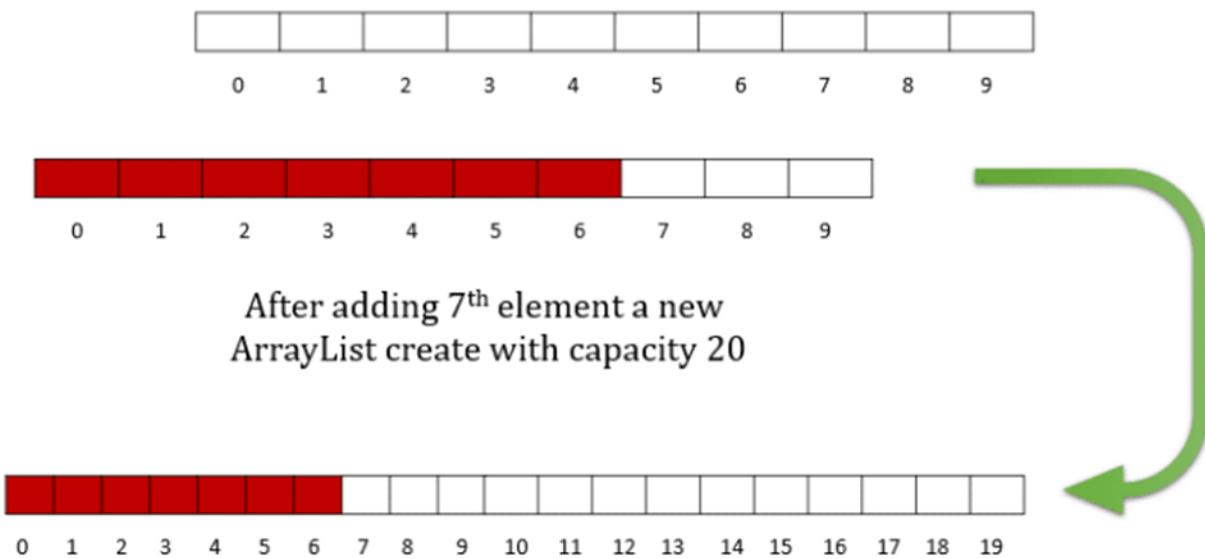
The size of ArrayList grows automatically which is fully based on load factor and current capacity. Basically, the load factor is the measure that decides when to increase the capacity of the ArrayList. The default load factor of an ArrayList is 0.75f.

ArrayList increases the capacity after each threshold(No of elements). Threshold =
(Current Capacity) * (Load Factor)
Threshold = $10 * 0.75 = 7$

Suppose we have an ArrayList with default capacity 10. We can add more than 10 elements in ArrayList.

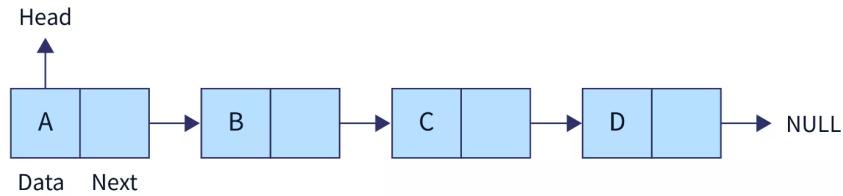
Whenever we perform an add operation, it checks whether the threshold value is equal to the number of elements present in ArrayList. If the size of the current element is

greater than the threshold. The JVM creates a new ArrayList and copies the old ArrayList into new. It creates the ArrayList size 10 to 20.



2. LinkedList

- The LinkedList class extends the AbstractSequentialList and it also extends the List, Deque and Queue interface.
- By this, we get a linked-list data structure.
- Linked List is a linear data structure where the elements are called as nodes.
- Here, each node has two fields- data and next. Data stores the actual piece of information and next points to the next node. 'Next' field is actually the address of the next node.
- Elements are not stored in a contiguous memory, so direct access to that element is not possible.
- LinkedList uses Doubly Linked List to store its elements while ArrayList internally uses a dynamic array to store its elements. LinkedList is faster in manipulation of data as it is node based which makes it unique.
- LinkedList is non-synchronised means multiple threads at a time can access the code. This means if one thread is working on LinkedList, other threads can also get a hold of it. Multiple operations on LinkedList can be performed at a time. For example, if addition is being performed by one thread, other operation can be performed by some other thread too.



SCALER
Topics

Constructors in the LinkedList:

In order to create a `LinkedList`, we need to create an object of the `LinkedList` class. The `LinkedList` class consists of various constructors that allow the possible creation of the list. The following are the constructors available in this class:

1. `LinkedList()`: This constructor is used to create an empty linked list. If we wish to create an empty `LinkedList` with the name `ll`, then, it can be created as:

```
LinkedList ll = new LinkedList();
```

2. `LinkedList(Collection C)`: This constructor is used to create an ordered list that contains all the elements of a specified collection, as returned by the collection's iterator. If we wish to create a `LinkedList` with the name `ll`, then, it can be created as:

```
LinkedList ll = new LinkedList(C);
```

Java LinkedList List Methods:

The following methods are inherited from List or Collection interface:

- `int size()`: to get the number of elements in the list.
- `boolean isEmpty()`: to check if list is empty or not.
- `boolean contains(Object o)`: Returns true if this list contains the specified element.

- `Iterator iterator()`: Returns an iterator over the elements in this list in proper sequence.
- `Object[] toArray()`: Returns an array containing all of the elements in this list in proper sequence.
- `boolean add(E e)`: Appends the specified element to the end of this list.
- `boolean remove(Object o)`: Removes the first occurrence of the specified element from this list.
- `boolean retainAll(Collection c)`: Retains only the elements in this list that are contained in the specified collection.
- `void clear()`: Removes all the elements from the list.
- `E get(int index)`: Returns the element at the specified position in the list.
- `E set(int index, E element)`: Replaces the element at the specified position in the list with the specified element.
- `ListIterator listIterator()`: Returns a list iterator over the elements in the list.
- `List subList(int fromIndex, int toIndex)`: Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive. The returned list is backed by this list, so non-structural changes in the returned list are reflected in this list, and vice-versa.

Java LinkedList Deque Methods:

The following methods are specific to `LinkedList` class which are inherited from `Deque` interface:

- `void addFirst(E e)`: Inserts the specified element at the beginning of this list.
- `void addLast(E e)`: Inserts the specified element at the end of this list.
- `E getFirst()`: Retrieves, but does not remove, the first element of this list. This method differs from `peekFirst` only in that it throws an exception if this list is empty.
- `E getLast()`: Retrieves, but does not remove, the last element of this list. This method differs from `peekLast` only in that it throws an exception if this list is empty.
- `E removeFirst()`: Removes and returns the first element from this list.
- `E removeLast()`: Removes and returns the last element from this list.
- `boolean offerFirst(E e)`: Inserts the specified element at the front of this list.
- `boolean offerLast(E e)`: Inserts the specified element at the end of this list.
- `E pollFirst()`: Retrieves and removes the first element of this list, or returns null if this list is empty.

- `E pollLast()`: Retrieves and removes the last element of this list, or returns null if this list is empty.
- `E peekFirst()`: Retrieves, but does not remove, the first element of this list, or returns null if this list is empty.
- `E peekLast()`: Retrieves, but does not remove, the last element of this list, or returns null if this list is empty.

Java LinkedList Real-time Usecases

we will discuss what is the best and what is the worst case scenarios to use `LinkedList` in Java applications. Best Usecase scenario:-

When our frequently used operation is adding or removing elements in the middle of the List, `LinkedList` is the best class to use.

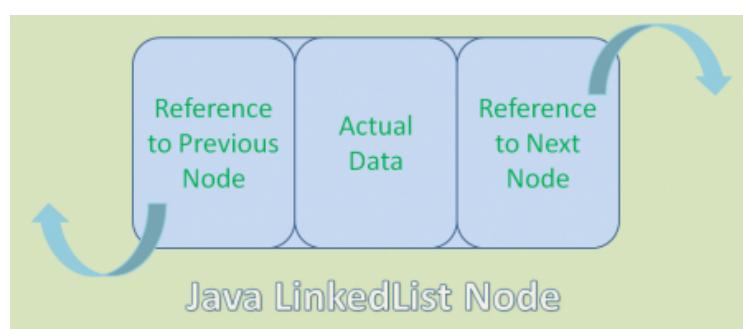
Why? Because we don't need to do more shifts to add or remove elements at the middle of the list. Please refer "How Insertion works in J Java `LinkedList`?" section to understand it in-detail. Worst Usecase scenario:-

When our frequently used operation is retrieving elements from list, then `LinkedList` is the worst choice.

Why? Because `LinkedList` supports only sequential access, does NOT support random access. Please refer "How Deletion works in J Java `LinkedList`?" section to understand it in-detail. NOTE:- `LinkedList` implements `List`, `Deque`, `Cloneable` and `Serializable`. But it does NOT implement `RandomAccess` interface.

Internal Representation of Java `LinkedList`:

As we know, internally Java `LinkedList` is implemented using Doubly Linked List. So `Java LinkedList` represents its elements as Nodes. Each Node is divided into 3 portions as shown below.



Here each Node is used for a specific purpose.

1. Left side Node Part is used to point to the previous Node (Or Element) in the LinkedList.
2. Right side Node Part is used to point to the next Node (Or Element) in the LinkedList.
3. Center Node Part is used to store actual data.

NOTE:- In JVM, LinkedList does NOT store it's elements in consecutive order. It stores it's elements at any available space and they are connected each other using Left and Right side Node portions as shown in the below diagram.



How Insertion works in Java LinkedList?

We have already seen how LinkedList stores it's elements as Nodes in the previous section. In this section, we will discuss about how Java LinkedList's Insertion operation works internally.

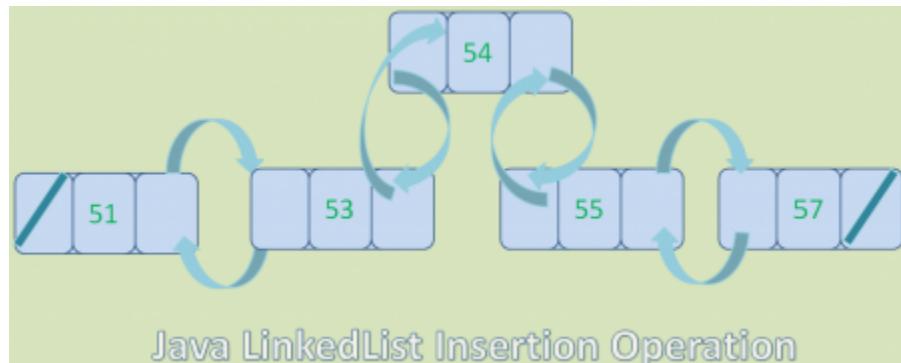
1. Let us assume our initial LinkedList has the following data.



3. Perform the following Insertion operation on this LinkedList

```
linkedList.add(2,54);
```

Here we are trying to perform Insertion operation to add new element with value “54” at index 2.5. Updated LinkedList looks like below.



How Deletion works in Java LinkedList?

We have already seen how LinkedList performs Insertion operation internally in the previous section. In this section, we will discuss about how Java LinkedList’s Deletion operation works internally.

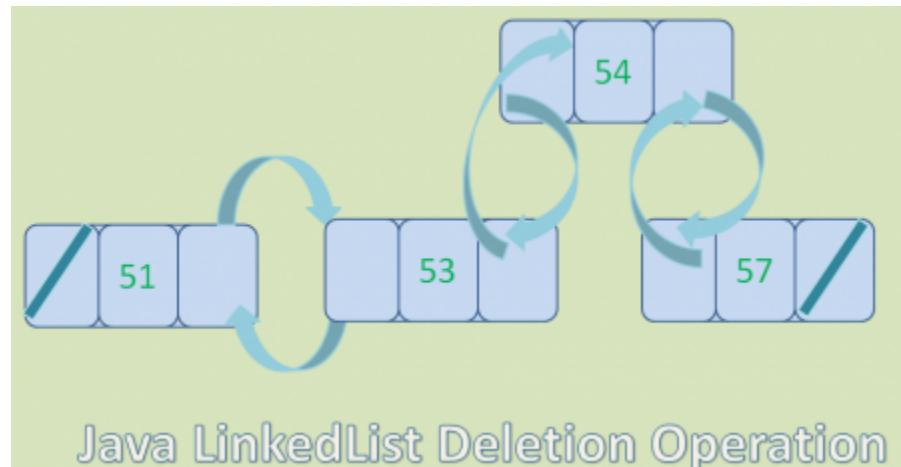
1. Let us assume our initial LinkedList has the following data.



3. Perform the following Insertion operation on this LinkedList

```
linkedList.remove(3);
```

Here we are trying to perform Deletion operation to delete an element which is available at index 3.5. Updated LinkedList looks like below.



Java LinkedList Basic Example:

```
import java.util.LinkedList;
import java.util.List;

public class LinkedListDemo
{
    public static void main(String[] args)
    {
        List names = new LinkedList();
        names.add("Rams");
        names.add("Posa");
        names.add("Chinni");
        names.add(2011);
```

```

        System.out.println("LinkedList content: " + names);
        System.out.println("LinkedList size: " + names.size());
    }
}

```

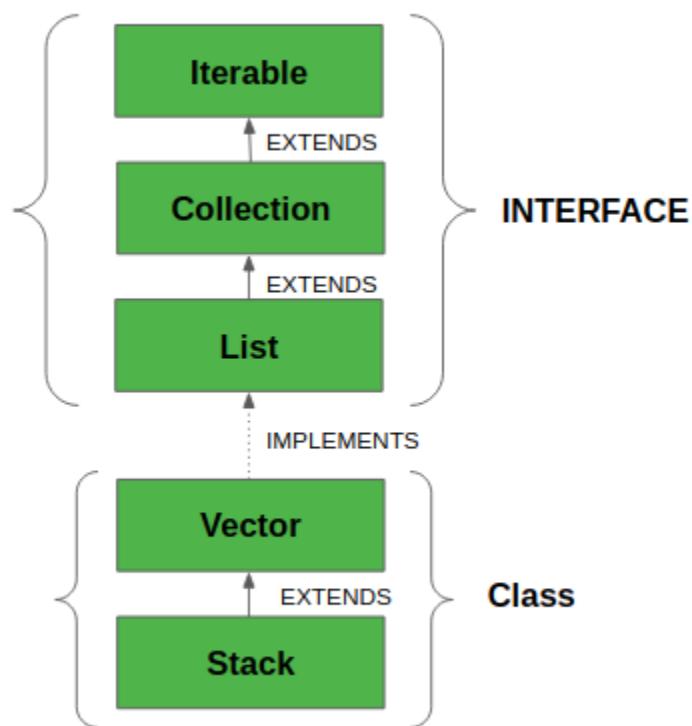
Output:-

LinkedList content: [Rams, Posa, Chinni, 2011]

LinkedList size: 4

Vector Class in java

The Vector class implements a growable array of objects. Vectors fall in legacy classes, but now it is fully compatible with collections. It is found in `java.util` package and implement the List interface, so we can use all the methods of the List interface as shown below as follows:



- Vector implements a dynamic array which means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index.
- They are very similar to ArrayList, but Vector is synchronized and has some legacy methods that the collection framework does not contain.
- It also maintains an insertion order like an ArrayList. Still, it is rarely used in a non-thread environment as it is synchronized, and due to this, it gives a poor performance in adding, searching, deleting, and updating its elements.
- The Iterators returned by the Vector class are fail-fast. In the case of concurrent modification, it fails and throws the ConcurrentModificationException.

Syntax:

```
public class Vector<E> extends AbstractList<E> implements List<E>, RandomAccess,  
Cloneable, Serializable
```

Here, E is the type of element.

- It extends AbstractList and implements List interfaces.
- It implements Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess interfaces.
- The directly known subclass is Stack.
- Important points regarding the Increment of vector capacity are as follows:

If the increment is specified, Vector will expand according to it in each allocation cycle. Still, if the increment is not specified, then the vector's capacity gets doubled in each allocation cycle. Vector defines three protected data members:

- int capacity Increment: Contains the increment value.
- int elementCount: Number of elements currently in the vector stored in it.
- Object elementData[]: Array that holds the vector is stored in it.

Common Errors in the declaration of Vectors are as follows:

- Vector throws an IllegalArgumentException if the InitialSize of the vector defined is negative.
- If the specified collection is null, It throws NullPointerException.

