**JAVA**

<https://cse.iitkgp.ac.in/~dsamanta/java/ch1.htm#Java%20Programming%20Paradigms>

<https://www.javatpoint.com/features-of-java>

Sun describes **Java** as a

Simple,

object-oriented,

distributed,

interpreted,

robust,

secure,

architecture neutral,

portable,

high-performance,

multithreaded, and

dynamic language.

Java is an object Oriented Programming (OOP) language.

It incorporates almost every OOP features.

**Object creation** through object template i.e. class,

data abstraction and encapsulation, data and code sharing through **inheritance**,

overloading concept through **polymorphism**, and data/process hiding etc. are some basic OOP features in Java.

**Data Abstraction** is the property by virtue of which only the essential details are displayed to the user. Abstraction is a process of hiding the implementation details and showing only functionality to the user.

The trivial or the non-essentials units are not displayed to the user.

**OR**

**Data Abstraction** may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details.

In java, abstraction is achieved by [interfaces](https://www.geeksforgeeks.org/interfaces-in-java/) and [abstract classes](https://www.geeksforgeeks.org/abstract-classes-in-java/).

There are no pointers in Java 🡪 Only references.

**Why Java is not a purely Object-Oriented Language?**

**Note: People consider JAVA as pure OOP bcz we cant write anything in JAVA w/o classes.**

There are **seven** qualities to be satisfied for a programming language to be pure Object Oriented. They are:

1. Encapsulation/Data Hiding
2. Inheritance
3. Polymorphism
4. Abstraction
5. All predefined types are objects 🡪 this can be taken care of using **Wrapper functions** in java
6. All user defined types are objects
7. All operations performed on objects must be only through methods exposed at the objects.

Java doesn’t support 5,7.

Java has primitive types like int,float etc.

In java, When we declares a class as **static** then it can be used without the use of an object in Java.

If we are using static function or static variable then we can’t call that function or variable by using dot(.) or class object defying object oriented feature.

Platform Independent Programming Environment:

The microprocessor that is inside every computer can understand and execute only the machine codes i.e. in terms of 0’s and 1’s.

But using high level programming, programmers use source code (English like) and using a translator, code is converted into the machine code relevant to the microprocessor.

 The same problem exists in case of computers having same microprocessors but different operating systems.

**Java** has overcome this problem of architecture and environment dependency.

**Java** solves this problem using the concept what is called **Java Virtual Machine (JVM).**

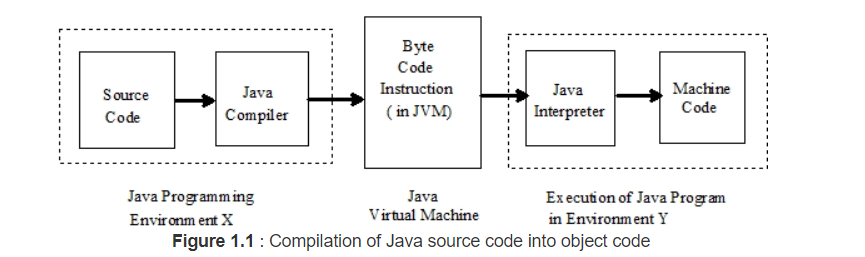
This **JVM** specifies a dummy CPU and dummy instruction set that looks like standard assembly code.

**The Java compiler transforms the source programs into the instructions of JVM.**

**The compiled Java code in known as Byte code instruction** (which are assembly code according to JVM).

Next**, Java interpreter** is there which is **specific to each environment (processor and OS)** and **converts the byte code instructions into the native processor instructions** **before executing it.**

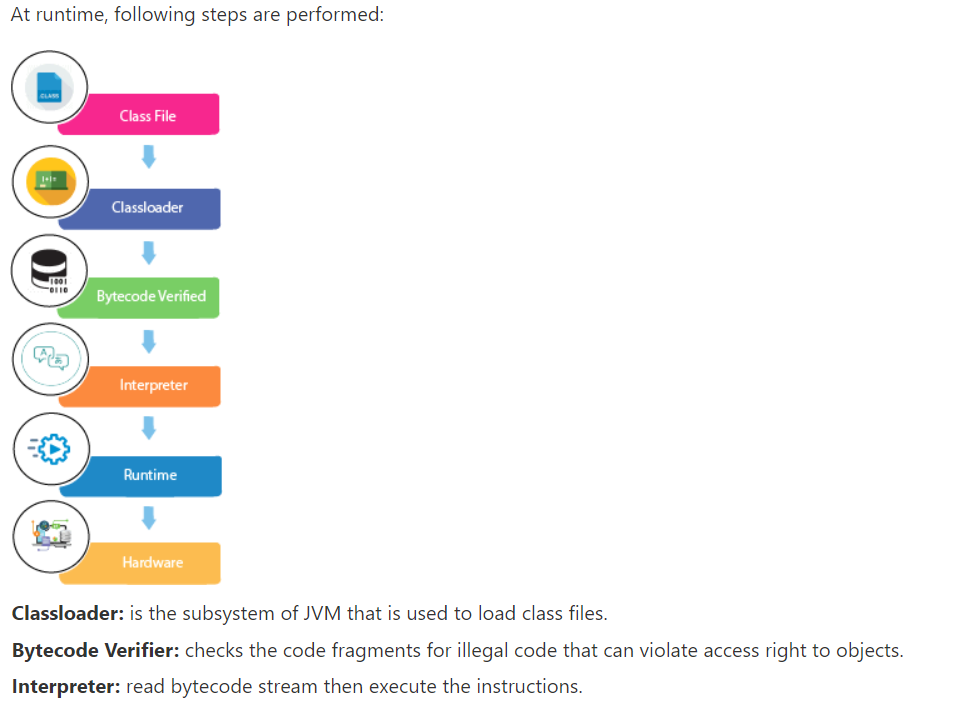
Thus, **converting a program source code into an object code** is compile halfway and interpret half way.



[**https://www.quora.com/Is-the-Java-virtual-machine-a-compiler-an-interpreter-neither-or-both-and-why**](https://www.quora.com/Is-the-Java-virtual-machine-a-compiler-an-interpreter-neither-or-both-and-why)

[**https://stackoverflow.com/questions/7674839/is-the-jvm-a-compiler-or-an-interpreter**](https://stackoverflow.com/questions/7674839/is-the-jvm-a-compiler-or-an-interpreter)

**What Happens at Runtime in Java?**



As the **Java byte codes are architecture and operating system independent** they are highly portable and can be executed on any system without change.

Java’s run time system supports **automatic garbage collectors (GC)** based memory management system.

Occasionally, **GC** will scan through all of the objects in a process to see if any of them are not used.

The obsolete objects are reclaimed from the memory.

As there is no explicit memory manipulation is required by the programmer, so Java programs are highly **robust.**

But in languages like C,C++ the memory needs to be released by the programmer himself failing which can cause memory leak.

Java supports **Multi-Threading.**

 A **byte code verifier in the Java interpreter** looks at the incoming byte codes and **verifies whether the compiled code is strictly language compliant**, in case it finds illegal code, the run time system rejects the code and refuses to run it; thus, trapping all malefice codes.

**High-Performance of JAVA:**

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.

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.

Java supports **call by value** only. There is no call by reference in java.

**JIT Compiler:**

**Just In Time 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 translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU**.

When the JIT compiler environment variable is on (the default), the JVM reads the .class file for interpretation and passes it to the JIT compiler.

The JIT compiler then compiles the bytecodes into native code for the platform on which it is running.

The basic element of object-oriented Programming in Java is a **class**.

A class defines the shape and behaviour of an **object**. i.e Class is a **blue-print/Template** for objects.

Every **object** has a **class** which defines the **structure of an object** (that means what are various component in it, termed as **member elements**) its **functional** **interfaces** (called **methods**) which decide what messages the object will respond to and how.

Class Class Name [extends SuperClassName ]

[ implements Interface ] {

[declaration of member elements ]

[ declaration of methods ]

}

Those included in [] are optional.

**State** of the object is defined in **Member variables**  of class.

**Behaviour** of object is defined in **Methods** of class.

Putting the member elements and methods together in the definition of a class called is called ***encapsulation.***

The **new operator** creates a single instance of a named class and returns a reference to that object.

Java has 3- Access modifiers

1. Public
2. Private
3. Protected.

These 3-access modifiers help in **Information-Hiding**.

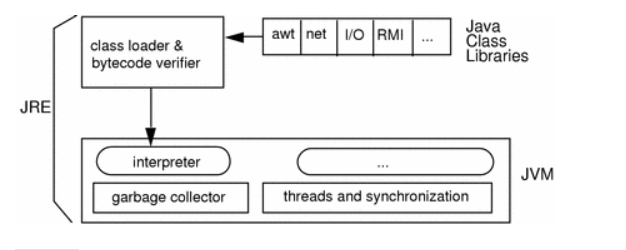
<https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqb/index.html>

**JVM VS JRE**

## **JRE Components**

The JRE is the software environment in which programs compiled for a typical JVM implementation can run. The runtime system includes:

* Code necessary to run Java programs, dynamically link native methods, manage memory, and handle exceptions.
* Implementation of the JVM
* It contains a set of libraries + other files that JVM uses at runtime.



## **JVM**

The JVM is an abstract computing machine, **having an instruction set that uses memory**.

 It is responsible for **Java's cross-platform portability.**

The **Java compiler,**javac, outputs bytecodes and puts them into a .class file.

The JVM then **interprets these bytecodes**, which can then be executed by any JVM implementation, thus providing Java's cross-platform portability.

