**OOP & Collections in JAVA**

**Submitted by**

**Srivaths.P**

**Table of Contents:**

**Topic Page No.**

**OOP ……………………………………………………………. …1**

**Abstraction…………………………………………….………… 3 - 6**

**Encapsulation……………………………………………..…….. 7 - 9**

**Inheritance……………………………………………………….10 - 18**

**Polymorphism………………………………………………...….19 - 22**

**Differences………………………………………………….….…....25**

**Collections………………………………………………….….….....24**

**Array List………………………………………………………...25 - 27**

**Linked List………………………………………………………..27 - 28**

**Vector Class……………………………………………..…….…….29**

**Stack Class……………………………………………………..……30**

**Hash Set………………………………………………….…….….32 - 33**

**LinkedHash Set………………………………………………..….34 - 35**

**Tree Set……………………………………………………………35 - 36**

**OOP**

**(Object Oriented Programming)**

## Object-oriented programming (OOP) is a programming language model in which programs are organized around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior. In fact all java programs are at least some extent object- oriented. They are A[bstraction](https://stackify.com/oop-concept-abstraction/)**,** [Encapsulation](https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/), Inheritance and Polymorphism.

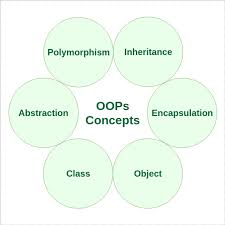
There are four main OOP concepts in Java. These are:

**Abstraction.** Abstraction means using simple things to represent complexity. In Java, abstraction means simple things like **objects**, **classes**, and **variables** represent more complex underlying code and data. This is important because it lets avoid repeating the same work multiple times.

**Encapsulation.** This is the practice of keeping fields within a class private, then providing access to them via public methods. It’s a protective barrier that keeps the data and code safe within the class itself. This way, we can re-use objects like code components or variables without allowing open access to the data system-wide.

**Inheritance.** This is a special feature of Object Oriented Programming in Java. It lets programmers create new classes that share some of the attributes of existing classes. This lets us build on previous work without reinventing the wheel.

**Polymorphism.** This Java OOP concept lets programmers use the same word to mean different things in different contexts. One form of polymorphism in Java is **method overloading**. That’s when different meanings are implied by the code itself. The other form is **method overriding**. That’s when the different meanings are implied by the values of the supplied variables.



**Abstraction**

Abstraction as an OOP concept in Java works by letting programmers create useful, reusable tools. For example, a programmer can create several different types of **objects**. These can be variables, functions, or data structures. Programmers can also create different **classes** of objects. These are ways to define the objects.

For instance, a class of variable might be an address. The class might specify that each address object shall have a name, street, city, and zip code. The objects, in this case, might be employee addresses, customer addresses, or supplier addresses.

Abstraction in Java can be achieved using Abstract Class and Abstract Method.

### Abstract Class

A class which is declared “abstract” is called as an abstract class. It can have abstract methods as well as concrete methods. A normal class cannot have abstract methods.

### Abstract Method

A method without a body is known as an Abstract Method. It must be declared in an abstract class. The abstract method will never be final because the abstract class must implement all the abstract methods.

**Rules of Abstract Method**

* Abstract methods do not have an implementation; it only has method signature
* If a class is using an abstract method they must be declared abstract. The opposite cannot be true. This means that an abstract class does not necessarily have an abstract method.
* If a regular class extends an abstract class, then that class must implement all the abstract methods of the abstract parent.

### Advantages of Abstraction

* The main benefit of using an abstract class is that it allows you to group several related classes as siblings.
* Abstraction helps to reduce the complexity of the design and implementation process of software.

**For Example:**

**package** oopc;

**import** org.thoughts.on.java.coffee.CoffeeException;

**import** java.utils.Map;

**public** **class** Abstraction {

**private** Map<CoffeeSelection, CoffeeBean> beans;

**public** CoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans) {

**this**.beans = beans

}

**public** Coffee brewCoffee(CoffeeSelection selection) **throws** CoffeeException {

Coffee coffee = **new** Coffee();

System.***out***.println(“Making coffee ...”);

**return** coffee;

}

}

Coffee Selection is a simple enum providing a set of predefined values for the different kinds of coffees.

**public** **enum** CoffeeSelection { ***FILTER\_COFFEE***, ***ESPRESSO***, ***CAPPUCCINO***;}

**public** **class** CoffeeBean {

**private** String name;

**private** **double** quantity;

**public** CoffeeBean(String name, **double** quantity) {

**this**.name = name;

**this**.quantity;

}

}

**public** **class** Coffee {

**private** CoffeeSelection selection;

**private** **double** quantity;

**public** Coffee(CoffeeSelection, **double** quantity) {

**this**.selection = selection;

**this**. quantity = quantity;

}

}

Prepare a Map of the available CoffeeBeans, instantiate a **new** CoffeeMachine object, and call the brewCoffee method with your preferred CoffeeSelection.

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** CoffeeApp {

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

// create a Map of available coffee beans

Map<CoffeeSelection, CoffeeBean> beans = **new** HashMap<CoffeeSelection, CoffeeBean>();

beans.put(CoffeeSelection.ESPRESSO,

**new** CoffeeBean("My favorite espresso bean", 1000));

beans.put(CoffeeSelection.FILTER\_COFFEE,

**new** CoffeeBean("My favorite filter coffee bean", 1000));

// get a new CoffeeMachine object

CoffeeMachine machine = **new** CoffeeMachine(beans);

// brew a fresh coffee

**try** {

Coffee espresso = machine.brewCoffee(CoffeeSelection.ESPRESSO);

} **catch**(CoffeeException e) {

e.printStackTrace();

}

} // end main

} // end CoffeeApp

You can see in this example that the abstraction provided by the Coffee Machine class hides all the details of the brewing process. That makes it easy to use and allows each developer to focus on a specific class.