Constructors

1. `Vector():` Creates a default vector of the initial capacity of 10.

```
Vector<E> v = new Vector<E>();
```

2. `Vector(int size):` Creates a vector whose initial capacity is specified by size.

```
Vector<E> v = new Vector<E>(int size);
```

3. `Vector(int size, int incr):` Creates a vector whose initial capacity is specified by size and increment is specified by incr. It specifies the number of elements to allocate each time a vector is resized upward.

```
Vector<E> v = new Vector<E>(int size, int incr);
```

4. `Vector(Collection c):` Creates a vector that contains the elements of collection c.

```
java-collection-framework-fundamentals-self-paced
```

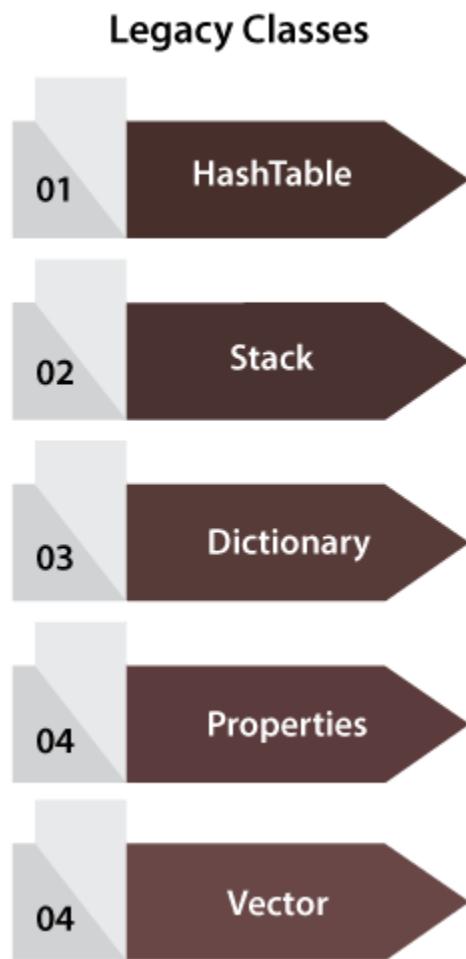
```
Vector<E> v = new Vector<E>(Collection c);
```

Legacy Class in Java:

In the past decade, the Collection framework wasn't included in Java. In the early version of Java, we have several classes and interfaces which allow us to store objects. After adding the Collection framework in JSE 1.2, for supporting the collections framework, these classes were re-engineered. So, classes and interfaces that formed the collections framework in the older version of Java are known as Legacy classes. For supporting generics in JDK5, these classes were re-engineered.

All the legacy classes are synchronized. The `java.util` package defines the following legacy classes:

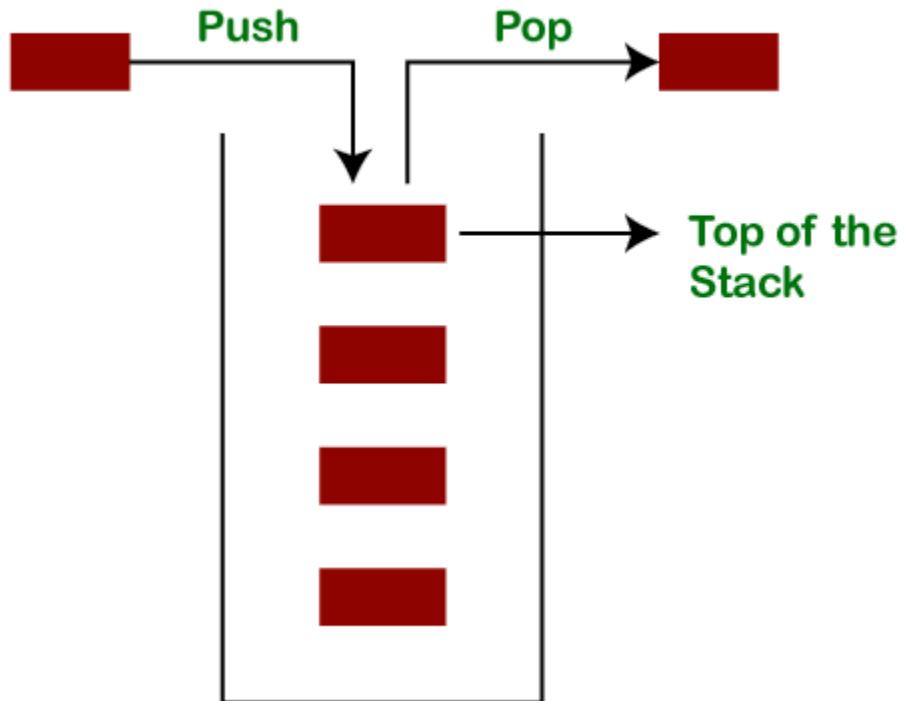
1. `HashTable`
2. `Stack`
3. `Dictionary`
4. `Properties`
5. `Vector`



Java Stack Class

The stack is a linear data structure that is used to store the collection of objects. It is based on Last-In-First-Out (LIFO). Java collection framework provides many interfaces and classes to store the collection of objects. One of them is the Stack class that provides different operations such as push, pop, search, etc.

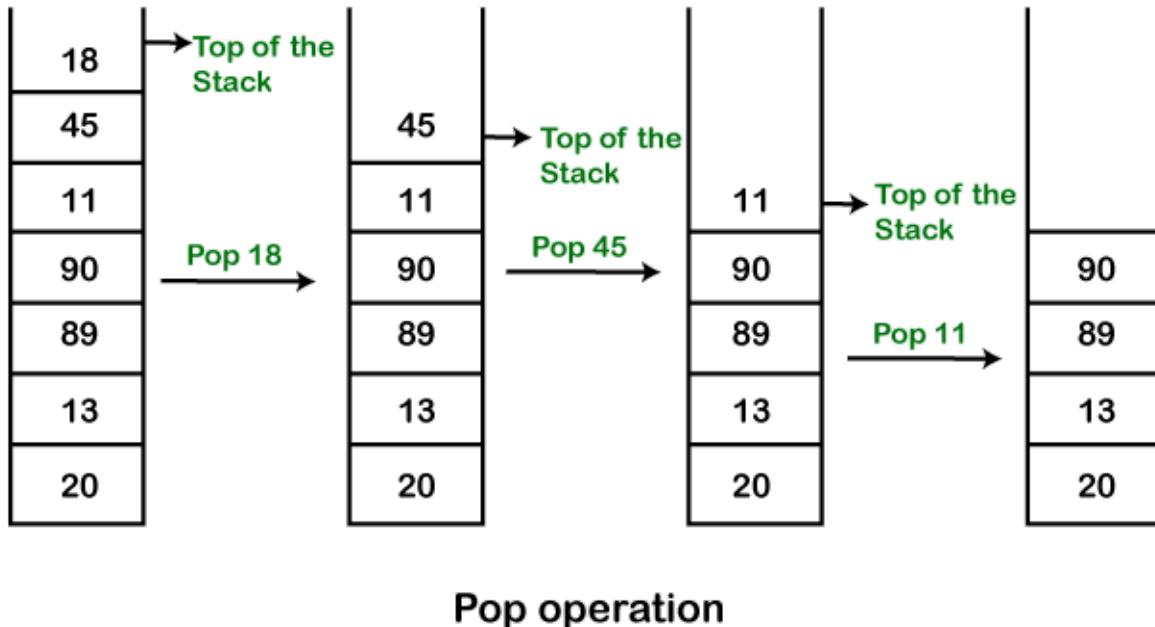
The stack data structure has the two most important operations that are push and pop. The push operation inserts an element into the stack and pop operation removes an element from the top of the stack. Let's see how they work on stack.



Let's push 20, 13, 89, 90, 11, 45, 18, respectively into the stack.



Let's remove (pop) 18, 45, and 11 from the stack.



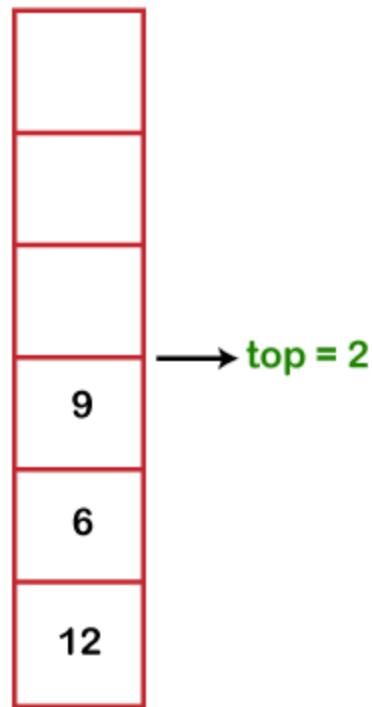
Empty Stack: If the stack has no element is known as an **empty stack**. When the stack is empty the value of the top variable is -1.

Empty Stack

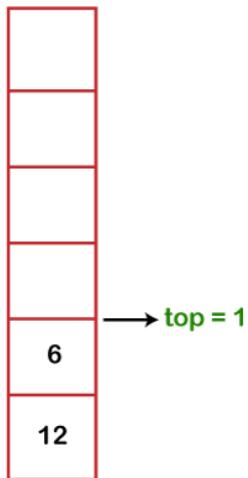


When we push an element into the stack the top is **increased by 1**. In the following figure,

- Push 12, top=0
- Push 6, top=1
- Push 9, top=2



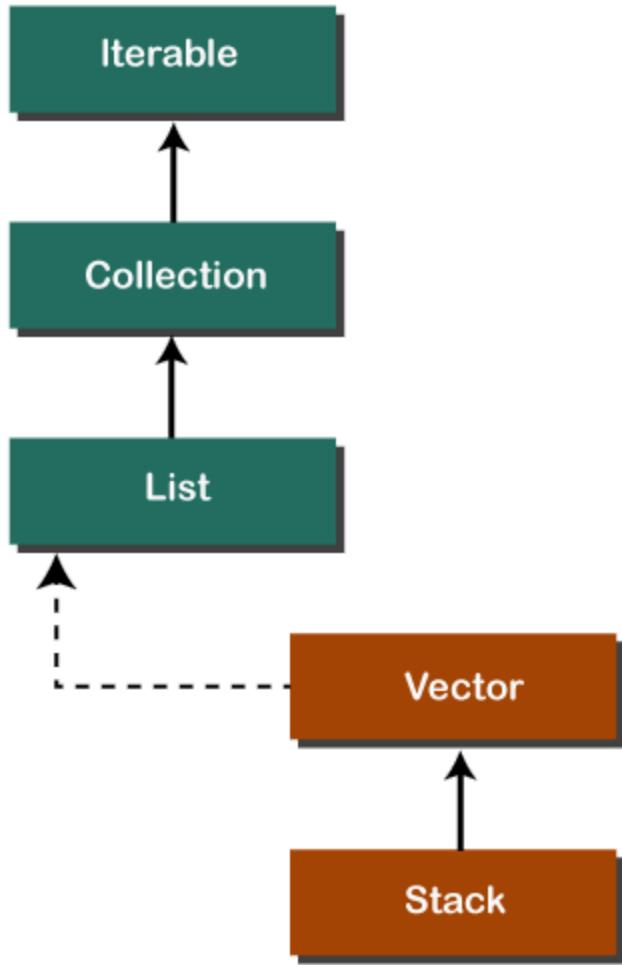
When we pop an element from the stack the value of top is **decreased by 1**. In the following figure, we have popped 9.



The following table shows the different values of the top.

Top value	Meaning
-1	It shows the stack is empty.
0	The stack has only an element.
N-1	The stack is full.
N	The stack is overflow.

In Java, **Stack** is a class that falls under the Collection framework that extends the **Vector** class. It also implements interfaces **List**, **Collection**, **Iterable**, **Cloneable**, **Serializable**. It represents the LIFO stack of objects. Before using the Stack class, we must import the `java.util` package. The stack class arranged in the Collections framework hierarchy, as shown below.



The class supports one default constructor `Stack()` which is used to create an empty stack.

Declaration:

```
public class Stack<E> extends Vector<E>
```

All Implemented Interfaces:

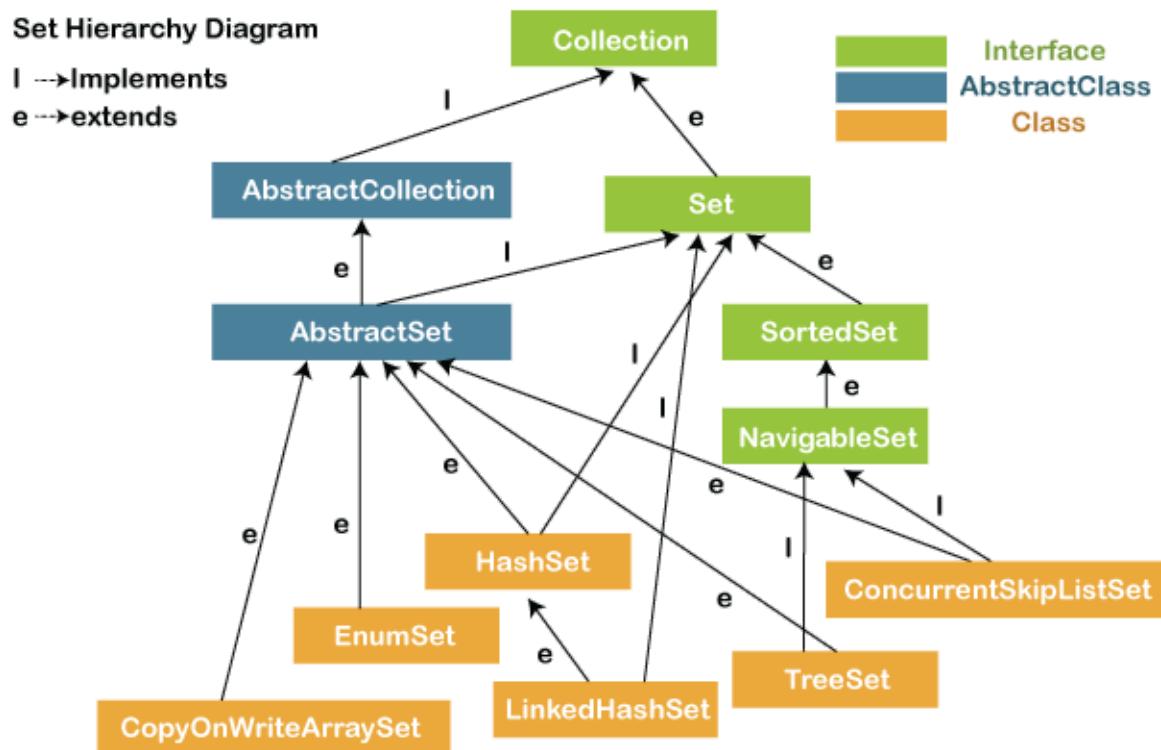
1. Serializable: It is a marker interface that classes must implement if they are to be serialized and deserialized.
2. Cloneable: This is an interface in Java which needs to be implemented by a class to allow its objects to be cloned.
3. Iterable<E>: This interface represents a collection of objects which is iterable – meaning which can be iterated.
4. Collection<E>: A Collection represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired.
5. List<E>: The List interface provides a way to store the ordered collection. It is a child interface of Collection.
6. RandomAccess: This is a marker interface used by List implementations to indicate that they support fast (generally constant time) random access.

Example:

```
import java.util.Stack;
public class StackEmptyMethodExample
{
    public static void main(String[] args)
    {
        //creating an instance of Stack class
        Stack<Integer> stk= new Stack<>();
        // checking stack is empty or not
        boolean result = stk.empty();
        System.out.println("Is the stack empty? " + result);
        // pushing elements into stack
        stk.push(78);
        stk.push(113);
        stk.push(90);
        stk.push(120);
        //prints elements of the stack
        System.out.println("Elements in Stack: " + stk);
        result = stk.empty();
        System.out.println("Is the stack empty? " + result);
    }
}
```

SET Interface

The **set** is an interface available in the **java.util** package. The **set** interface extends the Collection interface. An unordered collection or list in which duplicates are not allowed is referred to as a **collection interface**. The set interface is used to create the mathematical set. The set interface use collection interface's methods to avoid the insertion of the same elements. **SortedSet** and **NavigableSet** are two interfaces that extend the set implementation.



Hashing

We've all used hashing in our daily lives; remember how in university, we were all identified by a unique number called our Roll number? In this case, the Roll number serves as the key, and the student details (name, address, course, age, etc.) serve as the values, and we can easily store this data in an array where the Roll number serves as the index to store the associated details.

However, Hashing can be used when the keys are large or non-integer and cannot be used directly as an index. Hashing in java is primarily the process of converting a given key (roll no) into another value (array indexable integer in this case).

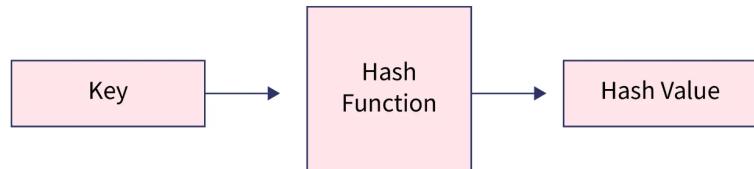
Firstly, hashing in java or in any other language uses a Hash function to convert given keys (Roll no) into some other useful value, called hash-value, based upon the requirement a hash-value can be an integer to be used as an array index, a secret code for some password etc... Then, the associated data (student details associated with particular roll no) is stored in the hash table in the form of <hash-value, associated data> pair.

What is Hashing in Java?