**Object:**

Object is instance of class which will have memory allocated.

Objects have 2 characteristics : ***State*** and ***Behaviour***

An object stores its state in ***fields*** (variables in some programming languages) and exposes its behavior through ***methods*** (functions in some programming languages).

**Methods operate on an object's internal state** and serve as the primary mechanism for object-to-object communication.

Storing **fields** and **methods** operating on them together is also called **Data encapsulation**

**Data Hiding**  is possible thru data encapsulation as the methods only will be able to access the state of object. So communication should take place thru methods of object only.

**Class**

 A ***class*** is the blueprint from which individual objects are created.

Class does not have memory allocated.

**Inheritance**

Object-oriented programming allows classes to ***inherit*** commonly used **state** and **behaviour** from other classes.

 In the Java programming language, each class is allowed to have only one **direct superclass**, and each superclass can have an unlimited number of ***subclasses***

class <Subclass Name> **extends** <Superclass name> {

// new fields and methods defining

// a mountain bike would go here

}

**Interface**

Objects define their interaction with the outside world through the methods that they expose.

Methods form the object's ***interface*** with the outside world.

In its most common form, an **interface** is a group of related methods with empty bodies.

# Package

A package is a namespace that organizes a set of related classes and interfaces.

# **Java Variables**

**Variable** is a name of memory location. **Variable** is the basic unit of storage in java.

All variables have scope which defines the visibility and lifetime of a variable.

All variables have associated types.

Type of a variable tells what values it can hold.

Ex: if type is integer then variable can hold only integral numbers.

if type is float then variable can hold decimal numbers too.

The Java programming language defines the following kinds of variables:

**Instance Variables (Non-Static Fields):**

**objects** store their individual states in "**non-static fields**", **Non-static** fields are also known as ***instance variables*** because their values are unique to each ***instance*** of a object.

**Class Variables (Static Fields)**

A ***class variable*** is any field declared with the **static** modifier; this tells the compiler that there is exactly **one copy of this variable in existence**, regardless of how many times the class has been instantiated.

**Local Variables**

Like how an object stores its state in fields, a **method** will often store its state in ***local variables***.

**local variables** are only visible to the methods in which they are declared; they are not accessible from the rest of the class.

**Parameters**

**These are the arguments in method signature.**

**Primitive Types in java:**

A **variable's data type** determines the **values** it may contain, plus the **operations** that may be performed on it.

A primitive type is predefined by the language.

The **byte** data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive).

The **short** data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive).

By default, the **int** data type is a 32-bit signed two's complement integer, which has a minimum value of -231 and a maximum value of 231-1.

The **long** data type is a 64-bit two's complement integer. The signed long has a minimum value of -263 and a maximum value of 263-1.

The **float** data type is a single-precision 32-bit IEEE 754 floating point.

The **double** data type is a double-precision 64-bit IEEE 754 floating point.

The **boolean** data type has only two possible values: true and false. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

The **char** data type is a single 16-bit Unicode character.  It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff'

|  |  |
| --- | --- |
| **Data Type** | **Default Value (for fields)** |
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0f |
| double | 0.0d |
| char | '\u0000' |
| String (or any object) | null |
| boolean | false |

**Local variables** are slightly different; the **compiler never assigns a default value** to an uninitialized **local** **variable**.

**Arrays**

An ***array*** is a container object that holds a fixed number of values of a single type.

**Ex:**

Array creation and initialization.

int[] anArray = {

100, 200, 300,

400, 500, 600,

700, 800, 900, 1000

};

**Int arr[] = new int[10];**

Finally, you can use the built-in **length** property to determine the size of any array.

## Copying Arrays

The System class has an **arraycopy** method that you can use to efficiently copy data from one array into another:

public static void **arraycopy**(Object src, int srcPos,

Object dest, int destPos, int length)

**Usage:** System.arraycopy(copyFrom, 2, copyTo, 0, 7);

The three int arguments specify the **starting position in the source array**, **the starting position in the destination array,** and **the number of array elements to copy.**

public static int binarySearch(int[] a, int key)

public static int binarySearch(int[] a,int fromIndex,int toIndex,

int key)

[**equals**](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#equals-int:A-int:A-)(int[] a, int[] a2)

[**fill**](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#fill-boolean:A-boolean-)(boolean[] a, boolean val)

[**sort**](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-char:A-)(char[] a)

[**sort**](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-char:A-int-int-)(char[] a, int fromIndex, int toIndex)

[**sort**](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-int:A-)(int[] a)

[**sort**](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-int:A-int-int-)(int[] a, int fromIndex, int toIndex)

**Note:** If a method() exists for one type then same syntax will be for all other types too.

## **The Type Comparison Operator instanceof**

The **instanceof** operator compares an object to a specified type.

You can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.

**Ex:** <Obj> instanceof <Class>

When using the **instanceof** operator, keep in mind that **null** is not an instance of anything.

# **Expressions, Statements, and Blocks**

An **expression** is a construct made up of variables, operators, and method invocations.

A **statement** forms a complete unit of execution.

The following types of **expressions** can be made into a statement by terminating the expression with a semicolon (;).

* Assignment expressions
* Any use of ++ or --
* Method invocations
* Object creation expressions

A **declaration statement** declares a variable.

**control flow statements** regulate the order in which statements get executed.

A ***block*** is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.

***Control flow statements***, break up the flow of execution.

**Ex:**  if, if-else, if-elseif-else….

# **The switch Statement**

The **switch** statement can have a number of possible execution paths.

A switch works with the byte, short, char, and int primitive data types.

It also works with *enumerated types* the [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) class.

The body of a switch statement is known as a ***switch block***.

A statement in the switch block can be labelled with one or more case or default labels.

An if-then-else statement can test expressions based on ranges of values or conditions, whereas a switch statement tests expressions based only on a single integer, enumerated value, or String object.

Study **“Fall Through”** in switch.

**Classes**

**Constructors** for initializing new objects, declarations for the fields etc.

## Access Modifiers

* public modifier—the field is accessible from all classes.
* private modifier—the field is accessible only within its own class.

In the spirit of **encapsulation**, it is common to make fields **private**. This means that they can only be ***directly*** accessed from the within the class itself.

We still need access to these values, however. This can be done ***indirectly*** by adding **public methods** that obtain the field values.

**Methods:**

***method signature***—the method's **name** and the **parameter types**.

a **method** might have the same name as other methods due to ***method overloading* i.e** you can use the **same name** for all the methods but pass a different argument list to each method.

## **Overloading Methods**

The Java programming language supports ***overloading* methods**, and **Java can distinguish between methods with different *method signatures.***

**Overloaded methods** are differentiated by the **number and the type of the arguments passed** into the method.

The **compiler does not consider return type** when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

**Note:** Overloaded methods should be used sparingly, as they can make code much less readable.

A class contains **constructors** that are invoked to create objects from the class blueprint.

Constructor, use the name of the class and have no return type.

**Java platform differentiates constructors based on the number of arguments in the list and their types**. Hence Constructors can be overloaded too.

The compiler automatically provides a **no-argument, default constructor** for any class without constructors.

**This default constructor will call the no-argument constructor of the superclass**.

In this situation, **the compiler will complain if the superclass does not have a no-argument constructor** so you must verify that it does.

If your class has no explicit superclass, then it has an **implicit superclass of Object**, which *does* have a no-argument constructor.

**Note:**

You can use access modifiers in a constructor's declaration to control which other classes can call the constructor.

 If another class cannot call a Class constructor, it cannot directly create that Class objects.

**Note:**

***Parameters*** refers to the list of variables in a method declaration.

***Arguments*** are the actual values that are passed in when the method is invoked.

When you invoke a method, the arguments used must match the declaration's parameters in type and order.

**Note:**

If you want to pass a method into a method, then use a [lambda expression](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) or a [method reference](https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html).

## Arbitrary Number of Arguments

You can use a construct called **varargs** to pass an arbitrary number of values to a method.

Behind the scenes when varargs are used an array is created.

To use **varargs**, you follow the type of the parameter by an ellipsis (three dots, ...), then a space, and the parameter name.

**The method can then be called with any number of that parameter, including none.**

**Ex:** public Polygon polygonFrom(Point... corners) -🡪 here ‘corners’ will be treated as array.

The method can be called either with an **array** or with a **sequence of arguments**.

The code in the method body will treat the parameter as an array in either case.

Reference data type parameters, such as **objects**, are also passed into methods *by* ***value.***

This means that when the method returns, the passed-in reference still references the same object as before.

*However*, the values of the object's **fields** *can* be changed in the method,

## **Instantiating a Class**

The **new operator instantiates a class by allocating memory** for a new object and **returning a reference** to that memory.

The new operator also invokes the **object constructor**.

The new operator returns a reference to the object it created.

All classes have at least one constructor.

If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, called the **default constructor**.

This default constructor calls the class parent's no-argument constructor, or the Object constructor if the class has no other parent.

If the parent has no constructor (Object does have one), the compiler will reject the program.

**The Garbage Collector**

The Java runtime environment deletes objects when it determines that they are no longer being used.

This process is called **garbage collection**.

An object is eligible for garbage collection when there are no more references to that object.

References that are held in a variable are usually dropped when the variable goes out of scope.

Or, you can explicitly drop an object reference by setting the variable to the special value null.

# **Returning a Value from a Method**

## **Returning a Class or Interface:**

---------------------------------------------------------

When a method uses a class name as its return type, then  the **class of the type of the returned object** must be either a **subclass** of, or the **exact class** of, the return type.

**Note:**

This technique, called ***covariant return type***, means that the return type is allowed to vary in the same direction as the subclass.

You also can use **interface names as return types**. In this case, the **object returned must implement** the specified **interface**.

# **Using the this Keyword**

Within an **instance method or a constructor**, this is a reference to the ***current object***  i.e the object whose method or constructor is called.

**Ex:**

public class Point {

public int x = 0;

public int y = 0;

**//constructor**

**public Point(int x, int y) {**

**this.x = x;**

**this.y = y;**

**}**

}

Each argument to the constructor shadows one of the object's fields i.e parameter ‘x’ shadows class variable ‘x’.

From within a constructor, you can also use the **this** keyword to **call another constructor in the same** class.

Doing so is called an ***explicit constructor invocation.***

If present**, the invocation of another constructor must be the first line in the constructor**.

# **Controlling Access to Members of a Class**

Access level modifiers determine whether other classes can use a particular field or invoke a particular method.

There are two levels of access control:

* **At the top level**—public, or *package-private* (no explicit modifier i.e default ). 🡪 class level
* At the **member level**—public, private, protected, or *package-private* (no explicit modifier i.e default ).

A class may be declared with the modifier public, that class is visible to all classes everywhere.

If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package.

At member level, there are  two additional access modifiers: private and protected.

The private modifier specifies that the member can only be accessed in its own class.

The protected modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Levels** | | | | |
| **Modifier** | **Class** | **Package** | **Subclass** | **World** |
| public | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| *no modifier* | Y | Y | N | N |
| private | Y | N | N | N |

When a number of objects are created from the same class blueprint, they each have their own distinct copies of ***instance variables***.

Sometimes, you want to have **variables that are common to all objects.**

This is accomplished with the static modifier.

Fields that have the static modifier in their declaration are called ***static fields* or *class variables***.

They are associated with the class, rather than with any object.

Every instance of the class shares a class variable, which is in one fixed location in memory.

Class variables are referenced by the class name itself.

**Note:** You can also refer to static fields with an object reference (but not suggested to use this way)

**Static methods**, which have the static modifier in their declarations, should be invoked with the class name, without the need for creating an instance of the class.

**Not all combinations of instance and class variables and methods are allowed**:

* **Instance methods** can access **instance variables and instance methods** directly.
* **Instance methods can access class variables and class methods directly**.
* Class methods can access class variables and class methods directly.
* Class methods ***cannot*** access instance variables or instance methods directly—**they must use an object reference.**
* **‘this’** keyword can’t be used inside static methods.

The final modifier indicates that the value of this field cannot change.

## **Static Initialization Blocks:**

A ***static initialization block*** is a normal block of code enclosed in braces, { }, and preceded by the static keyword.

A class can have any number of static initialization blocks.

They will be called in the order that they appear.

## **Initializing Instance Members**

There are two alternatives to using a constructor to initialize instance variables**: initializer blocks** and **final methods.**

The Java compiler **copies initializer blocks into every constructor.**

**A *final method* cannot be overridden in a subclass**.

# **Nested Classes**

The Java programming language allows you to define a class within another class.

Such a class is called a ***nested class.***

Nested classes are divided into two categories: non-static and static.

Non-static nested classes are called ***inner classes***.

Nested classes that are declared static are called ***static nested classes*.**

A nested class is a **member of its enclosing class**. Enclosing class means outer class.

**Non-static nested classes (inner classes)** have access to other members of the enclosing class, even if they are declared private.

 A nested class can be declared private, public, protected, or *package private*, but outer classes can only be declared public or *package private*.

## **Inner Classes**

As with instance methods and variables**, an inner class is associated with an instance of its enclosing class** and has direct access to that object's methods and fields.

Also, because an inner class is associated with an instance, it cannot define any static members itself.

Objects that are instances of an inner class exist *within* an instance of the outer class i.e An instance of InnerClass can exist only within an instance of OuterClass.

To instantiate an inner class, you must first instantiate the outer class.

**Ex:**

OuterClass outerObject = new OuterClass();

OuterClass.InnerClass innerObject = outerObject.new InnerClass();

There are two special kinds of inner classes: [local classes](https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html) and [anonymous classes](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html).

## **Static Nested Classes**

A static nested class is associated with its outer class.

And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.

## **Local and Anonymous Classes**

There are **two additional types of inner classes**.

1. You can declare an inner class within the body of a method. These classes are known as [local classes](https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html).
2. You can also declare an inner class within the body of a method without naming the class. These classes are known as [anonymous classes](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html).

**Local Class:**

A local class has access to the members of its enclosing class.

Local classes are similar to inner classes because they cannot define or declare any static members.

 a local class can only access local variables that are **declared final**.

Local classes in static methods i.e defined in the static methods  can only refer to static members of the enclosing class.

Local classes are non-static because they have access to instance members of the enclosing block.

**You cannot declare an interface inside a block; interfaces are inherently static i.e interfaces cant be declared inside methods().**

A local class can have **static members** provided that they are **constant variables**.

**Ex:**

public method()

{

Class <class>

{

}

}

# **Anonymous Classes**

Anonymous classes enable you to make your code more concise.

They enable you to declare and instantiate a class at the same time.

They are like local classes except that they do not have a name.

Use them if you need to use a local class only once.

* An anonymous class has access to the members of its enclosing class.
* An anonymous class cannot access local variables in its enclosing scope that are not declared as final or effectively final.
* Like a nested class, a declaration of a type (such as a variable) in an anonymous class shadows any other declarations in the enclosing scope that have the same name.
* You cannot declare static initializers or member interfaces in an anonymous class.
* An anonymous class can have static members provided that they are constant variables.

<https://www.javatpoint.com/anonymous-inner-class>

**Lambda expressions:**

[**https://www.javatpoint.com/java-lambda-expressions**](https://www.javatpoint.com/java-lambda-expressions)

A lambda expression is a short block of code which takes in parameters and returns a value.

**Why use Lambda Expression**

1. To provide the implementation of Functional interface.
2. Less coding.

(argument-list) -> {body}

Lambda expression provides implementation of **functional interface**.

An interface which has only one abstract method is called functional interface.

Java lambda expression is treated as a function, so compiler does not create .class file.

**Ex:**

arrL.forEach(n -> System.out.println(n));

**Here,** ‘arrL’ is array list.

Sorting based on custom logic using lambda expressions.

1. Collections.sort(list,(p1,p2)->{
2. **return** p1.name.compareTo(p2.name);
3. });

# **Method References**

You use [lambda expressions](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) to create anonymous methods.

The above created ‘comparator’ for sort() having (p1,p2) is called anonymous method.

# **Enum Types**

An ***enum*** *type* is a special data type that enables a variable to take set of predefined values.

The variable must be equal to one of the values that have been predefined for it.

**Ex:**

public enum Day {

SUNDAY, MONDAY, TUESDAY, WEDNESDAY,

THURSDAY, FRIDAY, SATURDAY

}

The **enum declaration defines a *class*** (called an *enum type*).

The enum class body can include methods and other fields.

*All* enums implicitly extend java.lang.Enum.

**Note:**

<enum-class>**.**values() -🡪 gives all enums and the values associated with it.

 The constructor for an enum type must be package-private or private access. You cannot invoke an enum constructor yourself.

# **Defining an Interface**

In the Java programming language, **an *interface* is a reference type**, like a class, that can contain *only* constants, method signatures, default methods, static methods, and nested types.

An interface is a **contract** (or a protocol, or a common understanding) of what the classes can do.

The interface in Java is a mechanism to achieve [*abstraction*](https://www.javatpoint.com/abstract-class-in-java).

There can be only abstract methods in the Java interface, not method body.

It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).

**Ex:**

public interface GroupedInterface extends Interface1, Interface2, Interface3 {

The public access specifier indicates that the interface can be used by any class in any package.

An interface can extend other interfaces. An interface can extend any number of interfaces.

The interface body can contain [abstract methods](https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html), [default methods](https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html), and [static methods](https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html#static).

All abstract, default, and static methods in an interface are implicitly public, so you can omit the public modifier.

An interface can contain constant declarations. All constant values defined in an interface are implicitly public, static, and final.

If you define a reference variable whose type is an interface, any object you assign to it *must* be an instance of a class that implements the interface.

<https://docs.oracle.com/javase/tutorial/java/IandI/nogrow.html>

You could also define new [static methods](https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html#static) to existing interfaces.

Users who have classes that implement interfaces enhanced with new default or static methods do not have to modify or recompile them to accommodate the additional methods.

# **Java Default Methods**

Methods which are defined inside the interface and tagged with default are known as default methods.

These methods are non-abstract methods.

**Ex:**

1. **interface** Sayable{
2. // Default method
3. **default** **void** say(){
4. System.out.println("Hello, this is default method");
5. }
6. // Abstract method
7. **void** sayMore(String msg);
8. }

As with default methods, static methods can also have body inside interface itself.

When you extend an interface that contains a default method, you can do the following:

* Not mention the default method at all, which lets your extended interface inherit the default method.
* Redeclare the default method, which makes it abstract.
* Redefine the default method, which overrides it.

Default methods enable you to add methods that accept lambda expressions as parameters to existing interfaces.

# **Static method**

Unlike other methods in Interface, these static methods contain the complete definition of the function and since the definition is complete and the method is static, therefore these methods cannot be overridden or changed in the implementation class.

It cannot be overridden in Implementation Classes.

**Static methods will be called** <Interface\_Name>**.**Static\_method().

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

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

# **Inheritance**

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviours of a parent object.

A class that is derived from another class is called a ***subclass*** (also a *derived class*, *extended class*, or *child class*).