Hashing in Java is the technique that enables us to store the data in the form of key-value pairs, by modifying the original key using the hash function so that we can use these modified keys as the index of an array and store the associated data at that index location in the Hash table for each key.

Hash Function:

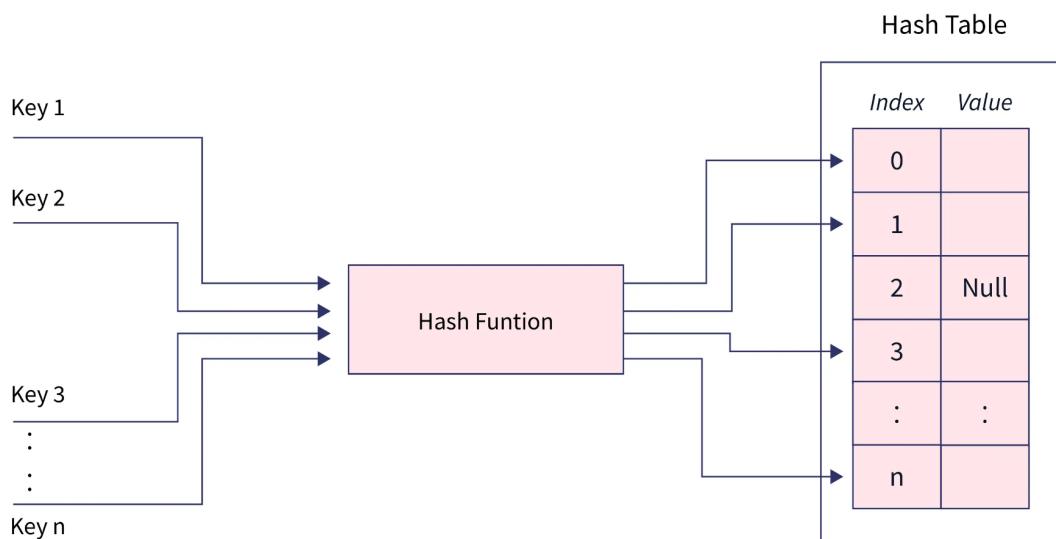
While implementing hashing in java it uses a function called hash function, it is the most important part of hashing; it transforms supplied keys into another fixed-size value (hash-value). The value returned by a hash function is called hash value, hash code, or simply hashes.



SCALER
Topics

Hash table

A hash table is an array that holds pointers to data that corresponds to a hashed key. Hash table uses hash values as the location index to store the associated data in the array.



SCALER
Topics

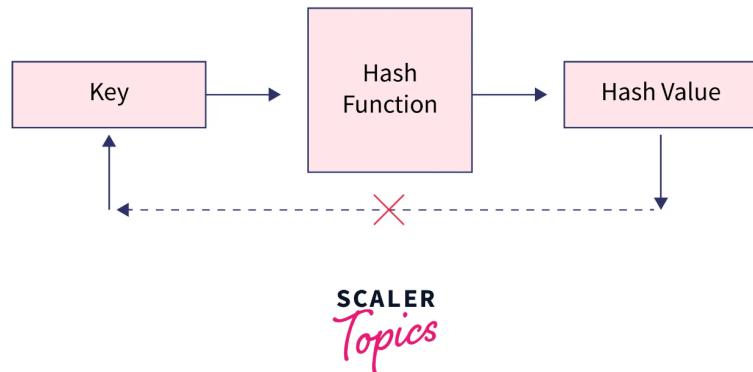
Basically, the given keys are converted into hash values using a hash function and these hash values are used as the index of the hash table to store the associated data.

Hashing in java can be termed as the entire process of storing data in a hash table in the form of key-value pairs, with the key computed using a hash function.

Characteristic of a Hashing Algorithm:

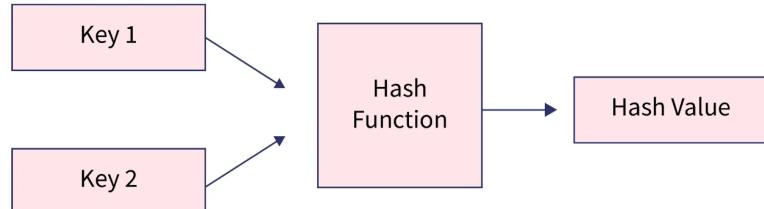
The hashing algorithm is used for generating fixed-length numeric hash-value from the input keys. We anticipate our hash function to have the following features because we created hashing for quick and efficient core operations.

- It should be very fast on its computation and should convert the given key into a hash value quickly.
- The algorithm should avoid generating a hash value from a message's (input) generated hash value (one way).



Collision

Hashing algorithms must avoid the collision. Actually, a collision occurs when two different inputs to the hash function are converted to the same hash value.



Collision

SCALER
Topics

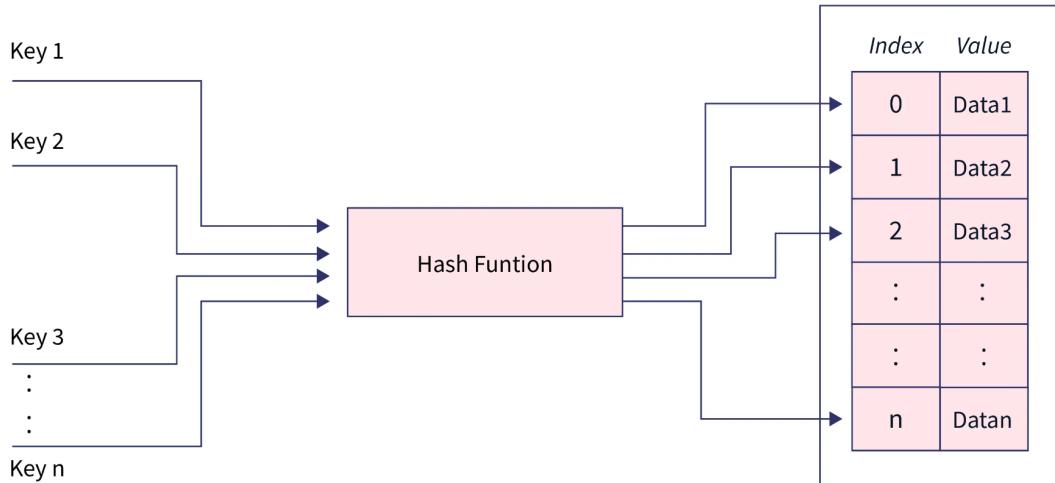
When the message's hash value changes even slightly, the message's hash value must change as well. The avalanche effect is what it's called.

How does Hashing Work?

Hashing in java is a two-step process:

Firstly, A hash function is used to turn an input key into hash values.

This hash-value is used as an index in the hash table and corresponding data is stored at that location in the table.



SCALER
Topics

The element is stored in a hash table and can be retrieved quickly using the hashed key. In order to retrieve data from the hash table, we first calculate the hash value for a given key and as we know this key was used as an index while storing data in the hash table so we extract the data from that index.

To better understand how hashing works, consider the following scenario: we are given the token number and the name of the people who will be vaccinated today. The person is uniquely identified by token no.

The table below contains the name and token numbers as stated above.

Token No.	Name
16	Virat
1	Alex
40	Ishika

5	Sonu
3	Mrinalini
38	John

Our goal is to create a quick and space-efficient hash table for data storage and retrieval.

To solve this problem, one naive solution we can think of is to use an array of size 41 so that we can be able to use each key(token no) as the index to the array and store the data at those index locations.

This works but it is inefficient and we will be wasting the majority of the space we have used because we will be having data stored at only six (1,5,3,16,38 and 40) locations out of 41. we should think of a method to narrow down the search space for us.

In such cases, we can use hashing. We can think of a function that is able to convert the given keys (token no) into less spread hashed keys and follows all the characteristics of the hash function discussed in the section above.

Choosing hash function

If we look closely at the keys, we can see that they can easily be converted to numbers from 0 to 10 if we use

$$\text{Hash(key)} = \text{key} \% 10.$$

Using this hash function we can observe that

- $\text{Hash}(16) = 16\%10 = 6$, indicating that the value corresponding to key 16 i.e. (virat) will be stored in the array at index 6.
- Similarly other keys can be hashed in the same way to find a suitable location in the array.
- Our hash table should be of size 10 because the hash function can be able to give hash values from 0 to 9.

Token No.	Name	$\text{Hash(key)} = \text{key}\%10$
16	Virat	$16\%10 = 6$
1	Alex	$1\%10 = 1$
40	Ishika	$40\%10 = 0$
5	Sonu	$5\%10 = 5$
3	Mrinalini	$3\%10 = 3$
38	John	$38\%10 = 8$

So our hash-table will look like:

Index	0	1	2	3	4	5	6	7	8	9
Value	Ishika	Alex		Mrinalini		Sonu	Virat		John	

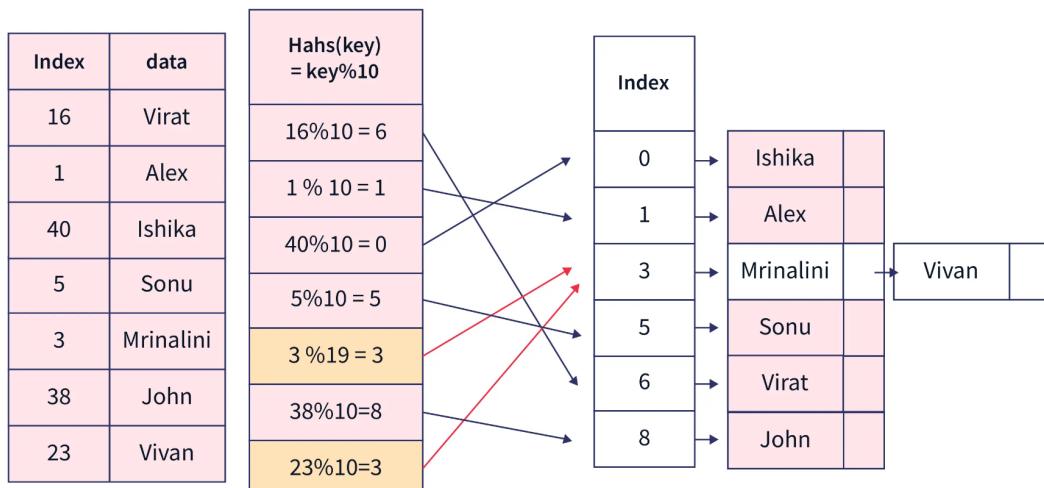
Now if we want to be able to get the name of the person with token no 38, then we can generate an index using the hash function we used above.

$\text{index} = \text{hash}(38) = 38\%10 = 8$.

and we can get the value from the array now, $\text{table}[8]$ will return John as the name of that person.

Now suppose we have one more person's data with token no 23, in that case, the hash function we used above will generate the same hash values for both 23 and 3. This is what we call collision when two different keys are converted to the same hash value by the hash function. We have no. of techniques to resolve collision.

Chaining: Although the hash function should minimize the collision, if it still happens then we can use an array of LinkedList as a hash table to store data, the fundamental idea is for each hash table slot to point to a linked list of records with the same hash value. This technique is called chaining type of hashing in java.



SCALER
Topics

Java HashSet

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing**.
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.

Difference between List and Set

A list can contain duplicate elements whereas Set contains unique elements only.

HashSet class declaration

Let's see the declaration for java.util.HashSet class.

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable,  
Serializable
```

HashSet extends Abstract Set<E> class and implements Set<E>, Cloneable, and Serializable interfaces where E is the type of elements maintained by this set. The directly known subclass of HashSet is LinkedHashSet.

Now for the maintenance of constant time performance, iterating over HashSet requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

- Initial Capacity: The initial capacity means the number of buckets when hashtable (HashSet internally uses hashtable data structure) is created. The number of buckets will be automatically increased if the current size gets full.

- Load Factor: The load factor is a measure of how full the HashSet is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

$$\text{Load Factor} = \frac{\text{Number of stored elements in the table}}{\text{Size of the hash table}}$$

If internal capacity is 16 and the load factor is 0.75 then the number of buckets will automatically get increased when the table has 12 elements in it.

Effect on performance: Load factor and initial capacity are two main factors that affect the performance of HashSet operations. A load factor of 0.75 provides very effective performance with respect to time and space complexity. If we increase the load factor value more than that then memory overhead will be reduced (because it will decrease internal rebuilding operation) but, it will affect the add and search operation in the hashtable. To reduce the rehashing operation we should choose initial capacity wisely. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operation will ever occur.

Note: The implementation in a HashSet is not synchronized, in the sense that if multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be “wrapped” using the Collections.synchronizedSet method. This is best done at creation time, to prevent accidental unsynchronized access to the set as shown below:

Syntax: Declaration of HashSet

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable
```

Constructors of HashSet class

In order to create a HashSet, we need to create an object of the HashSet class. The HashSet class consists of various constructors that allow the possible creation of the HashSet. The following are the constructors available in this class.

1. `HashSet():` This constructor is used to build an empty HashSet object in which the default initial capacity is 16 and the default load factor is 0.75. If we wish to create an empty HashSet with the name hs, then, it can be created as:

```
HashSet<E> hs = new HashSet<E>();
```

2. `HashSet(int initialCapacity):` This constructor is used to build an empty HashSet object in which the initialCapacity is specified at the time of object creation. Here, the default loadFactor remains 0.75.

```
HashSet<E> hs = new HashSet<E>(int initialCapacity);
```

3. `HashSet(int initialCapacity, float loadFactor):` This constructor is used to build an empty HashSet object in which the initialCapacity and loadFactor are specified at the time of object creation.

```
HashSet<E> hs = new HashSet<E>(int initialCapacity, float loadFactor);
```

4. `HashSet(Collection):` This constructor is used to build a HashSet object containing all the elements from the given collection. In short, this constructor is used when any conversion is needed from any Collection object to the HashSet object. If we wish to create a HashSet with the name hs, it can be created as:

```
HashSet<E> hs = new HashSet<E>(Collection C);
```

How HashSet Works Internally In Java?

HashSet internally uses **HashMap** to store its elements. Whenever you create a HashSet object, one **HashMap** object associated with it is also created. This HashMap object is used to store the elements you enter in the HashSet. The elements you add into HashSet are stored as **keys** of this HashMap object. The value associated with those keys will be a **constant**. In this post, we will see how HashSet works internally in Java with an example.

Every constructor of `HashSet` class internally creates one `HashMap` object. You can check this in the source code of `HashSet` class. Below is the some sample code of the constructors of `HashSet` class.

```
private transient HashMap<E, Object> map;

//Constructor - 1

public HashSet()
{
    map = new HashMap<>();           //Creating internally backing HashMap object
}

//Constructor - 2

public HashSet(Collection<? extends E> c)
{
    map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));   //Creating
internally backing HashMap object
    addAll(c);
}

//Constructor - 3

public HashSet(int initialCapacity, float loadFactor)
{
    map = new HashMap<>(initialCapacity, loadFactor);      //Creating internally
backing HashMap object
}

//Constructor - 4

public HashSet(int initialCapacity)
{
    map = new HashMap<>(initialCapacity);        //Creating internally backing
HashMap object
}
```

You can notice that each and every constructor internally creates one new `HashMap` object.

Whenever you insert an element into `HashSet` using `add()` method, it actually creates an entry in the internally backing `HashMap` object with element you have specified as its key and constant called “**PRESENT**” as its value. This “**PRESENT**” is defined in the `HashSet` class as below.

```
// Dummy value to associate with an Object in the backing Map  
private static final Object PRESENT = new Object();
```

Let's have a look at `add()` method of `HashSet` class.

```
public boolean add(E e)  
{  
    return map.put(e, PRESENT)==null;  
}
```

You can notice that, `add()` method of `HashSet` class internally calls `put()` method of backing `HashMap` object by passing the element you have specified as a key and constant “**PRESENT**” as its value.

`remove()` method also works in the same manner.

```
public boolean remove(Object o)  
{  
    return map.remove(o)==PRESENT;  
}
```

Let's see one example of `HashSet` and how it maintains `HashMap` internally.

```
public class HashSetExample  
{  
    public static void main(String[] args)  
    {  
        //Creating One HashSet object
```

```
HashSet<String> set = new HashSet<String>();

//Adding elements to HashSet

set.add("RED");

set.add("GREEN");

set.add("BLUE");

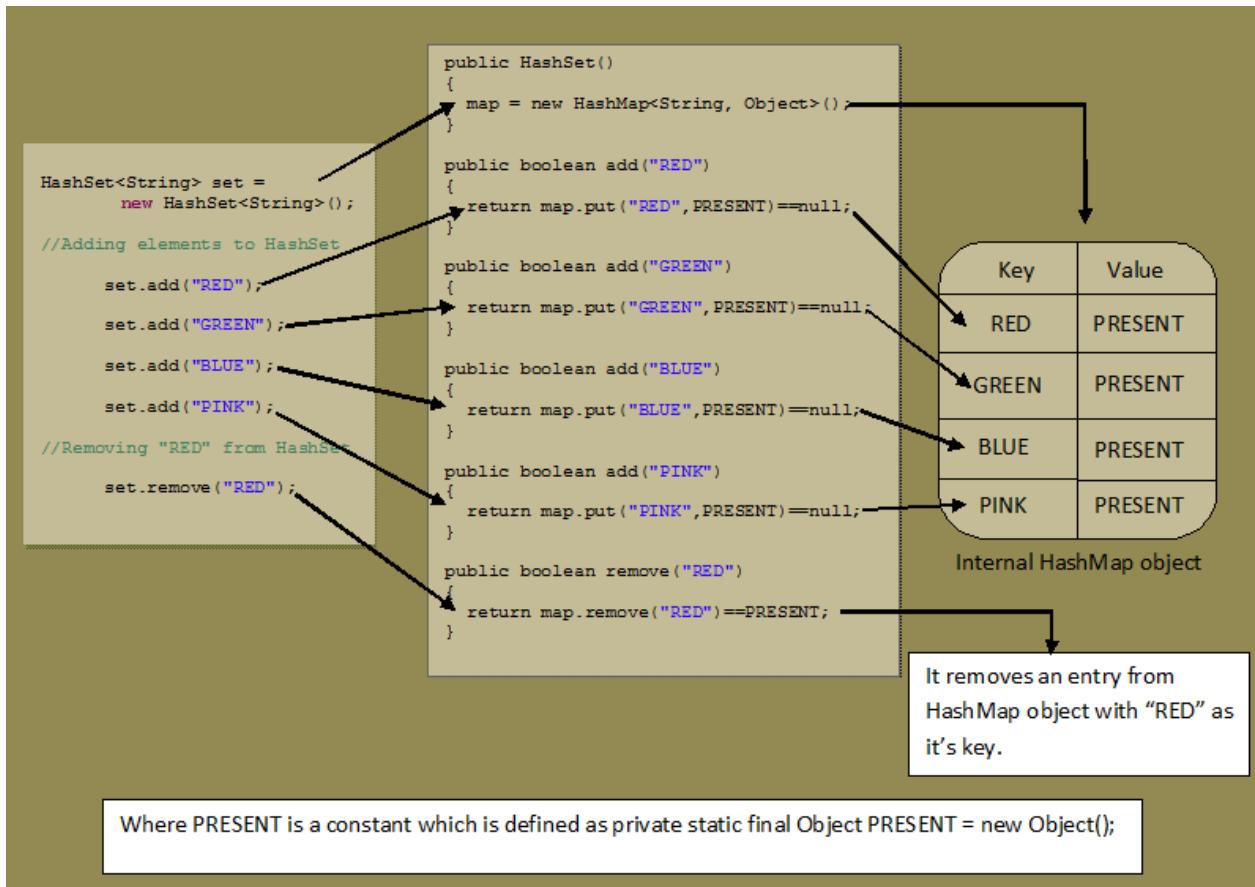
set.add("PINK");

//Removing "RED" from HashSet

set.remove("RED");
}

}
```

See the below picture how the above program works internally. You can observe that the internal HashMap object contains elements of HashSet as keys and constant “PRESENT” as their value.



In the same manner, all methods of HashSet class process internally backing HashMap object to get the desired result.

1. Why do we use HashSet in Java?

We use HashSet when we want a data structure that cannot contain any duplicate values and the values can be accessed in any order (i.e) the order of retrieval doesn't matter.

2. Is HashSet ordered in Java?

No, HashSet is unordered in Java. When we insert a new element to a HashSet, it doesn't guarantee the insertion order and when we iterate its elements, they can be accessed in any order.

3. How HashSet works internally in Java?

- HashSet internally uses HashMap to store the elements. When we insert an element to it using the add() method, the element is internally stored in a HashMap.
- For example, when we add 5 to HashSet via hashSet.add(5), it will be internally stored in HashMap via map.put(5, new Object()). Here, 5 is the key and new Object() is the value of the map entry.

4. Which is faster, HashMap or HashSet?

- HashSet internally uses HashMap to store elements. So there shouldn't be much difference in performance between HashMap and HashSet.
- Both computes hashCode to store objects, and calculating the hashCode of a string or an integer is faster than complex objects with many data members.
- So, the performance of HashMap and HashSet may be affected depending on the objects stored in them.

```
import java.util.*;

// Main class
// HashSetDemo
class Demo {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating an empty HashSet
        HashSet<String> h = new HashSet<String>();

        // Adding elements into HashSet
        // using add() method
        h.add("India");
        h.add("Australia");
```

```
h.add("South Africa");

// Adding duplicate elements
h.add("India");

// Displaying the HashSet
System.out.println(h);
System.out.println("List contains India or not:"
    + h.contains("India"));

// Removing items from HashSet
// using remove() method
h.remove("Australia");
System.out.println("List after removing Australia:"
    + h);

// Display message
System.out.println("Iterating over list:");

// Iterating over hashSet items
Iterator<String> i = h.iterator();

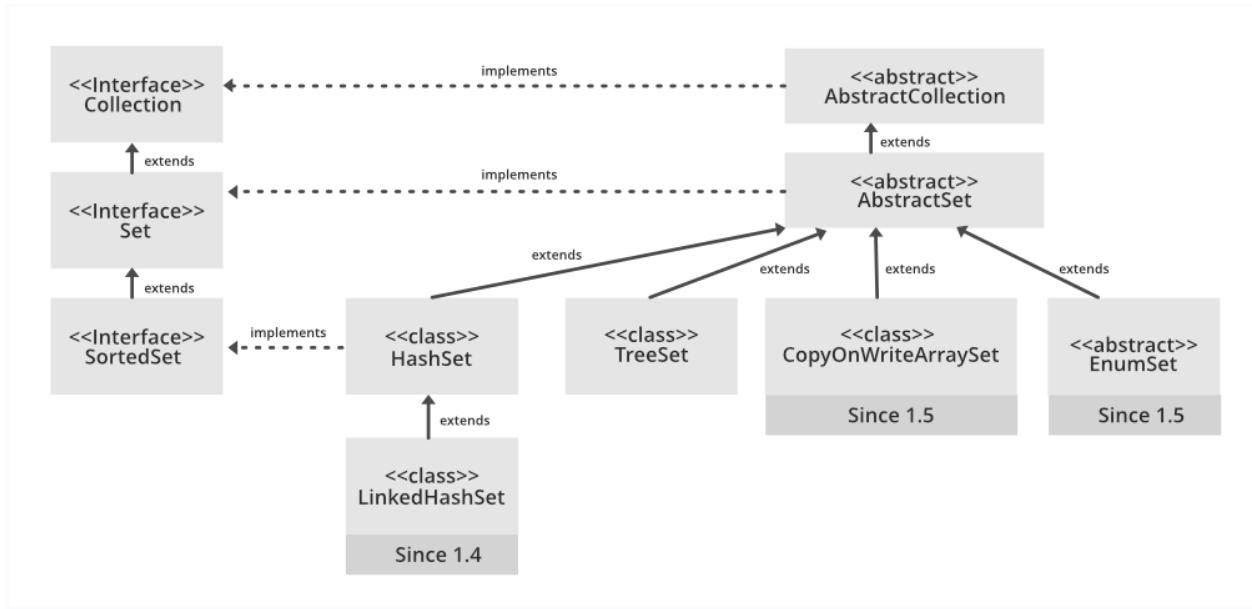
// Holds true till there is single element remaining
while (i.hasNext())

    // Iterating over elements
    // using next() method
    System.out.println(i.next());
}
}
```

LinkedHashSet In Java

The LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements. When the iteration order is needed to be maintained this class is used. When iterating through a HashSet the order is unpredictable, while a LinkedHashSet lets us iterate through the elements in the order in which they were inserted. When cycling through LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.

The Hierarchy of LinkedHashSet is as follows:



All Implemented Interfaces are as listed below:

- Serializable
- Cloneable,
- Iterable<E>
- Collection<E>
- Set<E>

Syntax: Declaration

```
public class LinkedHashSet<E> extends HashSet<E> implements Set<E>,  
Cloneable, Serializable
```

- Contains unique elements only like HashSet. It extends the HashSet class and implements the Set interface.
- Maintains insertion order.

Constructors of LinkedHashSet Class

1. `LinkedHashSet()`: This constructor is used to create a default HashSet

```
LinkedHashSet<E> hs = new LinkedHashSet<E>();
```

2. `LinkedHashSet(Collection C)`: Used in initializing the HashSet with the elements of the collection C.

```
LinkedHashSet<E> hs = new LinkedHashSet<E>(Collection c);
```

3. `LinkedHashSet(int size)`: Used to initialize the size of the LinkedHashSet with the integer mentioned in the parameter.

```
LinkedHashSet<E> hs = new LinkedHashSet<E>(int size);
```

4. `LinkedHashSet(int capacity, float fillRatio)`: Can be used to initialize both the capacity and the fill ratio, also called the load capacity of the LinkedHashSet with the arguments mentioned in the parameter. When the number of elements exceeds the capacity of the hash set is multiplied with the fill ratio thus expanding the capacity of the LinkedHashSet.

```
LinkedHashSet<E> hs = new LinkedHashSet<E>(int capacity, int fillRatio);
```

```
import java.util.LinkedHashSet;

// Main class
// LinkedHashSetExample
public class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating an empty LinkedHashSet of string type
        LinkedHashSet<String> linkedset
            = new LinkedHashSet<String>();

        // Adding element to LinkedHashSet
        // using add() method
        linkedset.add("A");
        linkedset.add("B");
        linkedset.add("C");
        linkedset.add("D");

        // Note: This will not add new element
        // as A already exists
        linkedset.add("A");
        linkedset.add("E");

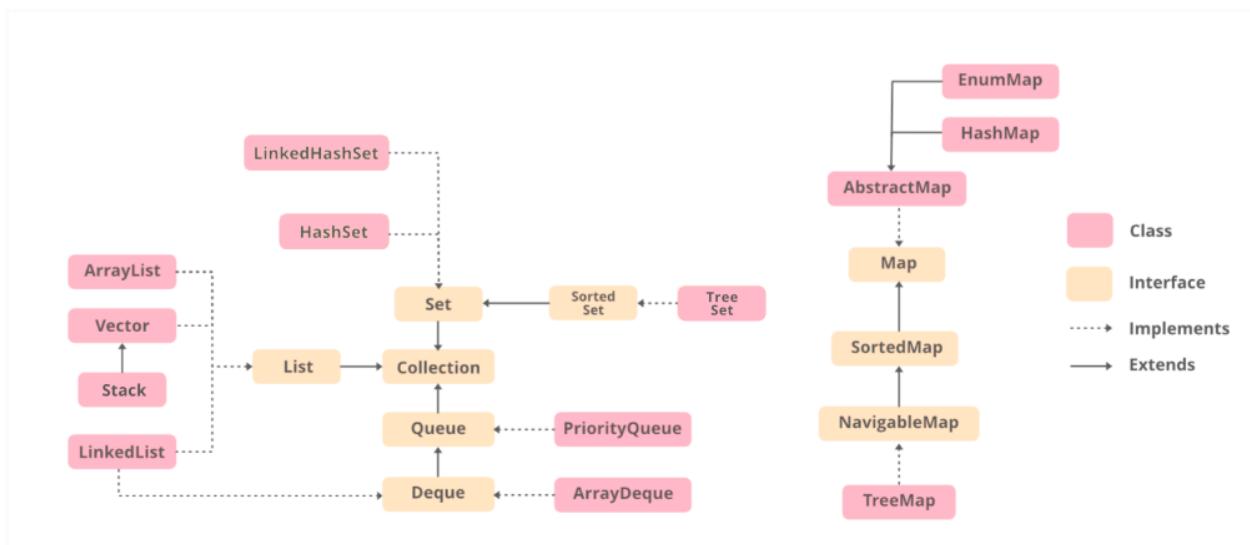
        // Getting size of LinkedHashSet
        // using size() method
        System.out.println("Size of LinkedHashSet = "
            + linkedset.size());

        System.out.println("Original LinkedHashSet:"
            + linkedset);
```

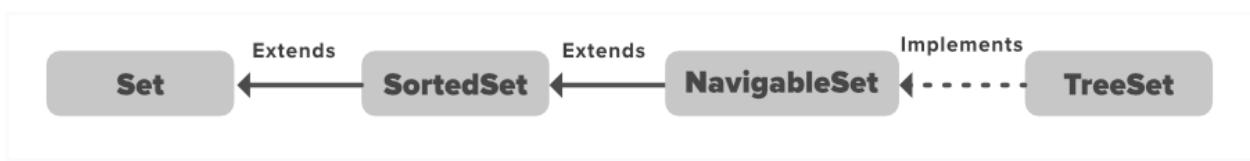
```
// Removing existing entry from above Set  
// using remove() method  
System.out.println("Removing D from LinkedHashSet:  
+ linkedset.remove("D"));  
  
// Removing existing entry from above Set  
// that does not exist in Set  
System.out.println(  
"Trying to Remove Z which is not "  
+ "present: " + linkedset.remove("Z"));  
  
// Checking for element whether it is present inside  
// Set or not using contains() method  
System.out.println("Checking if A is present="  
+ linkedset.contains("A"));  
  
// Now lastly printing the updated LinkedHashMap  
System.out.println("Updated LinkedHashSet: "  
+ linkedset);  
}  
}
```

TreeSet

TreeSet is one of the most important implementations of the SortedSet interface in Java that uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface.



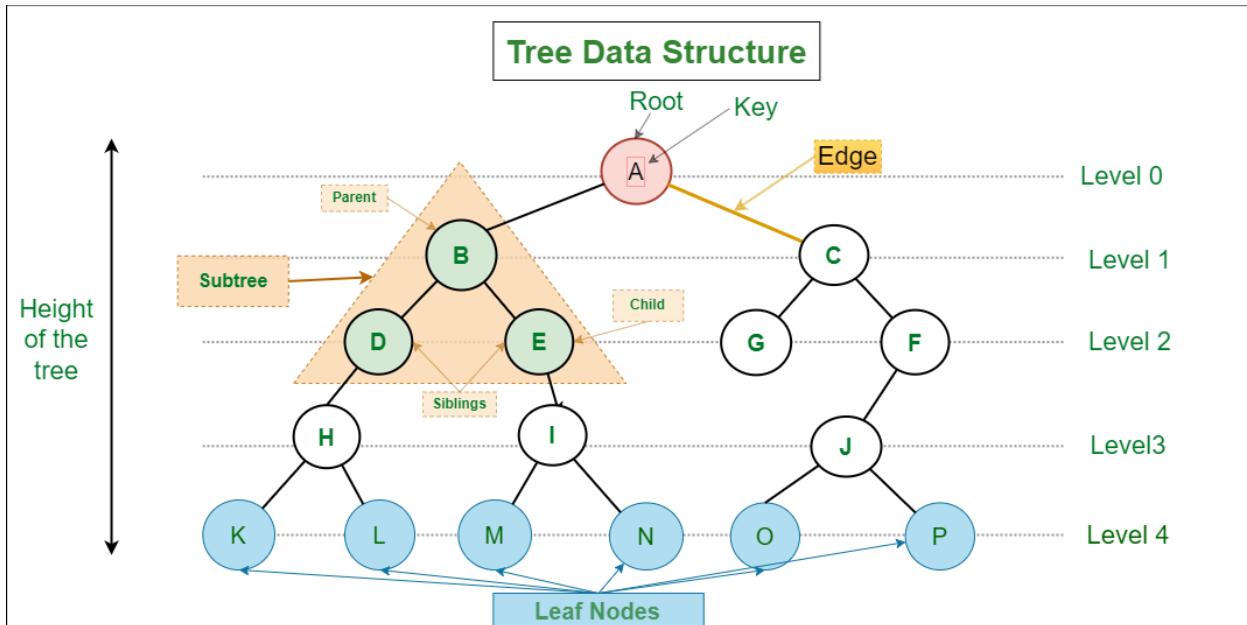
It can also be ordered by a Comparator provided at set creation time, depending on which constructor is used. The TreeSet implements a NavigableSet interface by inheriting the AbstractSet class.



The important points about the Java TreeSet class are:

- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quite fast.
- Java TreeSet class doesn't allow null elements.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains ascending order.
- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quite fast.

- Java TreeSet class doesn't allow null elements.
- Java TreeSet class is non-synchronized.
- Java TreeSet class maintains ascending order.
- The TreeSet can only allow those generic types that are comparable. For example The Comparable interface is being implemented by the StringBuffer class.



Note:

- An object is said to be comparable if and only if the corresponding class implements a Comparable interface.
- String, StringBuffer class and all the Wrapper classes already implement Comparable interface Hence, we DO NOT get a ClassCastException. But if we are creating TreeSet of user defined classes or any Java classes which does not implements comparable interface we will get ClassCastException. to solve this problem we can either implement Comparable to our user defined class or we can pass Comparator object in the Constructor while creating the set.