The class from which the subclass is derived is called a ***superclass*** (also a *base class* or a *parent class*).

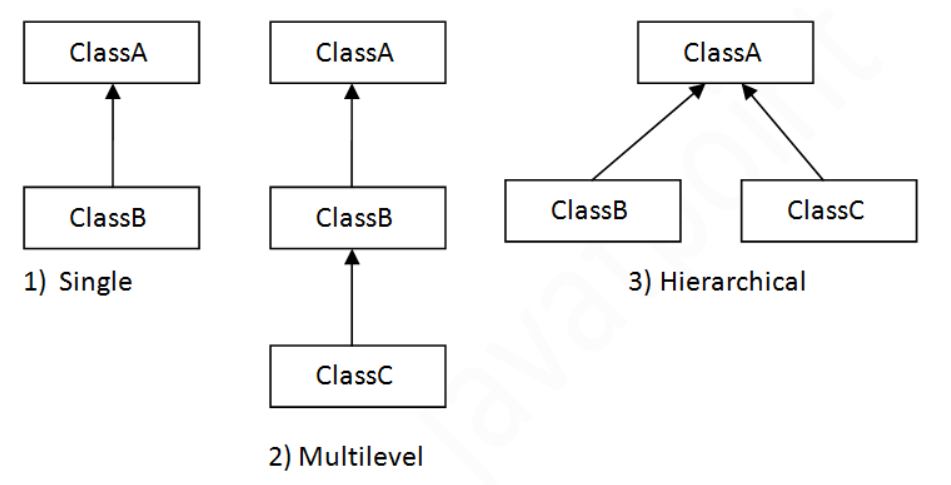
Excepting Object, which has no superclass, every class has one and **only one direct superclass (single inheritance)**.

In the absence of any other explicit superclass, every class is implicitly a subclass of Object.

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

**Inheritance** helps in code reusability.

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



A subclass **inherits** **all the *members* (fields, methods, and nested classes) from its superclass**.

**Constructors are not members, so they are not inherited by subclasses**, but the constructor of the superclass can be invoked from the subclass.

A subclass inherits all of the ***public* and *protected*** members of its parent, and also If the subclass is in the same package as its parent, it also inherits the ***package-private*** members of the parent.

A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing class—both fields and methods.

Therefore, a public or protected nested class inherited by a subclass has indirect access to all of the private members of the superclass.

For example, suppose that you are able to define a new class that extends multiple classes.

When you create an object by instantiating that class, that object will inherit fields from all of the class's superclasses.

What if methods or constructors from different superclasses instantiate the same field? Which method or constructor will take precedence?

Because interfaces do not contain fields, you do not have to worry about problems that result from multiple inheritance of state.

# **Aggregation in Java**

If a class have an entity reference, it is known as **Aggregation**.

Aggregation represents **HAS-A relationship**.

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

* 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.

<https://docs.oracle.com/javase/tutorial/java/IandI/override.html>

An **instance method** in a subclass with the **same signature** (name, plus the number and the type of its parameters**) and return type** as an instance method in the superclass ***overrides*** the superclass's method.

An **overriding method can also return a subtype of the type returned by the overridden method**. This subtype is called a ***covariant return type***.

When overriding a method, you might want to use the @Override annotation that instructs the compiler that you intend to override a method in the superclass.

If a subclass defines **a static method** **with the same signature as a static method in the superclass**, then the method in the subclass ***hides*** the one in the superclass.

## **Interface Methods** [**https://docs.oracle.com/javase/tutorial/java/IandI/override.html**](https://docs.oracle.com/javase/tutorial/java/IandI/override.html)

[Default methods](https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html) and [abstract methods](https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html) in interfaces are inherited like instance methods.

Instance methods are preferred over interface default methods.

Methods that are already overridden by other candidates are ignored.

Inherited instance methods from classes can override abstract interface methods.

**Note**: Static methods in interfaces are never inherited.

The access specifier for an overriding method **can allow more, but not less, access than the overridden method**.

For example**, a protected instance method in the superclass can be made public, but not private, in the subclass.**

**Note:**

In a subclass**, you can overload the methods inherited from the superclass**.

**Such overloaded methods neither hide nor override the superclass instance methods**—they are new methods, unique to the subclass.

# **Polymorphism**

The dictionary definition of **polymorphism** refers to a principle in biology in which an organism or species can have many different forms.

polymorphism is used to describe the situation in which a single statement can take on different definitions.

Polymorphism allows you to define one method and have multiple implementations 🡪 one class/interface/method with many forms.

The word “poly” means many and “morphs” means forms, So it means many forms.

**In Java polymorphism is mainly divided into two types:**

* Compile time Polymorphism
* Runtime Polymorphism

**Compile-time polymorphism**: It is also known as **static polymorphism**. This type of polymorphism is achieved by **function overloading.**

[Runtime polymorphism](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/): It is also known as **Dynamic Method Dispatch**. **It is a process in which a call to the overridden method is resolved at Runtime**.

This type of polymorphism is achieved by **Method Overriding**.

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.

Whenever, a **method is called on an object reference, the declared type of the object reference is checked at compile time** to make sure that the method exists in the declared class.

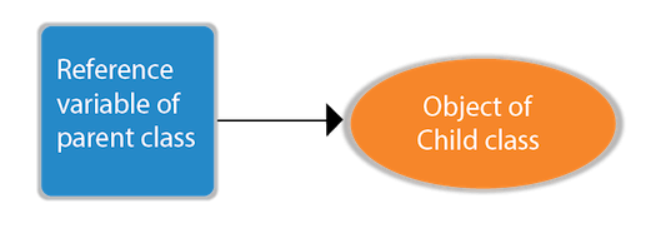
At run time, the object reference could be referring to an instance of some subclass of the declared reference type.

This form of dynamic run time resolution is one of the most powerful mechanisms that Object Oriented in Java brings to bear on code reuse and robustness.

Static methods cannot be overridden because they are not dispatched on the object instance at runtime. So we cant override static methods in java.

### Upcasting

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



#### **Runtime polymorphism can't be achieved by data members.**

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

There are two types of binding

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

Static Binding 🡪 type of object is determined at compile time.

Dynamic binding 🡪 type of the object is determined at runtime.

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

# **Super Keyword in Java**

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

## **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.

#### **Note: super() is added in each class constructor automatically by compiler if there is no super(). It will be the first statement.**

[**https://www.javatpoint.com/instance-initializer-block**](https://www.javatpoint.com/instance-initializer-block)

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

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

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

**final method is inherited but you cannot override it.**

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

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.\

**Constructor cant be declared final as constructors are not inherited.**

**Constructor declarations are not members.**

Constructors are not subject to hiding or [overriding](https://www.geeksforgeeks.org/overriding-in-java/#:~:text=In%20any%20object%2Doriented%20programming,super%2Dclasses%20or%20parent%20classes.&text=Method%20overriding%20is%20one%20of,java%20achieve%20Run%20Time%20Polymorphism.). When there is no chance of constructor overriding, there is no chance of modification also. When there is no chance of modification, then no sense of restricting modification there. We know that the final keyword restricts further modification. So a java constructor can not be final because it inherently it cannot be modified.

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**.

An object of subclass type is also a type of parent class.

## **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.

# **Abstract class in Java**

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user,  it shows only essential things to the user and hides the internal details.

There are two ways to achieve abstraction in java

1. **Abstract class**
2. **Interfaces.**

### 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.

It can have [constructors](https://www.javatpoint.com/java-constructor) and static methods also.

It can have final methods which will force the subclass not to change the body of the method.

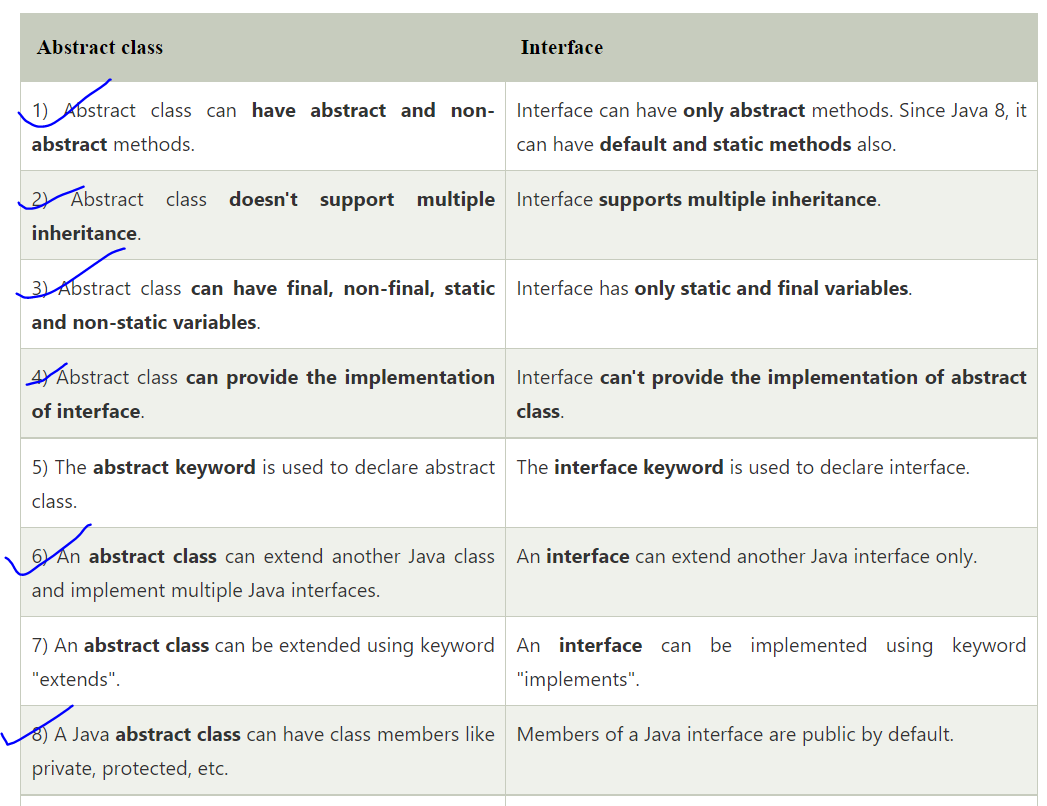
An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

An **interface** which has no member is known as a **marker or tagged interface**, for example, [Serializable](https://www.javatpoint.com/serialization-in-java), Cloneable

They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

Abstract class **doesn't support multiple inheritance**. But intefaces do support multiple inheritance.