- For an empty tree-set, when trying to insert null as the first value, one will get NPE from JDK 7. From JDK 7 onwards, null is not at all accepted by TreeSet. However, up to JDK 6, null was accepted as the first value, but any insertion of more null values in the TreeSet resulted in NullPointerException. Hence, it was considered a bug and thus removed in JDK 7.
- TreeSet serves as an excellent choice for storing large amounts of sorted information which are supposed to be accessed quickly because of its faster access and retrieval time.
- The insertion of null values into a TreeSet throws NullPointerException because while insertion of null, it gets compared to the existing elements, and null cannot be compared to any value.

How does TreeSet work Internally?

TreeSet is basically an implementation of a self-balancing binary search tree like Red-Black Tree. Therefore operations like add, remove, and search takes $O(\log(N))$ time. The reason is that in a self-balancing tree, it is made sure that the height of the tree is always $O(\log(N))$ for all the operations. Therefore, this is considered as one of the most efficient data structures in order to store the huge sorted data and perform operations on it. However, operations like printing N elements in the sorted order take $O(N)$ time.

Now let us discuss Synchronized TreeSet prior moving ahead. The implementation of a TreeSet is not synchronized. This means that if multiple threads access a tree set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing some object that naturally encapsulates the set. If no such object exists, the set should be “wrapped” using the Collections.synchronizedSortedSet method. This is best done at the creation time, to prevent accidental unsynchronized access to the set. It can be achieved as shown below as follows:

```
TreeSet ts = new TreeSet();
Set syncSet = Collections.synchronizedSet(ts);
```

Constructors of TreeSet Class are as follows:

In order to create a TreeSet, we need to create an object of the TreeSet class. The TreeSet class consists of various constructors which allow the possible creation of the TreeSet. The following are the constructors available in this class:

TreeSet(): This constructor is used to build an empty TreeSet object in which elements will get stored in default natural sorting order.

Syntax: If we wish to create an empty TreeSet with the name ts, then, it can be created as:

```
TreeSet ts = new TreeSet();
```

TreeSet(Comparator): This constructor is used to build an empty TreeSet object in which elements will need an external specification of the sorting order.

Syntax: If we wish to create an empty TreeSet with the name ts with an external sorting phenomenon, then, it can be created as:

```
TreeSet ts = new TreeSet(Comparator comp);
```

TreeSet(Collection): This constructor is used to build a TreeSet object containing all the elements from the given collection in which elements will get stored in default natural sorting order. In short, this constructor is used when any conversion is needed from any Collection object to TreeSet object.

Syntax: If we wish to create a TreeSet with the name ts, then, it can be created as follows:

```
TreeSet t = new TreeSet(Collection col);
```

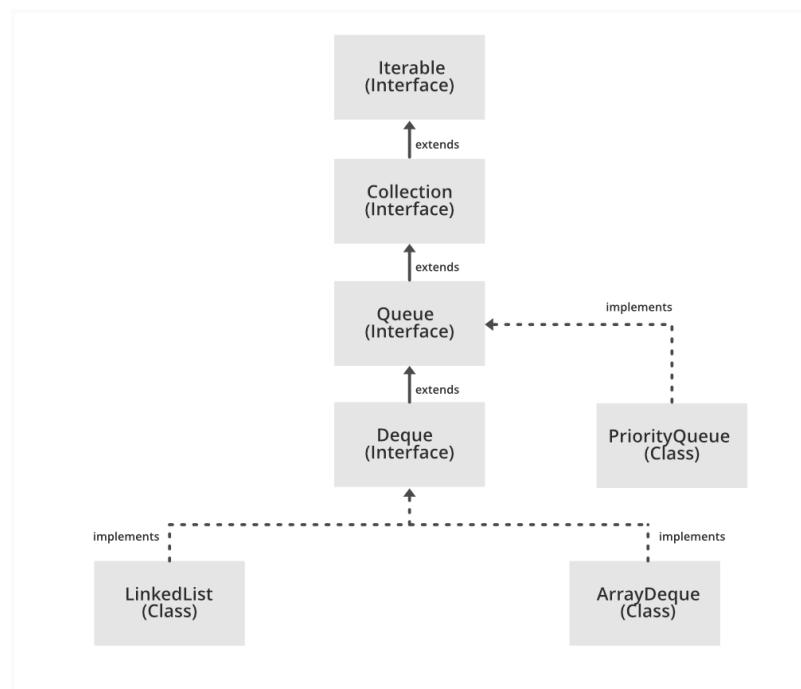
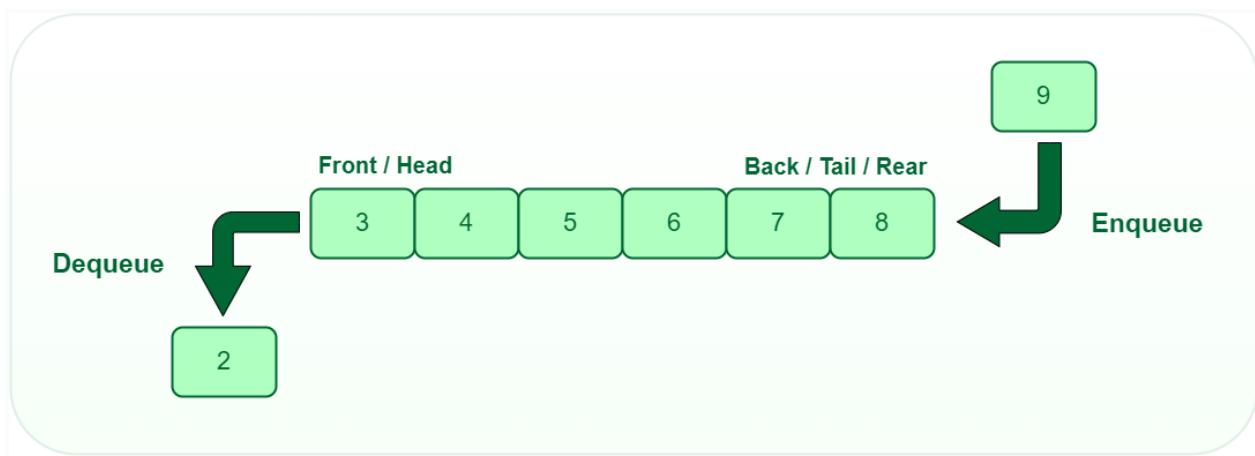
TreeSet(SortedSet): This constructor is used to build a TreeSet object containing all the elements from the given sortedset in which elements will get stored in default natural sorting order. In short, this constructor is used to convert the SortedSet object to the TreeSet object.

Syntax: If we wish to create a TreeSet with the name ts, then, it can be created as follows:

```
TreeSet t = new TreeSet(SortedSet s);
```

Queue Interface

The Queue interface is present in `java.util` package and extends the Collection interface. It is used to hold the elements about to be processed in FIFO(First In First Out) order. It is an ordered list of objects with its use limited to inserting elements at the end of the list and deleting elements from the start of the list, (i.e.), it follows the FIFO or the First-In-First-Out principle.

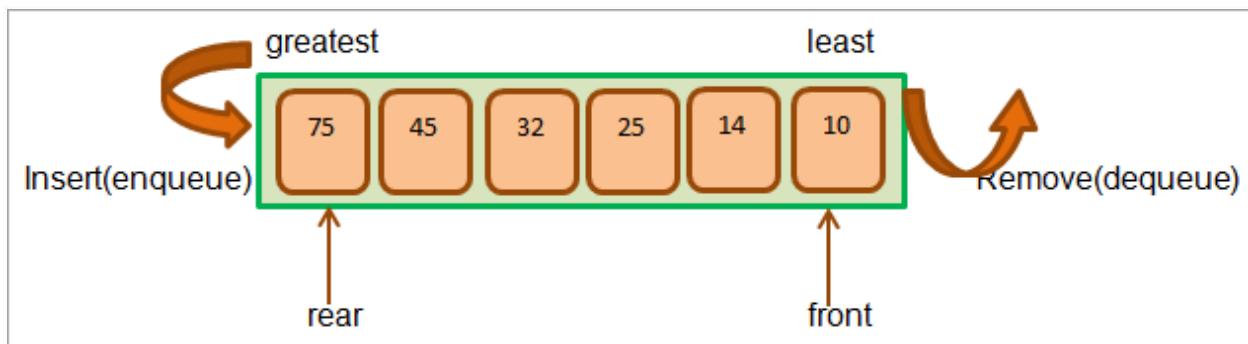


Being an interface the queue needs a concrete class for the declaration and the most common classes are the PriorityQueue and LinkedList in Java.

PriorityQueue in Java

A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a Queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue need to be processed according to the priority, that's when the PriorityQueue comes into play.

The PriorityQueue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.



Declaration:

```
public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable
```

where E is the type of elements held in this queue

A few important points on Priority Queue are as follows:

1. PriorityQueue doesn't permit null.
2. We can't create a PriorityQueue of Objects that are non-comparable
3. PriorityQueue are unbound queues.

4. The head of this queue is the least element with respect to the specified ordering.
If multiple elements are tied for the least value, the head is one of those elements
— ties are broken arbitrarily.
5. Since PriorityQueue is not thread-safe, java provides a PriorityBlockingQueue
class that implements the BlockingQueue interface to use in a java
multithreading environment.
6. The queue retrieval operations poll, remove, peek, and element access the
element at the head of the queue.
7. It provides $O(\log(n))$ time for add and poll methods.
8. It inherits methods from AbstractQueue, AbstractCollection, Collection, and
Object class.

Constructors:

1. PriorityQueue(): Creates a PriorityQueue with the default initial capacity (11) that
orders its elements according to their natural ordering.

```
PriorityQueue<E> pq = new PriorityQueue<E>();
```

2. PriorityQueue(Collection<E> c): Creates a PriorityQueue containing the elements in
the specified collection.

```
PriorityQueue<E> pq = new PriorityQueue<E>(Collection<E> c);
```

3. PriorityQueue(int initialCapacity): Creates a PriorityQueue with the specified initial
capacity that orders its elements according to their natural ordering.

```
PriorityQueue<E> pq = new PriorityQueue<E>(int initialCapacity);
```

4. PriorityQueue(int initialCapacity, Comparator<E> comparator): Creates a
PriorityQueue with the specified initial capacity that orders its elements according to the
specified comparator.

```
PriorityQueue<E> pq = new PriorityQueue(int initialCapacity, Comparator<E>  
comparator);
```

5. PriorityQueue(PriorityQueue<E> c): Creates a PriorityQueue containing the elements
in the specified priority queue.

```
PriorityQueue<E> pq = new PriorityQueue(PriorityQueue<E> c);
```

6. `PriorityQueue(SortedSet<E> c)`: Creates a PriorityQueue containing the elements in the specified sorted set.

```
PriorityQueue<E> pq = new PriorityQueue<E>(SortedSet<E> c);
```

```
import java.util.*;  
  
class PriorityQueueDemo {  
  
    // Main Method  
    public static void main(String args[])  
    {  
        // Creating empty priority queue  
        PriorityQueue<Integer> pQueue = new PriorityQueue<Integer>();  
  
        // Adding items to the pQueue using add()  
        pQueue.add(10);  
        pQueue.add(20);  
        pQueue.add(15);  
  
        // Printing the top element of PriorityQueue  
        System.out.println(pQueue.peek());  
  
        // Printing the top element and removing it  
        // from the PriorityQueue container  
        System.out.println(pQueue.poll());  
  
        // Printing the top element again  
        System.out.println(pQueue.peek());  
    }  
}
```

Deque interface

Deque interface present in `java.util` package is a subtype of the queue interface. The Deque is related to the double-ended queue that supports the addition or removal of elements from either end of the data structure. It can either be used as a queue(first-in-first-out/FIFO) or as a stack(last-in-first-out/LIFO). Deque is the acronym for double-ended queue.

Syntax: The deque interface is declared as:

```
public interface Deque extends Queue
```

Example:

```
import java.util.*;

public class DequeExample {
    public static void main(String[] args)
    {
        Deque<String> deque
            = new LinkedList<String>();

        // We can add elements to the queue
        // in various ways

        // Add at the last
        deque.add("Element 1 (Tail)");

        // Add at the first
        deque.addFirst("Element 2 (Head)");

        // Add at the last
        deque.addLast("Element 3 (Tail)");

        // Add at the first
        deque.push("Element 4 (Head)");
    }
}
```

```

// Add at the last
deque.offer("Element 5 (Tail)");

// Add at the first
deque.offerFirst("Element 6 (Head)");

System.out.println(deque + "\n");

// We can remove the first element
// or the last element.
deque.removeFirst();
deque.removeLast();
System.out.println("Deque after removing "
    + "first and last: "
    + deque);
}
}

```

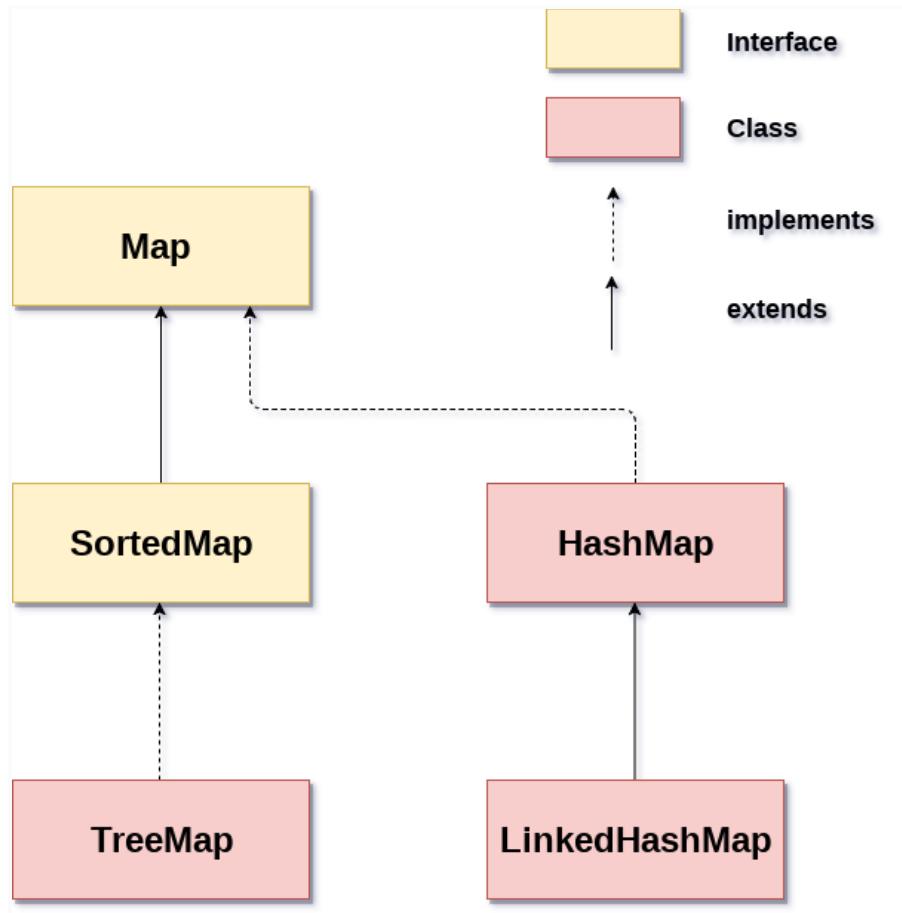
Java Map Interface

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

Java Map Hierarchy

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:



A Map doesn't allow duplicate keys, but you can have duplicate values. `HashMap` and `LinkedHashMap` allow null keys and values, but `TreeMap` doesn't allow any null key or value.

Characteristics of a Map Interface

- A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null values like the `HashMap` and `LinkedHashMap`, but some do not like the `TreeMap`.
- The order of a map depends on the specific implementations. For example, `TreeMap` and `LinkedHashMap` have predictable orders, while `HashMap` does not.
- There are two interfaces for implementing `Map` in java. They are `Map` and `SortedMap`, and three classes: `HashMap`, `TreeMap`, and `LinkedHashMap`.

Map.Entry Interface

Entry is the subinterface of Map. So we will be accessed it by Map.Entry name. It returns a collection-view of the map, whose elements are of this class. It provides methods to get key and value.

Methods of Map.Entry interface

Method	Description
K getKey()	It is used to obtain a key.
V getValue()	It is used to obtain value.
int hashCode()	It is used to obtain hashCode.
V setValue(V value)	It is used to replace the value corresponding to this entry with the specified value.
boolean equals(Object o)	It is used to compare the specified object with the other existing objects.
static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>> comparingByKey()	It returns a comparator that compare the objects in natural order on key.
static <K,V> Comparator<Map.Entry<K,V>> comparingByKey(Comparator<? super K> cmp)	It returns a comparator that compare the objects by key using the given Comparator.
static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V>> comparingByValue()	It returns a comparator that compare the objects in natural order on value.
static <K,V> Comparator<Map.Entry<K,V>> comparingByValue(Comparator<? super V> cmp)	It returns a comparator that compare the objects by value using the given Comparator.

HashMap

HashMap in Java is a part of the collections framework, which is found in the `java.util` package. It provides the basic implementation of the Map interface of Java. It stores the data in a key-value mapping in which every key is mapped to exactly one value of any data type. Keys should be unique as the key is used to retrieve the corresponding value from the map. Since Java 5, it is denoted as `HashMap<K,V>`, where K stands for Key and V for Value.

HashMap in Java is a collection that implements Map interface. `HashMap<K, V>` stores the data in (Key, Value) pairs. Here, keys are unique identifiers used to associate each value on a map.

HashMap is unsynchronized, therefore it's faster and uses less memory than HashTable. Being unsynchronized means HashMap doesn't guarantee any specific order of the elements. Also, HashMap in Java allows adding null keys but there should be only one entry with null as key. However, there can be any number of entries with null as value.

We should choose HashMap over HashTable for an unsynchronized or single-threaded application. Although, since JDK 1.8 HashTable has been deprecated.

Syntax:

The declaration for `java.util.HashMap` class is as follows:

```
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>,  
Cloneable, Serializable
```

The HashMap in Java implements Serializable, Cloneable, Map<K,V> interfaces and it extends AbstractMap<K,V> class.

Parameters:

The HashMap in Java takes two parameters which are as follows:

- K: It is the data type of keys maintained by the map. Keys should be unique.
- V: It is the data type of values maintained. Values need not be unique. It can be duplicated.

Basic Usage of HashMap in Java

HashMap in Java is the implementation of map, which is a key-value mapping. Now, one might question that if we simply want to add values, then why can't we use a list? Why do we need HashMap? The simple reason is performance. To find a specific element in a list, the time complexity is $O(n)$ for insertion and lookup and if the list is sorted, it will be $O(\log n)$ on binary search.

The advantage of HashMap is that the time complexity to insert and retrieve a value is $O(1)$ on average, and space complexity is $O(n)$.

Example: Create HashMap in Java

First, the `java.util.HashMap` package needs to be imported to create a HashMap in Java. After the import is done, we can create HashMap in Java as below:

```
HashMap<K, V> languages = new HashMap<>();
```

In the above code, a hashmap named `languages` is created. Here, K represents the key type and V represents the type of values such as below:

```
HashMap<String, Integer> languages = new HashMap<>();
```

Here, the type of key is `String`, and the type of value is `Integer`.

Example:

```
import java.util.HashMap;

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

```

// create a hashmap
HashMap<String, Integer> languages = new HashMap<>();

// add elements to hashmap
languages.put("Java", 8);
languages.put("JavaScript", 1);
languages.put("Python", 3);
System.out.println("HashMap: " + languages);
}
}

```

The `HashMap` in Java implements `Serializable`, `Cloneable`, `Map` interfaces. It extends `AbstractMap` class.

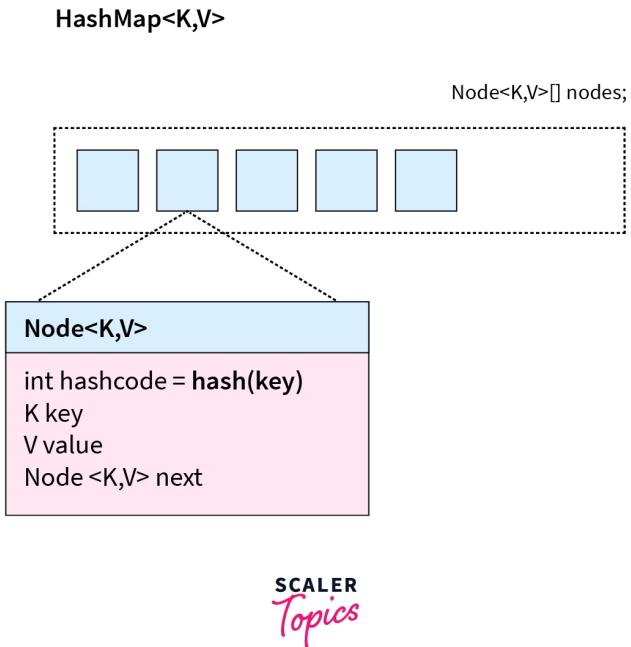
Constructors in `HashMap` Java

`HashMap` in Java has 4 constructors, and each one has public access modifier. The constructors are as follows:

- `HashMap()`
- `HashMap(int initialCapacity)`
- `HashMap(int initialCapacity, float loadFactor)`
- `HashMap(Map map)`

- **Hashing:** `HashMap` in Java works on the principle of hashing – an algorithm to map object data to some representative integer values. Hashing is a process of converting an object into integer form by using the method `hashCode()`. `hashCode()` method of `Object` class returns the memory reference of object in integer form. So the Hash function is applied to the key object to calculate the index of the bucket in order to store and retrieve any key-value pair.

- `HashMap<K,V>` manages an array of `Node<K,V>` objects that will be replaced by another larger array if all of its elements have been assigned value. `Node<K,V>` class consists of 4 fields. The `next` field is a reference to the next `Node` object. It may not be the next element in the array. `HashMap` ensures those `Node(s)` having the same hashcode will have consecutive references. This helps it quickly find all `Node(s)` with the same specified hashcode.



1. Capacity: Capacity is the number of buckets in the HashMap. The initial capacity is the capacity at which the Map is created. The initial default capacity of the HashMap is 16`.
 2. Load Factor: The capacity is expanded as the number of elements in the HashMap increases. The load factor is the measure that decides when to increase the capacity of the Map. The default load factor is 75% of the capacity.
 3. Threshold: The threshold of a HashMap in Java is approximately the product of the current capacity and load factor.
 4. Rehashing: Rehashing is the process of re-calculating the hash code of already stored entries. When the number of entries in the hash table exceeds the threshold value, the Map is rehashed so that it has approximately twice the number of buckets as before.
 5. Collision: A collision occurs when a hash function returns the same bucket location for two different keys.

Let's suppose a HashMap is created with the initial default capacity of 16 and the default load factor of 0.75. So, the threshold is $16 * 0.75 = 12$, which means that it will increase the capacity from 16 to 32 after the 12th entry (key-value pair) is added. This will be done by rehashing.

Features of HashMap in Java:

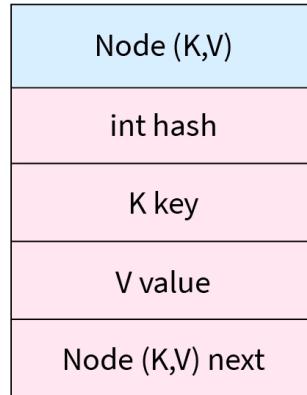
- HashMap in Java uses the technique of Hashing. That's why it is called HashMap. Hashing function maps a big number or string to a small integer that can be used as an index. A shorter value helps in indexing and faster searches.
- HashMap in Java is found in the java.util package.
- In HashMap, the keys should be unique but values can be duplicated. It means that an X key can't contain more than one value that is X mapped to only one value. But other Y, Z keys can contain duplicate or the same values already mapped to X key.
- HashMap in Java can also have null key and values. But there could be only one null key. However, there could be any number of null values corresponding to different keys.
- Unlike HashTable, HashMap is not synchronized. It is an unordered collection that doesn't guarantee any specific order of the elements.
- To retrieve any value from the HashMap, one should know the associated key.
- HashMap extends an abstract class AbstractMap and implements Cloneable and Serializable interfaces.

How does HashMap Work Internally?

As we have discussed, HashMap in Java stores the elements using key-value mapping where we can retrieve an element from the HashMap by its key. The key-value pairs are stored as instances of inner class HashMap. The entry which has key-value mapping stored as attributes where the key has been marked as final (immutable).

Internally, the HashMap is an array of nodes. HashMap makes use of arrays and LinkedList for storing key-value pairs.

Given below is a structure of a node of HashMap that is programmatically represented as a class.



SCALER
Topics

As seen from the node representation above, a node has a structure similar to a linked list node. An array of these nodes is called Bucket. HashMap stores elements in buckets and the number of buckets is called capacity.

When we add a value to the map, the bucket in which the value will be stored is determined by the key's hashCode() method. Basically, a hash value is calculated using the key's hash code. This hash value is used to calculate the index in the array for storing Entry object.

When we try to retrieve the value, the bucket is calculated by the HashMap in the same way – using hashCode(), which actually calculates the first index location. Then in that bucket, it iterates through the objects found and uses the key's equals() method to find the exact match.

To avoid having many buckets with multiple values, the capacity is doubled if 75% (the load factor) of the buckets become non-empty. The default value for the load factor is 75%, and the default initial capacity is 16. Both can be set in the constructor as discussed above.

Performance of HashMap:

The performance of a HashMap is affected by two parameters:

- Initial Capacity: The initial capacity is simply the capacity (number of buckets) during creation.
- Load Factor: The load factor or LF, is a measure of how full the hash map should be after adding some values before it is resized.
- The initial default capacity is 16 and the default load factor is 0.75. We can create a HashMap with custom values as well. However, for custom values, we should understand the performance implications. When the number of hash map entries exceeds the product of LF and capacity, then rehashing occurs i.e. another internal array is created with twice the size of the initial one, and all entries are moved over to new bucket locations in the new array.

A low initial capacity reduces space cost but increases the frequency of rehashing. Rehashing is obviously a very expensive process. So if we have many entries, we should set a considerably high initial capacity. But if we set the initial capacity too high, we will pay the cost in iteration time. So a high initial capacity is good for a large number of entries coupled with little to no iteration. A low initial capacity is good for a few entries with a lot of iteration.

Inserting a value into a HashMap takes, on the average case, $O(1)$ time. The hash function is computed, the bucket is chosen, and then the item is inserted. In the worst-case scenario, all of the elements will have hashed to the same value, which means either the entire bucket list must be traversed or, in the case of open addressing, the entire map must be probed until an empty spot is found. Therefore, in the worst case, insertion takes $O(n)$ time.

Note: From Java 8 onward, Java has started using Self-Balancing BST instead of a linked list for chaining. The advantage of self-balancing BST is, in the worst case (when every key maps to the same slot) search time is $O(\log n)$.

Use cases of HashMap:

- We can use HashMap for the implementation of Phonebook by using key-value pair of Name - Number.
- For Dictionary Application HashMap can be used. As Dictionary will be having a lot of words so words can be used as key which are mapped to values - Meaning, Description of word.
- Wherever you are putting a list of objects in an array or list and then retrieving the value based on some attributes of an object, you can use hashmap. This is used extensively as in-memory cache for static/near static values. All your system properties, static business data - country codes, zip codes, cities, etc. - can be stored in a hashmap and retrieved.
- Even distributed caching systems like Couchbase, membase, redis are sort of hashmaps extended to handle high concurrency and large scale data through replication and distribution.

Important Point about HashMap:

1. HashMap in Java is an implementation of the Map interface. It is a part of the collections framework which is found in `java.util` package.
2. HashMap stores the data in key-value mapping where every key is mapped to a value. Keys can not be duplicated but values can be duplicated.
3. HashMap is unsynchronised, and it doesn't store the elements in a particular order.
4. HashMap allows storing multiple null values and only one null key.
5. HashMap in Java has 4 constructors, and each one has public access modifier.
6. For traversal over any structure of the Collection Framework (like HashMap), we can use the iterator interface.
7. **Real-world application of HashMap example:** In storing the records of the employees of a company where employee ID is stored as a key and employee data is stored as a value in the HashMap. It helps for faster retrieval of employee data corresponding to an employee ID.

HashTable In Java

The Hashtable class in Java is one of the oldest members of the Java Collection Framework. A hash table is an unordered collection of key-value pairs, with a unique key for each value.

In a hash table, data is stored in an array of list format, with a distinct index value for each data value. The hash table provides an efficient combination of retrieval, edit, and delete operations.

The Hashtable class in Java is a concrete implementation of a Dictionary and was originally a part of `java.util` package. The Hashtable class creates a hash table by mapping keys to values.

In a hashtable, any non-null object can be used as a key or as a value. The objects used as keys must implement the `hashCode` and `equals` methods in order to effectively store and retrieve items from a hashtable.

Hashtable is not just a data structure, but also a Java Collection API class. Despite the fact that both the array and hashtable data structures are intended for fast search, i.e. constant time search operation, also known as $O(1)$ search, the fundamental difference between them is that the array requires an index, whereas the hash table requires a key, which could be another object.

This code example generates a hashtable of numbers. It employs the bird numbers as keys:

Example:

```
import java.util.Hashtable;
import java.util.Enumeration;

public class Main {

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

```
Enumeration birds;
String key;

// Creating a Hashtable
Hashtable<String, String> hashtable =
    new Hashtable<String, String>();

// Adding Key and Value pairs to Hashtable
hashtable.put("Bird1","Pigeon");
hashtable.put("Bird2","BlueBird");
hashtable.put("Bird3","Swan");
hashtable.put("Bird4","Parrot");
hashtable.put("Bird5","Sparrow");

birds = hashtable.keys();
while(birds.hasMoreElements()) {
    key = (String) birds.nextElement();
    System.out.println("Key: " +key+ " & Value: " +
        hashtable.get(key));
}
}
```

As depicted above, a hashtable does not guarantee the order of records inserted in it.

Java Hashtable was added in the JDK 1.0 version which is a part of the `java.util.Hashtable` package. Prior to JDK 1.2 version, keys and values were mapped using hashtables. Later on, Java 1.2 version changed the Hashtable such that it now implements the Map interface. Thus, Hashtable has been included in the Java Collections framework. However, since it does not exactly adhere to the Java Collections Framework, it is now considered a "legacy class."

Parameters

Two parameters are accepted by hashtable:

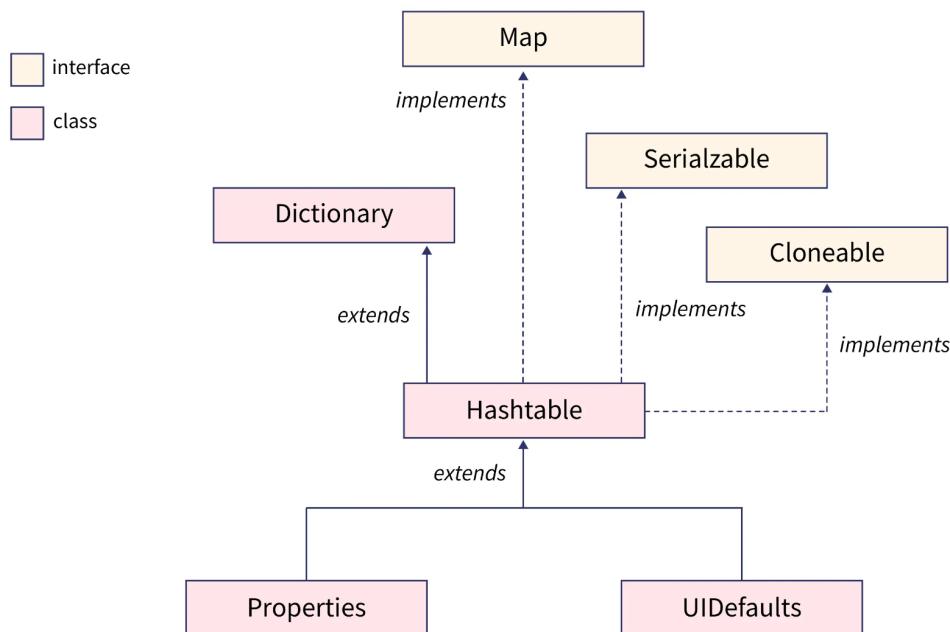
K – the type of keys maintained by this map

V – the type of mapped values

Internal Working of Hashtable

Before getting into how hashtables operate in Java, let's have a look at how they're implemented.

The hashtable class extends the Dictionary class and implements Map, Cloneable, and Serializable interfaces. The following diagram clearly depicts the Hashtable hierarchy in Java.



A Hashtable is an array of a list. Each list is referred to as a bucket that contains the key/value pairs in them. It uses the hashCode() method for identifying which bucket should be assigned to the key/value combination. Hashcode, in general, use a non-negative integer that is equal for equal Objects but may or may not be equal for unequal Objects. The equals() function is used by the hashtable to detect if two items are equal or not.