Using interfaces we can achieve total abstraction but abstract classes support non-abstract methods also.



<https://www.baeldung.com/java-interface-vs-abstract-class>

# **Wrapper classes in Java**

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

The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

## **Use of Wrapper classes in Java**

* **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.

**public static Wrapper valueOf(String s); 🡪** used to create a Wrapper class object for the given string.

**public static Wrapper valueOf(primitive p); 🡪** to create a Wrapper object for the given primitive type.

1. public byte byteValue()
2. public short shortValue()
3. public int intValue()
4. public long longValue()
5. public float floatValue()
6. public float doubleValue()

The above are the methods to get the primitive for the given Wrapper Object.

**Ex:**

public static void main(String[] args)

    {

        Integer I = new Integer(130);

        System.out.println(I.byteValue());

        System.out.println(I.shortValue());

        System.out.println(I.intValue());

        System.out.println(I.longValue());

        System.out.println(I.floatValue());

        System.out.println(I.doubleValue());

    }

**primitive parseXxx(String s) :** Every Wrapper class except character class contains the following parseXxx() method to find primitive for the given String object.

Ex: parseInt(s),parseLong(s).

**toString(), toString(primitive p) 🡪** other methods in wrapper classes.

**Collection Framework**

Any group of individual objects which are represented as a single unit is known as the collection of the objects.

**Collection** is a container of objects. And collection itself is an object too.

A **framework**, or **software framework**, is a platform that provides a foundation for developing complex/ large **software** applications using the functionalities provided by framework.

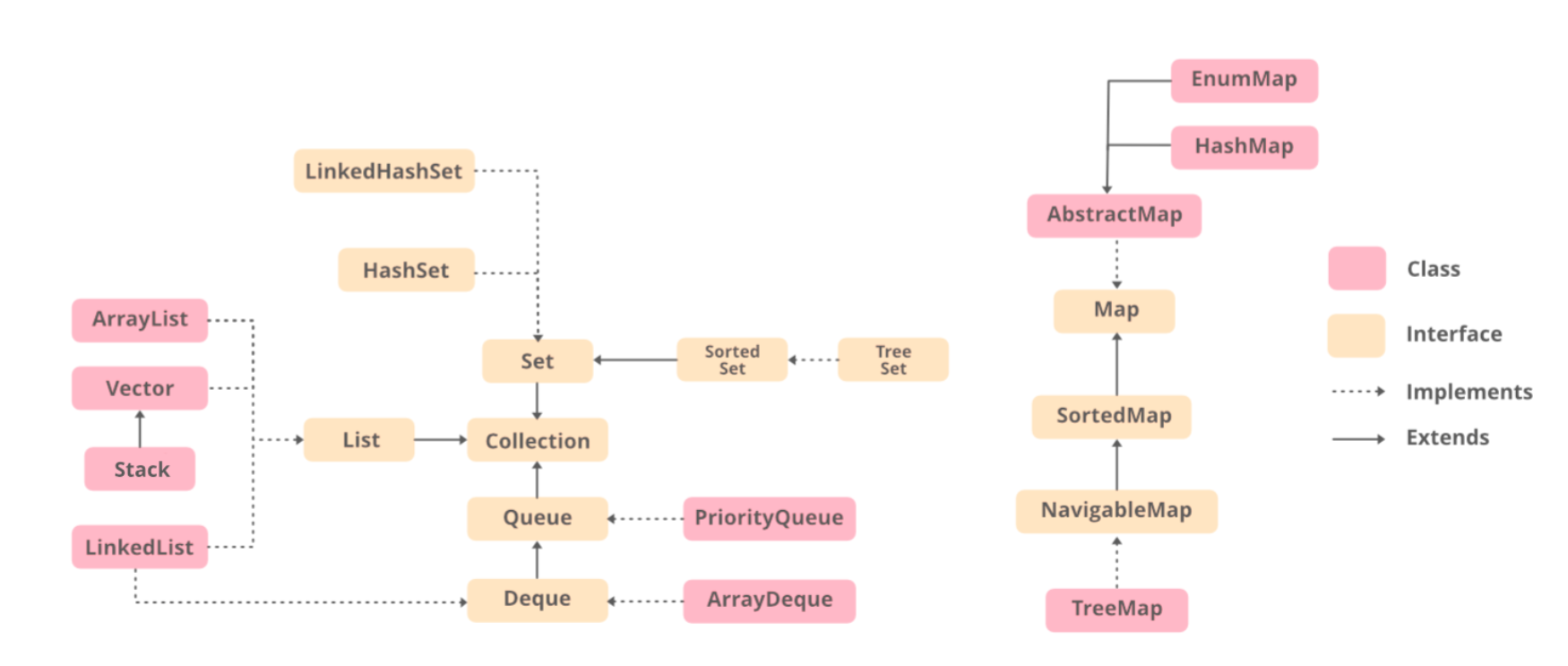
**Ex:** [**https://stackoverflow.com/questions/2964140/what-is-a-software-framework**](https://stackoverflow.com/questions/2964140/what-is-a-software-framework)

A framework is a set of [classes](https://www.geeksforgeeks.org/classes-objects-java/) and [interfaces](https://www.geeksforgeeks.org/interfaces-in-java/) which provide a ready-made architecture.

### Hierarchy of the Collection Framework

**The utility package, (java.util)** contains all the classes and interfaces that are required by the collection framework.

The collection framework contains an interface named an **iterable** interface which provides the iterator to iterate through all the collections.



### Methods of the Collection Interface

|  |  |
| --- | --- |
| Method | Description |
| [add(Object)](https://www.geeksforgeeks.org/collection-add-method-in-java-with-examples/) | This method is used to add an object to the collection. |
| [**addAll(Collection c)**](https://www.geeksforgeeks.org/collections-addall-method-in-java-with-examples/) | This method adds all the elements in the given collection ‘c’ to this collection. |
| [**clear()**](https://www.geeksforgeeks.org/collection-clear-method-in-java-with-examples/) | This method removes all of the elements from this collection. |
| [**contains(Object o)**](https://www.geeksforgeeks.org/collection-contains-method-in-java-with-examples/) | This method returns true if the collection contains the specified element. |
| **equals(Object o)** | This method compares the specified object with this collection for equality. |
| [**isEmpty()**](https://www.geeksforgeeks.org/collection-isempty-method-in-java-with-examples/) | This method returns true if this collection contains no elements. |
| **iterator()** | This method returns an iterator over the elements in this collection. |
| [**max()**](https://www.geeksforgeeks.org/collections-max-method-in-java-with-examples/) | This method is used to return the maximium value present in the collection. |
| **remove(Object o)** | This method is used to remove the given object from the collection. If there are duplicate values, then this method removes the first occurrence of the object. |
| **removeAll(Collection c)** | This method is used to remove all the objects mentioned in the given collection from the collection. |
| **removeIf(Predicate filter)** | This method is used to removes all the elements of this collection that satisfy the given [predicate](https://www.geeksforgeeks.org/mathematic-logic-predicates-quantifiers/). |
| **size()** | This method is used to return the number of elements in the collection. |
| **toArray()** | This method is used to return an array containing all of the elements in this collection. |

### Interfaces in Collections

### ****Iterable Interface:**** This is the root interface for the entire collection framework.

### The main functionality of this interface is to provide an iterator for the collections.

### Therefore, this interface contains only one abstract method which is the iterator.

### *Iterator*iterator();

### ****Collection Interface**:** This interface extends the iterable interface and is implemented by all the classes in the collection framework.

### This interface provides all basic functionalities for every collection like add(),remove(),size() etc.

### [List Interface:](https://www.geeksforgeeks.org/list-interface-java-examples/) This is a child interface of the collection interface.

### This interface is dedicated to the data of the list type in which we can store all the ordered collection of the objects.

### This also allows duplicate data to be present in it.

# **ArrayList in Java**

It provides us with dynamic arrays in Java.

1. **Adding Elements:**
   1. **add(Object):** This method is used to add an element at the **end** of the ArrayList.
   2. **add(int index, Object):** This method is used to add an element **at a specific index** in the ArrayList.
2. **Changing Elements:**

 After adding the elements, if we wish to change the element, it can be done using the [set()](https://www.geeksforgeeks.org/arraylist-set-method-in-java-with-examples/) method.

**set(index, value);**

1. **Removing Elements:**
   1. **remove(Object):** This method is used to **simply remove an object** from the ArrayList. If there are multiple such objects, then the first occurrence of the object is removed.
   2. **remove(int index):** Since an ArrayList is indexed, this method takes an integer value which simply removes the element present at that specific index in the ArrayList. After removing the element, all the elements are moved to the left to fill the space.
2. **Fetch element at index:**
   1. **get(i):** used to get element at index-i.