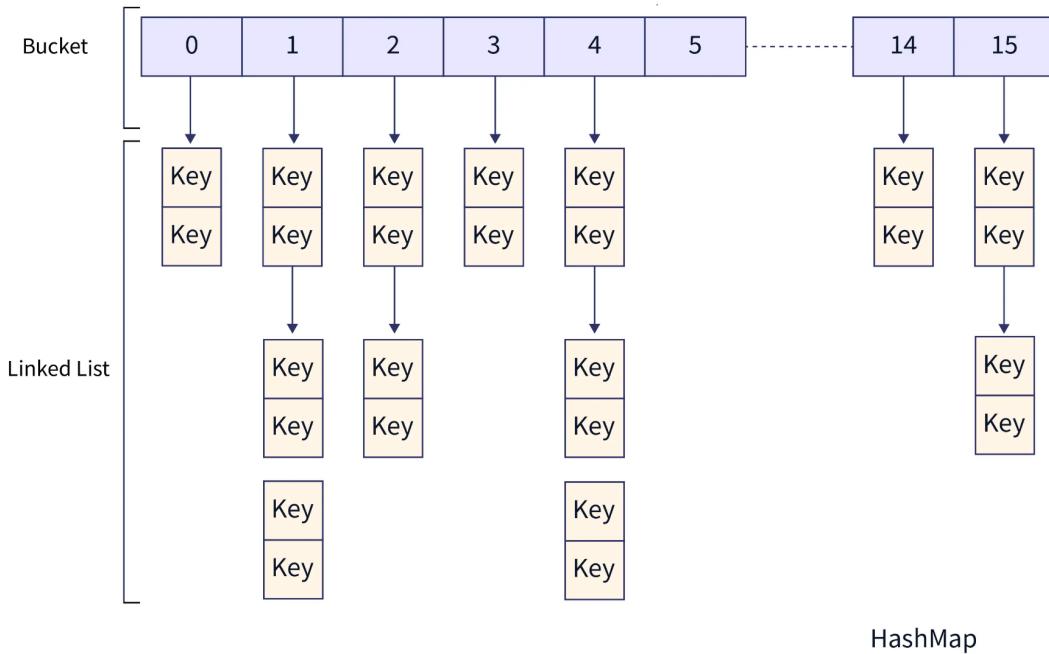
The hash function assists in locating a specific key in the bucket list. In theory, a hash function is a function that creates an address in a table when given a key. For an object, a hash function always returns a number.

Note: It should be noted that the Java Hashtable class includes unique elements and does not support null keys or values. You cannot even attempt to acquire a null key since it would throw a NullPointerException.

Two identical items will always have the same number, however, two unequal objects might not always have distinct numbers. When we add entries into a hashtable, it is possible that different objects (through the equals() function) may have the same hashcode. This is referred to as a collision.

The pairings that map to a single bucket (array index) are saved in a list, and the list reference is saved in the array index.

Hashtables are simple to understand when visualized. Hash tables are often created as arrays of linked lists. If we consider a table that stores city's names, after a few insertions it might be laid out in memory as below, where ()-enclosed integers represent hash values of the text entered as the city's name.

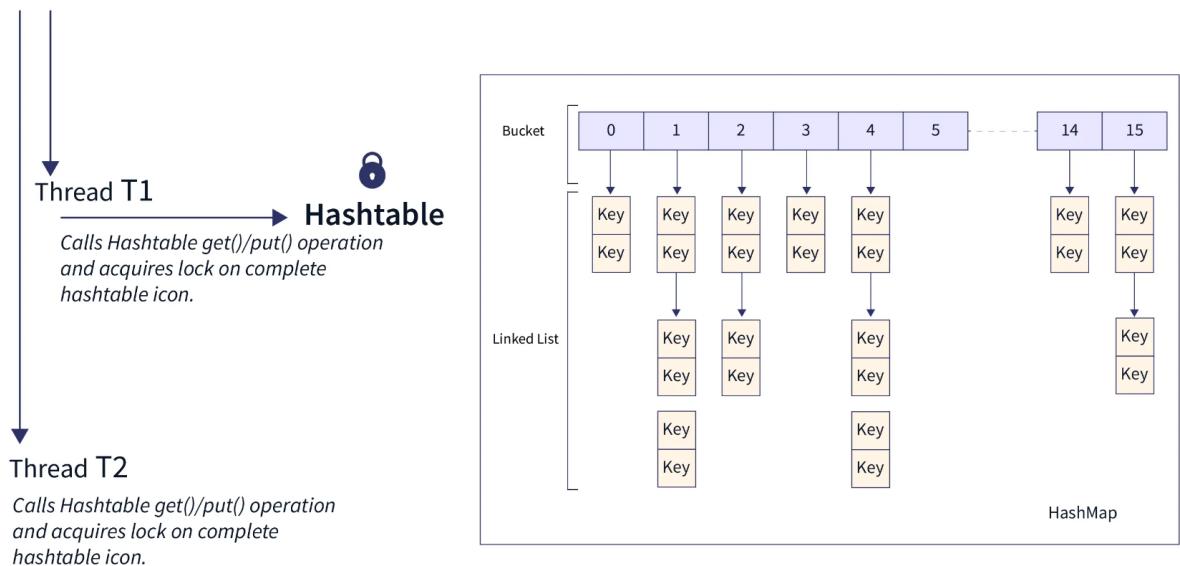


SCALER
Topics

- Each entry in the array (at indices [0], [1], and so on) is called a bucket. It begins with a potentially empty linked list containing key-value pairs which in this case is key mapped to city names.
- Each value (e.g. "Ontario" with hash 12) is linked from a bucket which is identified by the formula: $\text{Hash_number \% Number of buckets} = \text{Index of Bucket}$
 $12 \% 10 == [2]$ Here, % is the modulo operator, which gives the remainder when the hash number is divided by the number of buckets. So, the city name with hash number 12 will be placed in a bucket with the index number 2.
- Multiple data values may collide at and be linked from the same bucket, often due to the fact that their hash values collide after calculating the modulo operation (e.g. $12 \% 10 == [2]$, and $42 \% 10 == [2]$). Most hash tables manage collisions by comparing the whole value (here text) of a value being searched or added to each value existing in the linked list at the hashed-to bucket. This results in somewhat decreased speed but no functional confusion.

Hashtable is synchronized and offers thread safety comparable to concurrentHashMap. However, from the perspective of performance, Hashtable writes operations employ hashtable wide lock, which locks the whole hashtable object.

Look at the below illustration for a better understanding.



SCALER
Topics

To get the lock on the whole hashtable object, thread T1 runs the get() and put() operations on the hashtable. As a result, if Thread T2 performs get() or put(), it must wait until Thread T1 completes the operation and releases the lock on the object.

Features of Hashtable in Java

In Java, a hashtable is a component of the collections framework in which it implements a Map interface. For backward compatibility, it inherits from the obsolete and utterly useless Dictionary class.

After learning what a hashtable in Java is and how it operates, let's examine some of its key features:

- Key Based Hashtables need a key to search the value, which might be another object, in contrast to arrays, which require an index.
- Dynamic Capacity By employing chaining and linked lists, the hashtable may store more items than the internal array can hold.
- Performance Hashtable performance may be $O(n)$ in the worst scenario when you need to go through a linked list in order to identify the proper value object due to a collision, despite the fact that hashtables are designed for fast search, that is, constant time search operations, commonly known as $O(1)$ search.

Note: This has been somewhat improved now when JDK utilizes binary tree rather than linked lists from Java 8 and worst-case performance is now pegged to $O(\log N)$.

- Usage Hashtables stores mapping, or pairings of key-value objects. As a result, it may be used to hold information on clients, library books, or other records.
- Sorting Although a Hashtable cannot be sorted, sorted data can be extracted by sorting the list of keys from the Hash table and retrieving values in that order by using a TreeMap or LinkedHashMap.
- Collision As different data items may collide at and be associated from the same bucket, most often because their hash values conflict, collisions are possible in hashtables. This can be handled by using various collision avoidance algorithms like open addressing or chaining.
- Synchronization Since the hashtable class is synchronized, it is thread-safe. The Hashtable class instance cannot be accessed simultaneously by more than one thread. As a result, its operations are slower as compared to HashMaps in Java.

Note: If a thread-safe implementation is not required, it is advisable to use HashMap instead of Hashtable.

Note: If a thread-safe highly-concurrent implementation is needed, then it is advised to use ConcurrentHashMap instead of Hashtable.

Constructors of Java Hashtable Class

Both parameterized and non-parameterized constructors may be used to generate a Hashtable in a variety of ways:

Constructor	Description
Hashtable()	It is the default constructor of a hash table that constructs a new and empty hashtable with a default initial capacity (11) and load factor (0.75).
Hashtable(int capacity)	It constructs a new, empty hashtable with the specified initial capacity and default load factor (0.75).
Hashtable(int capacity, float loadFactor)	It constructs a new, empty hashtable with the specified initial capacity and the specified load factor.
Hashtable(Map<? extends K, ? extends V> t)	It constructs a new hashtable with the same mappings as the given Map.

Example:

```
import java.util.Hashtable;

public class HashtableExample {
    public static void main(String[] args)
    {
        // Declaring hashtable using Hashtable()
        Hashtable<Integer, String> hash1 = new Hashtable<>();

        // Declaring hashtable using Generics
        Hashtable<Integer, String> hash2
            = new Hashtable<Integer, String>();

        // Adding elements to hashtable
        // using put() method
        hash1.put(1, "Free");
        hash1.put(2, "Courses");
        hash1.put(3, "on");
```

```

hash1.put(4, "Different");
hash1.put(5, "CS Topics");
hash1.put(6, "By");
hash1.put(7, "Scaler Topics");

hash2.put(1, "Scaler");
hash2.put(2, "By");
hash2.put(3, "InterviewBit");

// Print key-value mappings to the console
System.out.println("Mappings of Hashtable 1 : " + hash1);
System.out.println("Mappings of Hashtable 2 : " + hash2);
}
}

```

LinkedHashMap

LinkedHashMap in Java is similar to that of HashMap with an additional feature, maintaining the order of elements inserted. The main advantage of using LinkedHashMap is that it maintains and tracks the order of insertion where elements can be inserted and accessed in their order. Features:

- It contains values based on keys and implements the map interface by extending the HashMap class.
- It only contains unique elements
- It might have one key which is null and multiple null values
- It is non-synchronized
- It is the same as HashMap with a difference in maintaining the insertion order. For instance, when a code containing HashMap is executed, different orders of elements are shown in the output.

Declaration

```

public class LinkedHashMap<K,V>
extends HashMap<K,V>
implements Map<K,V>

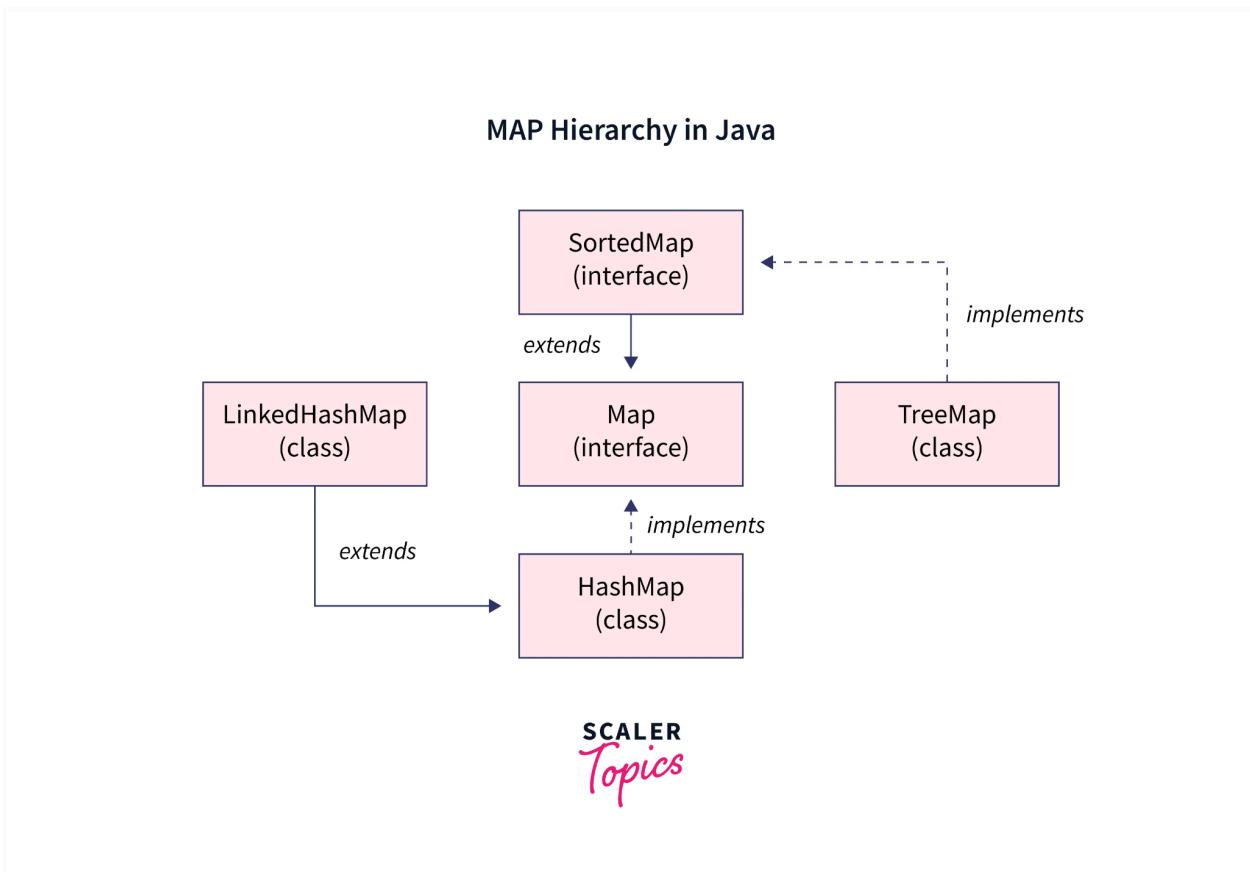
```

Parameters

Here, K is the key type of object and V is the value of the Object

K – The type of the keys in the map.

V – The type of values mapped in the map. The `Map<K, V>` interface is implemented, and `HashMap` class is extended. The hierarchy of `LinkedHashMap` is shown below:



How LinkedHashMap Work Internally?

The data in class is stored in form of notes. The `LinkedHashMap` implementation is very similar to that of a doubly linked list. Each node of the linked hashmap is as follows:

Previous	Key	Value	Next
----------	-----	-------	------



- Hash- the key of input is transformed into a hash which is a key but in a shorter form so that the insertion and search operation of keys are faster.
- Key- this class extends the HashMap which is why data is stored in form of key-value pair.
- Value - a value is associated with every key. It stores the value of the key which can be of any form.
- Next - because the LinkedHashMap stores the order of insertion, it also contains the address of the next node of the LinkedHashMap.
- Previous - it contains the address of the previous node.

Synchronized LinkedHashMap

The LinkedHashMap implementation is not synchronized. If the hash map is being accessed by multiple threads concurrently, then one of the maps is being modified structurally and should be synchronized externally. This is accomplished by synchronizing the object that is used for encapsulating the map. If there is no such object then it should be wrapped using the Collections.synchronizedMap method. Map m = Collections.synchronizedMap(new LinkedHashMap(...));

Constructors of Java Linked Hashmap Class

For creating a `LinkedHashMap` in Java, an object needs to be created for accessing the class. It contains various constructors that make it possible to create the array list. The following are the constructors available:

constructor	description	syntax
<code>LinkedHashMap()</code>	It is used for constructing the default constructor.	<code>LinkedHashMap<K, V> hm = new LinkedHashMap<K, V>();</code>
<code>LinkedHashMap(int capacity)</code>	Used for initializing a <code>LinkedHashMap</code> with a specified capacity.	<code>LinkedHashMap<K, V> hm = new LinkedHashMap<K, V>(int capacity);</code>
<code>LinkedHashMap(Map<? extends K, ? extends V> map)</code>	Used for initializing a <code>LinkedHashMap</code> with specified elements of a map.	<code>LinkedHashMap<K, V> hm = new LinkedHashMap<K, V>(Map<? extends K, ? extends V> map);</code>
<code>LinkedHashMap(int capacity, float fillRatio, boolean Order)</code>	Used to initialize both, the capacity and the fill ratio along with the insertion order.	<code>LinkedHashMap<K, V> hm = new LinkedHashMap<K, V>(int capacity, float fillRatio, boolean Order);</code>

Example:

```
import java.util.*;
class LinkedHashMap1{
    public static void main(String args[]){
        LinkedHashMap<Integer, String> hm=new LinkedHashMap<Integer, String>();
        hm.put(101,"Jenny");
        hm.put(102,"Naman");
        hm.put(103,"Rohan");
        for(Map.Entry m:hm.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
        }
    }
}
```

```
}
```

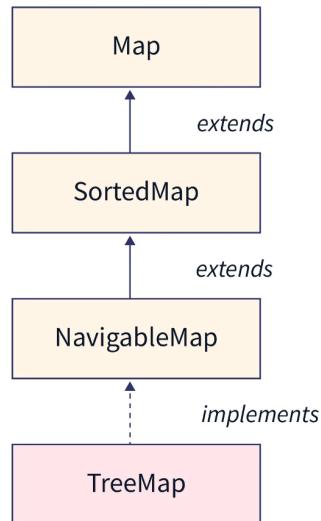
- The LinkedHashMap in Java maintains insertion order (or, optionally, access order).
- Features:
 - It contains values based on keys and implements the map interface by extending the HashMap class.
 - It only contains unique elements
 - It might have one key which is null and multiple values which are null
 - It is non-synchronized
 - It is the same as HashMap with a difference in maintaining the insertion order. For instance, when a code containing HashMap is executed, different orders of elements are shown in the output.