**ex:**

**ArrayList<String> al = new ArrayList<>();**

**Iterator<String> iter = list.iterator(); // iterator.**

# **Vector Class in Java**

The **Vector** class implements a growable array of objects.

They are very similar to [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/) but Vector is synchronized means its thread-safe.

**All methods for adding, removing, updating are same as Arraylist.**

**Important points regarding Increment of vector capacity:**

If **the increment is specified**, **Vector will expand according to it in each allocation cycle** but if the increment is not specified then the **vector’s capacity gets doubled** in each allocation cycle.

[**https://www.geeksforgeeks.org/java-util-vector-class-java/**](https://www.geeksforgeeks.org/java-util-vector-class-java/)

# **Stack Class in Java**

Stack extends **Vector** Class.

1. **Adding ele:** push(ele).
2. **Accessing the top ele:** peek().
3. **Removing ele:** pop().
4. [empty()](https://www.geeksforgeeks.org/stack-empty-method-in-java/) : Returns true if stack is empty.

Stack also inherits all methods from **Vector** class.

### Methods inherited from class java.util.Vector

|  |  |
| --- | --- |
| [add(Object obj)](https://www.geeksforgeeks.org/vector-add-method-in-java/) | Appends the specified element to the end of this Vector. |
| [add(int index, Object obj)](https://www.geeksforgeeks.org/vector-add-method-in-java/) | Inserts the specified element at the specified position in this Vector. |
| [addAll(Collection c)](https://www.geeksforgeeks.org/vector-addall-method-in-java/) | Appends all of the elements in the specified Collection to the end of this Vector,  in the order that they are returned by the specified Collection’s Iterator. |
| [addElement(Object o)](https://www.geeksforgeeks.org/vector-addelement-method-in-java/#:~:text=addElement()%20method%20is%20used,()%20method%20of%20Vector%20class.) | Adds the specified component to the end of this vector, increasing its size by one. |
| [capacity()](https://www.geeksforgeeks.org/vector-capacity-method-in-java/) | Returns the current capacity of this vector. |
| [clear()](https://www.geeksforgeeks.org/vector-clear-method-in-java/) | Removes all the elements from this Vector. |
| [contains(Object o)](https://www.geeksforgeeks.org/vector-contains-method-in-java/) | Returns true if this vector contains the specified element. |
| [isEmpty()](https://www.geeksforgeeks.org/vector-isempty-method-in-java/) | Tests if this vector has no components. |
| [size()](https://www.geeksforgeeks.org/vector-size-method-in-java/#:~:text=size()%20method%20in%20Java%20is%20used%20to%20get%20the,elements%20present%20in%20the%20Vector.&text=Parameters%3A%20The%20method%20does%20not,elements%20present%20in%20the%20Vector.) | Returns the number of components in this vector. |
| [toArray()](https://www.geeksforgeeks.org/vector-toarray-method-in-java-with-examples/) | Returns an array containing all of the elements in this Vector in the correct order. |

# **LinkedList in Java**

1. **Adding Elements:**
   1. **add(Object):** This method is used to add an element at the end of the LinkedList.
   2. **add(int index, Object):** This method is used to add an element at a specific index in the LinkedList.
2. **Changing Elements:**
   1. **Set(index,ele):** used to update the element at indx.
3. **Removing Elements:** 
   1. **remove(Object):** This method is used to simply remove an object from the LinkedList. If there are multiple such objects, then the first occurrence of the object is removed.
   2. **remove(int index):** Since a LinkedList is indexed, this method takes an integer value which simply removes the element present at that specific index in the LinkedList.
4. **Fetching the element:** 
   1. **get(i):** Used to get the element at index-i.

 Internally, the LinkedList is implemented using the [doubly linked list data structure](https://www.geeksforgeeks.org/doubly-linked-list/).

|  |  |  |
| --- | --- | --- |
| [**add(int index, E element)**](https://www.geeksforgeeks.org/java-util-linkedlist-add-method-in-java/) | This method Inserts the specified element at the specified position in this list. |  |
| |  |  | | --- | --- | | [**add(E e)**](https://www.geeksforgeeks.org/java-util-linkedlist-add-method-in-java/) | This method Appends the specified element to the end of this list. | |  |  | |  |  |
| [**addFirst(E e)**](https://www.geeksforgeeks.org/linkedlist-addfirst-method-in-java/) | This method Inserts the specified element at the beginning of this list. |

|  |  |
| --- | --- |
| [**addLast(E e)**](https://www.geeksforgeeks.org/linkedlist-addlast-method-in-java/) | This method Appends the specified element to the end of this list. |
|  |  |

|  |  |
| --- | --- |
| [**clear()**](https://www.geeksforgeeks.org/linkedlist-clear-method-in-java/) | This method removes all of the elements from this list. |
|  |  |
| [contains(Object o)](https://www.geeksforgeeks.org/linkedlist-contains-method-in-java/) | This method returns true if this list contains the specified element. |
|  |  |
| [get(int index)](https://www.geeksforgeeks.org/linkedlist-get-method-in-java/) | This method returns the element at the specified position in this list. |

|  |  |
| --- | --- |
| [**getFirst()**](https://www.geeksforgeeks.org/java-util-linkedlist-get-getfirst-getlast-java/) | This method returns the first element in this list. |
| [**getLast()**](https://www.geeksforgeeks.org/linkedlist-getlast-method-in-java/) | This method returns the last element in this list. |

|  |  |
| --- | --- |
| [**indexOf(Object o)**](https://www.geeksforgeeks.org/linkedlist-indexof-method-in-java/) | This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element. |
| [**lastIndexOf(Object o)**](https://www.geeksforgeeks.org/linkedlist-lastindexof-method-in-java/) | This method returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element. |

|  |  |
| --- | --- |
| [**peek()**](https://www.geeksforgeeks.org/java-util-linkedlist-peek-peekfirst-peeklast-java/) | This method retrieves, but does not remove, the head (first element) of this list. |
| [**peekFirst()**](https://www.geeksforgeeks.org/java-util-linkedlist-peek-peekfirst-peeklast-java/) | This method retrieves, but does not remove, the first element of this list, or returns null if this list is empty. |
| [**peekLast()**](https://www.geeksforgeeks.org/java-util-linkedlist-peek-peekfirst-peeklast-java/) | This method retrieves, but does not remove, the last element of this list, or returns null if this list is empty. |
| [**poll()**](https://www.geeksforgeeks.org/java-util-linkedlist-poll-pollfirst-polllast-%20examples-java/) | This method retrieves and removes the head (first element) of this list. |
| [**pollFirst()**](https://www.geeksforgeeks.org/java-util-linkedlist-poll-pollfirst-polllast-%20examples-java/) | This method retrieves and removes the first element of this list, or returns null if this list is empty. |
| [**pollLast()**](https://www.geeksforgeeks.org/java-util-linkedlist-poll-pollfirst-polllast-%20examples-java/) | This method retrieves and removes the last element of this list, or returns null if this list is empty. |

|  |  |
| --- | --- |
| [**size()**](https://www.geeksforgeeks.org/linkedlist-size-method-in-java/) | This method returns the number of elements in this list. |

|  |  |
| --- | --- |
| [**toArray()**](https://www.geeksforgeeks.org/linkedlist-toarray-method-in-java-with-example/) | This method returns an array containing all of the elements in this list in proper sequence (from first to last element). |

|  |  |
| --- | --- |
| [**remove()**](https://www.geeksforgeeks.org/linkedlist-remove-method-in-java/) | This method retrieves and removes the head (first element) of this list. |

|  |  |
| --- | --- |
| [**removeFirst()**](https://www.geeksforgeeks.org/linkedlist-removefirst-method-in-java/) | This method removes and returns the first element from this list. |
| [removeLast()](https://www.geeksforgeeks.org/linkedlist-removelast-method-in-java/) | This method removes and returns the last element from this list. |

# **Queue Interface In Java**

Being an interface the queue needs a concrete class for the declaration and the most common classes are the [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/" \t "_blank) and [LinkedList](https://www.geeksforgeeks.org/linked-list-in-java/) in Java.

*Queue<Obj> queue = new PriorityQueue<Obj> ();*

**Queue<Obj> q = new LinkedList<>();**

|  |  |
| --- | --- |
| Method | Description |
| [add(element)](https://www.geeksforgeeks.org/queue-add-method-in-java/) | This method is used to add elements at the tail of queue. More specifically, at the last of linked-list if it is used, or according to the priority in case of priority queue implementation. |
| [peek()](https://www.geeksforgeeks.org/queue-peek-method-in-java/) | This method is used to view the head of queue without removing it. It returns Null if the queue is empty. |
| [poll()](https://www.geeksforgeeks.org/queue-poll-method-in-java/) | This method removes and returns the head of the queue. It returns null if the queue is empty. |
| [remove()](https://www.geeksforgeeks.org/queue-remove-method-in-java/) | This method removes and returns the head of the queue. It throws NoSuchElementException when the queue is empty. |
| isEmpty() | True if queue is empty |
|  |  |

# **PriorityQueue in Java**

The PriorityQueue is based on the priority heap.

Default implementation in Java of priority queue is **Min-Heap.**

PriorityQueue doesn’t permit null.

PriorityQueue are unbound queues.

It provides O(log(n)) time for add and poll methods.

*PriorityQueue<E> pq = new PriorityQueue<E>();*

*PriorityQueue<E> pq = new PriorityQueue(Comparator<E> comparator);*

*PriorityQueue<E> pq = new PriorityQueue<E>(****Collections.reverseOrder()****); //* ***Max-heap.***

|  |  |
| --- | --- |
| [add(E e)](https://www.geeksforgeeks.org/priorityqueue-add-method-in-java/) | Inserts the specified element into this priority queue. |
| [clear()](https://www.geeksforgeeks.org/priorityqueue-clear-method-in-java/#:~:text=clear()%20method%20is%20used,only%20empty%20an%20existing%20PriorityQueue.) | Removes all of the elements from this priority queue. |
| [contains​(Object o)](https://www.geeksforgeeks.org/priorityqueue-contains-method-in-java/#:~:text=PriorityQueue.,any%20particular%20element%20or%20not.&text=Return%20Value%3A%20The%20method%20returns,queue%20otherwise%20it%20returns%20False.) | Returns true if this queue contains the specified element. |
| [iterator()](https://www.geeksforgeeks.org/priorityqueue-iterator-method-in-java/) | Returns an iterator over the elements in this queue. |
| [remove​(Object o)](https://www.geeksforgeeks.org/priorityqueue-remove-method-in-java/) | Removes a single instance of the specified element from this queue, if it is present. |
| [toArray()](https://www.geeksforgeeks.org/priorityqueue-toarray-method-in-java/#:~:text=toArray(arr%5B%5D)%20method%20in,the%20previous%20method%20without%20parameters.) | Returns an array containing all of the elements in this queue. |
| [isEmpty()](https://www.geeksforgeeks.org/abstractcollection-isempty-method-in-java-with-examples/) | Returns true if this collection contains no elements. |
| size() | Returns the number of elements in this collection. |
| [peek()](https://www.geeksforgeeks.org/queue-peek-method-in-java/) | Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. |
| [poll()](https://www.geeksforgeeks.org/queue-poll-method-in-java/) | Retrieves and removes the head of this queue, or returns null if this queue is empty |

Iterator iterator = pq.iterator();

***// pq is priorityqueue***

# **Deque interface in Java**

The Deque is related to the double-ended queue that supports addition or removal of elements from either end of the data structure.

*Deque<Obj> deque = new ArrayDeque<Obj> ();*

Deque<Obj> dq = new LinkedList<>();

|  |  |
| --- | --- |
| [add(element)](https://www.geeksforgeeks.org/deque-add-method-in-java/) | This method is used to add an element at the tail of the queue. If the Deque is capacity restricted and no space is left for insertion, it returns an IllegalStateException. The function returns true on successful insertion. |
| [addFirst(element)](https://www.geeksforgeeks.org/deque-addfirst-method-in-java-with-examples/) | This method is used to add an element at the head of the queue. If the Deque is capacity restricted and no space is left for insertion, it returns an IllegalStateException. The function returns true on successful insertion. |
| [addLast(element)](https://www.geeksforgeeks.org/deque-addlast-method-in-java/) | This method is used to add an element at the tail of the queue. If the Deque is capacity restricted and no space is left for insertion, it returns an IllegalStateException. The function returns true on successful insertion. |
| [**contains()**](https://www.geeksforgeeks.org/deque-contains-method-in-java/) | This method is used to check whether the queue contains the given object or not. |
| [**getFirst()**](https://www.geeksforgeeks.org/deque-getfirst-method-in-java/) | This method is used to retrieve, but not remove, the first element of this deque. |
| [**getLast()**](https://www.geeksforgeeks.org/deque-getlast-method-in-java/) | This method is used to retrieve, but not remove, the last element of this deque. |
| [**iterator()**](https://www.geeksforgeeks.org/deque-iterator-method-in-java/) | This method returns an iterator for the deque. The elements will be returned in order from first (head) to last (tail). |
| **peek()** | This method is used to retrieve the element at the head of the deque but doesn’t remove the element from the deque. This method returns null if the deque is empty. |
| **peekFirst()** | This method is used to retrieve the element at the head of the deque but doesn’t remove the element from the deque. This method returns null if the deque is empty. |
| **peekLast()** | This method is used to retrieve the element at the tail of the deque but doesn’t remove the element from the deque. This method returns null if the deque is empty. |
| **poll()** | This method is used to retrieve and remove the element at the head of the deque. This method returns null if the deque is empty. |
| **pollFirst()** | This method is used to retrieve and remove the element at the head of the deque. This method returns null if the deque is empty. |
| **pollLast()** | This method is used to retrieve and remove the element at the tail of the deque. This method returns null if the deque is empty. |
| **removeFirst()** | This method is used to remove an element from the head of the queue. |
| **removeLast()** | This method is used to remove an element from the tail of the queue. |
| **size()** | This method is used to find and return the size of the deque. |

# **HashSet in Java**

No guarantee is made as to the iteration order of the set which means that the class does not guarantee the constant order of elements over time.

This class permits the null element.

The underlying data structure for HashSet is [Hashtable](https://www.geeksforgeeks.org/hashtable-in-java/)..

duplicate values are not allowed.

Objects that you insert in HashSet are not guaranteed to be inserted in the same order.

NULL elements are allowed in HashSet.

|  |  |
| --- | --- |
| [add(E e)](https://www.geeksforgeeks.org/hashset-add-method-in-java/) | Used to add the specified element if it is not present, if it is present then return false. |
| [clear()](https://www.geeksforgeeks.org/hashset-clear-method-in-java/) | Used to remove all the elements from set. |
| [contains(Object o)](https://www.geeksforgeeks.org/hashset-contains-method-in-java/) | Used to return true if an element is present in set. |
| [remove(Object o)](https://www.geeksforgeeks.org/hashset-remove-method-in-java/) | Used to remove the element if it is present in set. |
| [iterator()](https://www.geeksforgeeks.org/hashset-iterator-method-in-java/) | Used to return an iterator over the element in the set. |
| [isEmpty()](https://www.geeksforgeeks.org/hashset-isempty-method-in-java/) | Used to check whether the set is empty or not. Returns true for empty and false for a non-empty condition for set. |
| [size()](https://www.geeksforgeeks.org/hashset-size-method-in-java/) | Used to return the size of the set. |
| [toArray()](https://www.geeksforgeeks.org/abstractcollection-toarray-method-in-java-with-examples/) | This method is used to form an array of the same elements as that of the Set. |

# LinkedHashSet

The **LinkedHashSet** is an ordered version of [HashSet](https://www.geeksforgeeks.org/hashset-in-java/) that maintains a doubly-linked List across all elements.

When the iteration order is needed to be maintained this class is used.

*LinkedHashSet<E> hs = new LinkedHashSet<E>();*

|  |  |
| --- | --- |
| [add(E e)](https://www.geeksforgeeks.org/hashset-add-method-in-java/) | Adds the specified element to this set if it is not already present. |
| [clear()](https://www.geeksforgeeks.org/hashset-clear-method-in-java/) | Removes all of the elements from this set. |
| [contains(Object o)](https://www.geeksforgeeks.org/hashset-contains-method-in-java/) | Returns true if this set contains the specified element. |
| [isEmpty()](https://www.geeksforgeeks.org/hashset-isempty-method-in-java/) | Returns true if this set contains no elements. |
| [iterator()](https://www.geeksforgeeks.org/hashset-iterator-method-in-java/) | Returns an iterator over the elements in this set. |
| [remove(Object o)](https://www.geeksforgeeks.org/hashset-remove-method-in-java/) | Removes the specified element from this set if it is present. |
| [size()](https://www.geeksforgeeks.org/hashset-size-method-in-java/) | Returns the number of elements in this set (its cardinality). |

# TreeSet in Java

TreeSet is one of the most important implementations of the [SortedSet interface](https://www.geeksforgeeks.org/sortedset-java-examples/) in Java that uses a [Tree](https://www.geeksforgeeks.org/binary-tree-data-structure/) for storage.

It can also be ordered by a **Comparator** provided at set creation time

Objects in a TreeSet are stored in a sorted and ascending order. Balanced Binary search tree is used for its implementation.

*TreeSet ts = new TreeSet(Comparator comp);*

*TreeSet ts = new TreeSet(Comparator comp);*

[**add(Object o)**](https://www.geeksforgeeks.org/treeset-add-method-in-java/)

|  |  |
| --- | --- |
| [**ceiling?(E e)**](https://www.geeksforgeeks.org/treeset-ceiling-method-in-java-with-examples/) | This method returns the least element in this set greater than or equal to the given element, or null if there is no such element. |
| [clear()](https://www.geeksforgeeks.org/treeset-clear-method-in-java/) | This method will remove all the elements. |
| [contains(Object o)](https://www.geeksforgeeks.org/treeset-contains-method-in-java/) | This method will return true if a given element is present in TreeSet else it will return false. |
| [descendingSet?()](https://www.geeksforgeeks.org/treeset-descendingset-method-in-java-with-examples/) | This method returns a reverse order view of the elements contained in this set. |
| [first()](https://www.geeksforgeeks.org/treeset-first-method-in-java/) | This method will return the first element in TreeSet if TreeSet is not null else it will throw NoSuchElementException. |

|  |  |
| --- | --- |
| [**floor?(E e)**](https://www.geeksforgeeks.org/treeset-floor-method-in-java-with-examples/) | This method returns the greatest element in this set less than or equal to the given element, or null if there is no such element. |

|  |  |
| --- | --- |
| [**isEmpty()**](https://www.geeksforgeeks.org/treeset-isempty-method-in-java/) | This method is used to return true if this set contains no elements or is empty and false for the opposite case. |
| [**Iterator iterator()**](https://www.geeksforgeeks.org/treeset-iterator-method-in-java/) | Returns an iterator for iterating over the elements of the set. |
| [**last()**](https://www.geeksforgeeks.org/treeset-last-method-in-java/) | This method will return the last element in TreeSet if TreeSet is not null else it will throw NoSuchElementException. |
| [pollFirst?()](https://www.geeksforgeeks.org/treeset-pollfirst-method-in-java/) | This method retrieves and removes the first (lowest) element, or returns null if this set is empty. |
| [pollLast?()](https://www.geeksforgeeks.org/treeset-polllast-method-in-java-with-example/) | This method retrieves and removes the last (highest) element, or returns null if this set is empty. |
| [remove(Object o)](https://www.geeksforgeeks.org/treeset-remove-method-in-java/) | This method is used to return a specific element from the set. |
| [size()](https://www.geeksforgeeks.org/treeset-size-method-in-java/) | This method is used to return the size of the set or the number of elements present in the set. |