TreeMap

A TreeMap is similar to an ordinary HashMap except for the fact that it stores the key-value pairs in the natural sorting order of the keys, i.e., the non-decreasing order. Alternatively, we can also define our own sorting order with the help of a Comparator.

Features of a TreeMap

- TreeMap in Java is a part of the Java Collection framework where it implements the Map interface and NavigableMap as well. It extends AbstractMap class.
- It doesn't allow null keys (like Map). However, there can be multiple null values in a TreeMap. It throws NullPointerException in the case of any null key insertion.
- It stores the key-value pairs in the natural sorting order of the keys or a sorting order defined with the help of a Comparator.
- TreeMap is not synchronized. To get a synchronized version of the TreeMap, we can use Collections.synchronizedSortedMap method.



SCALER
Topics

Declaration

NavigableMap, being an interface, cannot have any objects created with it. So, it has to be implemented with the abstract class, AbstractMap which provides the implementation of the Map interface. Now, since AbstractMap is an abstract class, TreeMap extends AbstractMap to create objects.

```
public class TreeMap<K,V> extends AbstractMap<K,V> implements
    NavigableMap<K,V>, Serializable, Cloneable
```

Parameters

TreeMap is represented as `TreeMap<K, V>` where:

- K stands for key and can be of any type
- V stands for the corresponding value for a key

Internal Working of TreeMap in Java

TreeMap internally uses a Red-Black tree, a self-balancing binary search tree containing an extra bit for the color (either red or black). The sole purpose of the colors is to make sure at every step of insertion and removal that the tree remains balanced.

For a red-black tree,

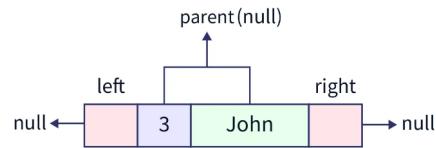
- the nodes must be red or black.
- the root node must be black in color.
- each and every path from the root node to a null should consist of the same number of black nodes.
- Any red node cannot have another red node as its neighbor.

Let's take up an example where we would insert the following key-value mappings into the TreeMap and see what happens behind the scenes.

3	John
1	Alex
4	Chris
5	Marsh
2	Tim

SCALER
Topics

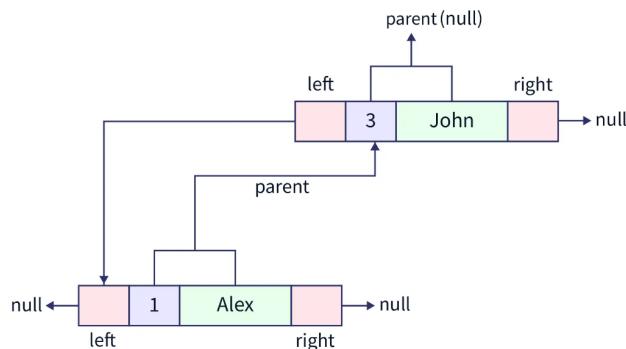
We have the first key-value mapping as (key: 3, value: John) and this mapping will be considered the root of the TreeMap construction. Here's what the structure of the TreeMap looks like.



SCALER
Topics

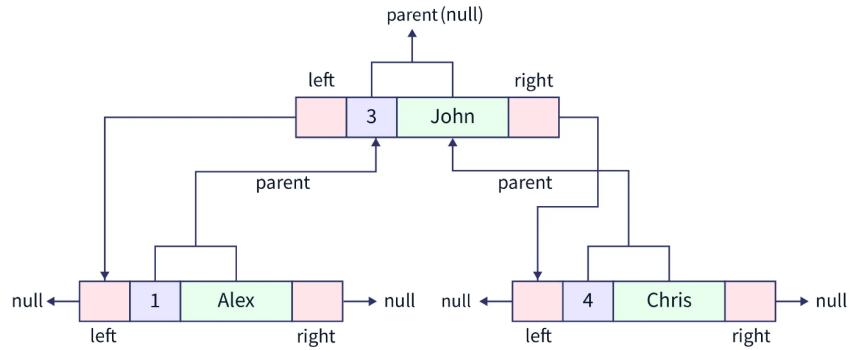
Since it is kind of a binary search tree, so it will consist of left and right pointers, where the keys lower than the parent will be put as the left child and the ones greater than the parent will be put as the right child.

The second key-value mapping (key: 1, value: Alex) is lower than the root key. Therefore, it will be placed as the left child of the root element. Also, this key-value mapping will currently have its left and right pointers as null.



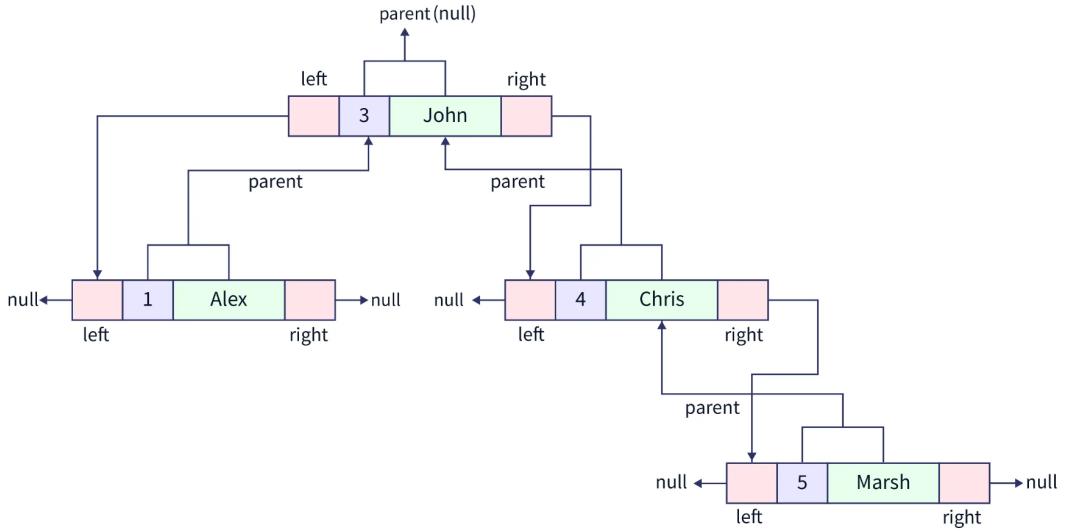
SCALER
Topics

The third key-value mapping (key: 4, value: Chris) has its key greater than the root key and will be placed as the right child of the root element. It will have left and right pointers as null.



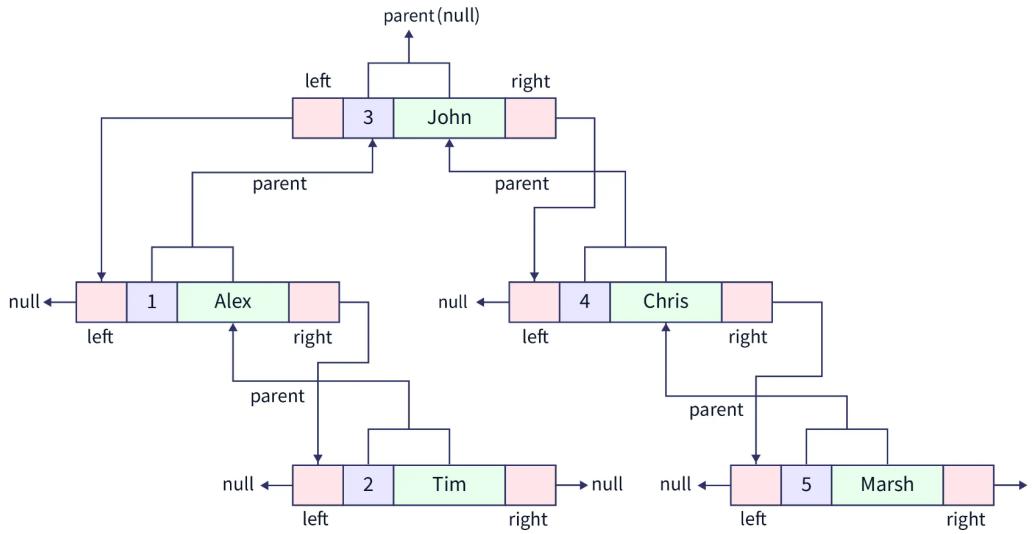
SCALER
Topics

When the fourth key-value mapping (key: 5, value: Marsh) is inserted into the TreeMap, it is greater than the root element and also greater than the succeeding child of the root element (key: 4, value: Chris). So, it will be placed as the right child of the mapping with key 4. The right pointer of the mapping with key 4 will now be pointing to the current key-value mapping.



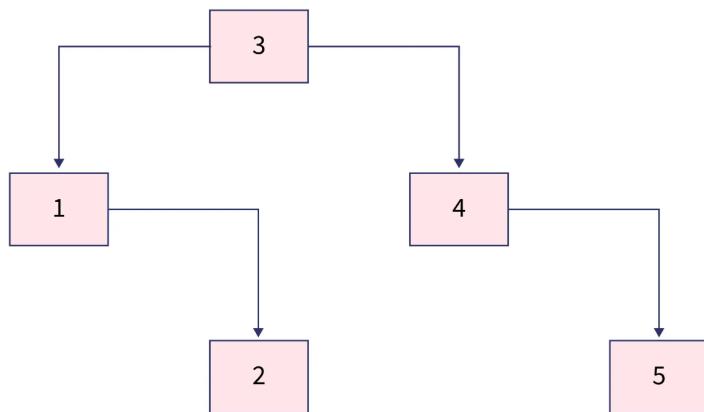
SCALER
Topics

Finally, the last key-value mapping (key: 2, value: Tim) is lower than the root key but also greater than the left child of the root key, i.e. (key: 1, value: Alex). Then, it must be placed as the right child of the key-value mapping with key 1. Here's the final structure of the TreeMap.



SCALER
Topics

The figure below shows the final tree with just the keys.



SCALER
Topics

The order of the tree will be an inorder traversal, i.e. 1, 2, 3, 4, 5.

Constructors of Java TreeMap

There are four ways to define the constructor for a TreeMap. These constructors define how the objects will be created in order to be inserted into the TreeMap. They are listed below.

- `TreeMap()`
- `TreeMap(Comparator comp)`
- `TreeMap(Map M)`
- `TreeMap(SortedMap sm)`

Constructors	Purpose
<code>TreeMap()</code>	The default constructor creates an empty TreeMap where the key-value mappings will be sorted in the natural sorting order of the keys.
<code>TreeMap(Comparator comp)</code>	The customized sorting order can be implemented with the help of Comparator. For instance, sorting the names in descending order.
<code>TreeMap(Map M)</code>	Creates a TreeMap from the already existing map that might have randomly stored key-value mappings.
<code>TreeMap(SortedMap sm)</code>	SortedMap interface can be implemented with the TreeMap class as it provides additional methods for mappings.

```
import java.util.*;

// Main class
public class Main{
    // Main driver method
    public static void main(String[] args)
    {
        // Creating a treemap {key: Integer, value: String}
        TreeMap<Integer, String> tmap = new TreeMap<>();
```

```

        // Storing key-value pairs by using put() method
        tmap.put(1 , "Alex");
        tmap.put(5 , "Marsh");
        tmap.put(3 , "John");
        tmap.put(4 , "Chris");
        tmap.put(2, "Tim");

        // Printing the elements of TreeMap
        System.out.println("Employees: " + tmap);
    }
}

```

TreeMap(Comparator Comp)

Let's say we need to sort the employees based on the flexographic decreasing order of their names. In this case, the natural sorting order cannot be our choice, we need to go for Comparator which would help us implement our own sorting order.

Let's take up the same example, but now, the names will be the keys while their id will be the corresponding values. The output should be based on the descending order of the names.

```

import java.util.*;

// Main class
public class Main{
    // Main driver method
    public static void main(String[] args)
    {
        // Creating a treemap {key: String, value: Integer}
        // The treemap internally uses a custom comparator to
        // define our own sorting order
        TreeMap<String, Integer> tmap =
            new TreeMap<>(new CustomComparator());

        // Storing key-value pairs by using put() method
        tmap.put("Alex", 17);
        tmap.put("Marsh", 29);
        tmap.put("John", 43);
    }
}

```

```

        tmap.put("Chris", 13);
        tmap.put("Tim", 19);

        // Printing the elements of TreeMap
        System.out.println("Employees: " + tmap);
    }
}

// class CustomComparator that implements Comparator and uses compare method
class CustomComparator implements Comparator{

    // The compare method will compare the names of the employee
    public int compare(Object emp1, Object emp2)
    {
        String name1 = emp1.toString();
        String name2 = emp2.toString();

        // The preference will be given in the lexicographic decreasing order
        return name2.compareTo(name1);
    }
}

```

- A **TreeMap** in Java is similar to an ordinary **HashMap** except for the fact that it stores the key-value pairs in the natural sorting order of the keys, i.e., the non-decreasing order.
- Alternatively, we can also define our own sorting order with the help of a **Comparator**.
- **TreeMap** in Java is a part of the Java Collection framework.
- It doesn't allow null keys and throws **NullPointerException** when it finds a null key. However, there can be multiple null values in a **TreeMap**.
- **TreeMap** is not synchronized.
- **TreeMap** is represented as **TreeMap<K,V>** where K stands for key and V stands for the corresponding value for a key.
- **TreeMap** uses a Red-Black tree as its data structure which is a self-balancing tree consisting of red and black nodes.
- There can be four different ways to define a constructor for the **TreeMap**:
 - **TreeMap()**
 - **TreeMap(Comparator comp)**

- **TreeMap(Map M)**
- **TreeMap(SortedMap sm)**
- Apart from the common methods to access and put the elements in the TreeMap (put(), remove(), etc.), we have different methods to find the first and the last keys as well as entries in the TreeMap.
- Similarly, ceiling, floor, higher and lower keys as well as entry can also be found with the help of higherKey(), higherEntry(), lowerKey(), ceilingKey(), ceilingEntry() method, etc.

Difference

What is the Difference Between HashMap and TreeMap?

Characteristics	HashMap	TreeMap
Data Structure	Hashtable	Red-Black Tree
Null keys	Only a single null key is allowed.	No null keys are allowed.
Nature of elements	Since there's no sorting order, it accepts heterogeneous elements.	Homogenous elements are required as it sorts them on the basis of their keys.
Performance	It is faster than TreeMap as it performs all operations (insertion, removal) in O(1) time.	It is slower than HashMap as it performs the same operations in O(log n) as it compares every element before storing them.
Interface	Map, Serializable, and Cloneable interface.	NavigableMap, Serializable, and Cloneable interface.

Difference between HashMap and HashSet

Basic	HashSet	HashMap
Implements	Set interface	Map interface
Duplicates	No	Yes duplicates values are allowed but no duplicate key is allowed
Dummy values	Yes	No
Objects required during an add operation	1	2
Adding and storing mechanism	HashMap object	Hashing technique
Speed	It is comparatively slower than HashSet	It is comparatively faster than HashSet because of hashing technique has been used here.
Null	Have a single null value	Single null key and any number of null values
Insertion Method	Add()	Put()

Difference between TreeMap and TreeSet in Java

S. No.	TreeSet	TreeMap
1.	TreeSet implements SortedSet in Java.	TreeMap implements Map Interface in Java
2.	TreeSet stored a single object in java.	TreeMap stores two Object one Key and one value.
3.	TreeSet does not allow duplication Object in java.	TreeMap in java allows duplication of values.
4.	TreeSet implements NavigableSet in Java.	TreeMap implements NavigableMap in Java.
5.	TreeSet is sorted based on objects.	TreeMap is sorted based on keys.

Similarities between TreeSet and TreeMap in java.

- Both TreeMap and TreeSet belong to `java.util` package.
- Both are the part of the Java Collection Framework.
- They do not allow null values.
- Both are sorted. Sorted order can be natural sorted order defined by Comparable interface or custom sorted order defined by Comparator interface.
- They are not synchronized by which they are not used in concurrent applications.
- Both Provide $O(\log(n))$ time complexity for any operation like `put`, `get`, `containsKey`, `remove`.
- Both TreeSet and TreeMap Internally uses Red-Black Tree.

Differences between HashMap and HashTable in Java

Difference Between Hashmap and Hashtable		
S. No.	Hashmap	
1.	No method is synchronized.	Every method is synchronized.
2.	Multiple threads can operate simultaneously and hence hashmap's object is not thread-safe.	At a time only one thread is allowed to operate the Hashtable's object. Hence it is thread-safe.
3.	Threads are not required to wait and hence relatively performance is high.	It increases the waiting time of the thread and hence performance is low.
4.	Null is allowed for both key and value.	Null is not allowed for both key and value. Otherwise, we will get a null pointer exception.
5.	It is introduced in the 1.2 version.	It is introduced in the 1.0 version.
6.	It is non-legacy.	It is a legacy.