# TreeMap in Java

 The map is sorted according to the natural ordering of its keys, or by a [Comparator](https://www.geeksforgeeks.org/comparator-interface-java/) provided at map creation time,

TreeMap in Java does not allow null keys (like Map)

TreeMap is based upon a [redblack tree](https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/) data structure

 TreeMap<Integer, String> tree\_map = new TreeMap<Integer, String>();

TreeMap<Student, Integer> tree\_map = new TreeMap<Student, Integer>(**Comparator obj**);

|  |  |
| --- | --- |
| [clear()](https://www.geeksforgeeks.org/treemap-clear-method-in-java/) | The method removes all mappings from this TreeMap and clears the map. |
| [clone()](https://www.geeksforgeeks.org/treemap-clone-method-in-java/) | The method returns a shallow copy of this TreeMap. |
| [containsKey(Object key)](https://www.geeksforgeeks.org/treemap-containskey-method-in-java/) | Returns true if this map contains a mapping for the specified key. |
| [containsValue(Object value)](https://www.geeksforgeeks.org/treemap-containsvalue-method-in-java/) | Returns true if this map maps one or more keys to the specified value. |
| [**entrySet()**](https://www.geeksforgeeks.org/treemap-entryset-method-in-java/) | Returns a set view of the mappings contained in this map. |
| [**firstKey()**](https://www.geeksforgeeks.org/java-util-treemap-firstentry-firstkey-java/) | Returns the first (lowest) key currently in this sorted map. |
| [**get(Object key)**](https://www.geeksforgeeks.org/treemap-get-method-in-java/) | Returns the value to which this map maps the specified key. |
| [**keySet()**](https://www.geeksforgeeks.org/treemap-keyset-method-in-java/) | The method returns a Set view of the keys contained in the treemap. |
| [**lastKey()**](https://www.geeksforgeeks.org/treemap-lastkey-method-in-java/) | Returns the last (highest) key currently in this sorted map. |
| [**put(Object key, Object value)**](https://www.geeksforgeeks.org/treemap-put-method-in-java/) | The method is used to insert a mapping into a map. |
| [**remove(Object key)**](https://www.geeksforgeeks.org/treemap-remove-method-in-java/) | Removes the mapping for this key from this TreeMap if present. |
| [**size()**](https://www.geeksforgeeks.org/treemap-size-method-in-java/) |  |
| [**subMap((K startKey, K endKey)**](https://www.geeksforgeeks.org/treemap-submap-method-in-java/) | The method returns the portion of this map whose keys range from startKey, inclusive, to endKey, exclusive. |
| [**values()**](https://www.geeksforgeeks.org/treemap-values-method-in-java/) | Returns a collection view of the values contained in this map. |

For Iterating a Map

for (Map.Entry<String, Integer> e : **mapobj**.entrySet())

            System.out.println("Key: " + e.getKey() + " Value: " + e.getValue());

# HashMap

 It allows to store the null keys as well, but there should be only one null key object

This class makes no guarantees as to the order of the map.

Default hashmap will have ,HashMap sized with initial capacity 16.

Internally HashMap contains an array.

|  |  |
| --- | --- |
| [containsKey(Object key)](https://www.geeksforgeeks.org/hashmap-containskey-method-in-java/) | Returns true if this map contains a mapping for the specified key. |
| [containsValue(Object value)](https://www.geeksforgeeks.org/hashmap-containsvalue-method-in-java/) | Returns true if this map maps one or more keys to the specified value. |
| [entrySet()](https://www.geeksforgeeks.org/hashmap-entryset-method-in-java/) | Returns a Set view of the mappings contained in this map. |
| [get(Object key)](https://www.geeksforgeeks.org/hashmap-get-method-in-java/) | Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key. |
| [isEmpty()](https://www.geeksforgeeks.org/hashmap-isempty-method-in-java/) | Returns true if this map contains no key-value mappings. |
| [keySet()](https://www.geeksforgeeks.org/hashmap-keyset-method-in-java/#:~:text=util.,the%20key%20elements%20in%20them.) | Returns a Set view of the keys contained in this map. |
| [put(K key, V value)](https://www.geeksforgeeks.org/hashmap-put-method-in-java/) | Associates the specified value with the specified key in this map. |
| [remove(Object key)](https://www.geeksforgeeks.org/hashmap-remove-method-in-java/) | Removes the mapping for the specified key from this map if present. |
| [size()](https://www.geeksforgeeks.org/hashmap-size-method-in-java/#:~:text=HashMap.,or%20mappings%20in%20the%20Map.&text=Parameters%3A%20The%20method%20does%20not,pairs%20present%20in%20the%20map.) | Returns the number of key-value mappings in this map. |
| [values()](https://www.geeksforgeeks.org/hashmap-values-method-in-java/) | Returns a Collection view of the values contained in this map. |
| [clear()](https://www.geeksforgeeks.org/hashmap-clear-method-in-java/#:~:text=HashMap.,mappings%20from%20a%20specified%20HashMap.&text=Parameters%3A%20The%20method%20does%20not,does%20not%20return%20any%20value.) | Removes all of the mappings from this map. |

**Linked HashMap**

[**https://www.geeksforgeeks.org/linkedhashmap-class-java-examples/**](https://www.geeksforgeeks.org/linkedhashmap-class-java-examples/)

**Strings**

[**https://www.javatpoint.com/java-string**](https://www.javatpoint.com/java-string)

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.

**in Java, string is an object that represents a sequence of characters.**

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.

#### **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).

1. String s=**new** String("Welcome");

In such case, [**JVM**](https://www.javatpoint.com/jvm-java-virtual-machine)**will create a new string object in normal (non-pool) heap memory**, and the **literal "Welcome" will be placed in the string constant pool.**

|  |  |
| --- | --- |
| [char charAt(int index)](https://www.javatpoint.com/java-string-charat) | returns char value for the particular index |
| [int length()](https://www.javatpoint.com/java-string-length) | returns string length |
| [String substring(int beginIndex)](https://www.javatpoint.com/java-string-substring) | returns substring for given begin index. |
| [String substring(int beginIndex, int endIndex)](https://www.javatpoint.com/java-string-substring) | returns substring for given begin index and end index. |
| [boolean contains(CharSequence s)](https://www.javatpoint.com/java-string-contains) | returns true or false after matching the sequence of char value. |
| [boolean equals(Object another)](https://www.javatpoint.com/java-string-equals) | checks the equality of string with the given object. |
| [boolean isEmpty()](https://www.javatpoint.com/java-string-isempty) | checks if string is empty. |
| [String concat(String str)](https://www.javatpoint.com/java-string-concat) | concatenates the specified string. |
| [String replace(char old, char new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of the specified char value. |
| [String[] split(String regex)](https://www.javatpoint.com/java-string-split) | returns a split string matching regex. |
| [String[] split(String regex, int limit)](https://www.javatpoint.com/java-string-split) | returns a split string matching regex and limit. |
| [int indexOf(int ch)](https://www.javatpoint.com/java-string-indexof) | returns the specified char value index. |
| [String toLowerCase()](https://www.javatpoint.com/java-string-tolowercase) | returns a string in lowercase. |
| [String toUpperCase()](https://www.javatpoint.com/java-string-touppercase) | returns a string in uppercase. |
| [String trim()](https://www.javatpoint.com/java-string-trim) | removes beginning and ending spaces of this string. |
| [static String valueOf(int value)](https://www.javatpoint.com/java-string-valueof) | converts given type into string. It is an overloaded method. |

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.

### Why string objects are immutable in java?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Because java uses the concept of string literal.  Suppose there are 5 reference variables,all referes to one object "sachin".  If one reference variable changes the value of the object, it will be affected to all the reference variables.  That is why string objects are immutable in java.  There are three ways to compare String in Java:   1. By Using equals() Method 🡪 compares values of 2 strings 2. By Using == Operator 🡪 The == operator compares references not values. 3. By compareTo() Method  **Java StringBuffer Class** <https://www.javatpoint.com/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.   |  |  | | --- | --- | | StringBuffer() | It creates an empty String buffer with the initial capacity of 16. | | StringBuffer(String str) | It creates a String buffer with the specified string.. |   **append(String s) also** The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double).  **Conversions**   1. **Char to Int and viceversa**   Char ch = ‘a’  Int I = Integer.parseInt(String.valueOf(ch));  Int j = Character.getNumericValue(ch); // works for negative numbers as well.  Char ch = (char) int\_val   1. **String to Integer and vice versa**   Int I = Integer.valueOf(string);  int x = Integer.parseInt(string);  String y = String.valueOf(int)  String x = Integer.toString(int)  **Generics and MultiThreading, Java I/O pending.** |
|  |
|  |