So, abstraction basically hides the information and provides only that information which the programmer wants the user to know. The trivial information for the user is kept hidden.

### Encapsulation

Encapsulation lets us re-use functionality without jeopardizing security. It’s a powerful OOP concept in Java because it helps us save a lot of time. For example, we may create a piece of code that calls specific data from a database. It may be useful to reuse that code with other databases or processes. Encapsulation lets us do that while keeping our original data private. It also lets us alter our original code without breaking it for others who have adopted it in the meantime.

**For Example**:

**package** com.whitesheet

**public** **class** Student {

**private** String name;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name

}

//save as Test.java

**package** com.whitesheet

**class** Test {

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

Student s = **new** Student();

s.setName(“sailo”);

System.out.println(s.getName());

}

}

**Output: sailo**

**Data Hiding in Java**

Frequently, Java encapsulation is referred as **data hiding**. But more than data hiding, encapsulation concept is meant for better management or grouping of related data.

To achieve a lesser degree of encapsulation in Java, you can use modifiers like "protected" or "public". With encapsulation, developers can change one part of the code easily without affecting other.

**Getter and Setter Methods in Java**

If a data member is declared "private", then it can only be accessed within the same class. No outside class can access data member of that class. If you need to access these variables, you have to use public "getter" and "setter" methods.

Getter and Setter's methods are used to create, modify, delete and view the variables values.

**For Example**

**public** **class** Employee

{

**private** **int** empNum;

**private** String empName;

**public** **int** getEmpNum()

{

**return** empNum;

}

**public** **void** setEmpNum(**int** empNum)

{

**this**.empNum = empNum;

}

**public** String getEmpName()

{

**return** empName;

}

**public** **void** setEmpName(String empName)

{

**this**.empName = empName;

}

}// Sample class end.

**public** **class** Test {

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

Employee employee1 = **new** Employee();

employee1.setEmpNum(1001);

employee1.setEmpName("nrit1");

**int** eno = employee1.getEmpNum();

String name = employee1.getEmpName();

System.***out***.println(eno + "\t" + name + "\t" );

}

}// end of Test class.

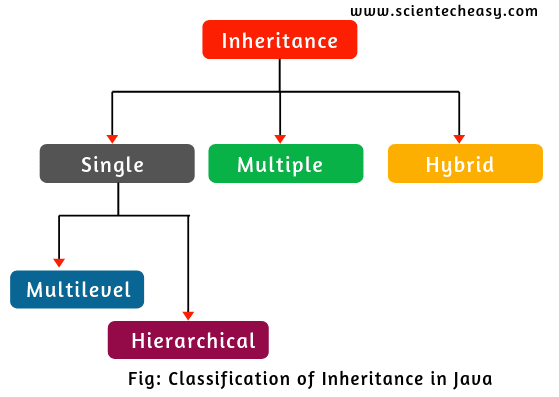
**Advantages of Encapsulation in Java**

* Encapsulation is binding the data with its related functionalities. Here functionalities mean "methods" and data means "variables"
* So we keep variable and methods in one place. That place is "class." Class is the base for encapsulation.
* With Java Encapsulation, you can hide (restrict access) to critical data members in your code, which improves security
* As we discussed earlier, if a data member is declared "private", then it can only be accessed within the same class. No outside class can access data member (variable) of other class.
* However, if you need to access these variables, you have to use **public "getter" and "setter"** methods.

So, encapsulation is defined as the wrapping up of data under a single unit. it is a protective class that prevents the data from being accessed by the code outside that class.

### Inheritance

Inheritance is another labor-saving Java OOP concept. It works by letting a new class adopt the properties of another. We call the inheriting class a **subclass** or a **child class**. The original class is often called the **parent**. We use the keyword **extends** to define a new class that inherits properties from an old class.



* **Super Class:** The class whose features are inherited is known as super class(or a base class or a parent class).
* **Sub Class:** The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
* **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

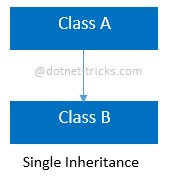
OOPs support the six different types of inheritance as given below :

* Single inheritance
* Multi-level inheritance
* Multiple inheritance
* Multipath inheritance
* Hierarchical Inheritance
* Hybrid Inheritance

### Single inheritance

In this inheritance, a derived class is created from a single base class.

In the given example, Class A is the parent class and Class B is the child class since Class B inherits the features and behavior of the parent class A.



**For Example:**

**public** **class** Sample {

**int** a;

**int** b;

**public** **void** setAB(**int** x, **int** y)

{

a=x;

b=y;

}

**public** **void** dispAB()

{

System.***out***.println (a+"\t"+b);

}

}

**public** **class** Test **extends** Sample {

**int** c;

**int** d;

**public** **void** setData(**int** x, **int** y, **int** z, **int** k)

{

setAB(x,y);

c=z;

d=k;

}

**public** **void** display()

{

dispAB();

System.***out***.println (c+"\t"+d);

}

}

**public** **class** InheritTest {

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

Test t1 = **new** Test();

t1.setData(10,20,30,40);

t1.display();

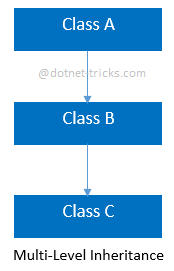
}

}

### Multi-level inheritance

In this inheritance, a derived class is created from another derived class.

In the given example, class c inherits the properties and behavior of class B and class B inherits the properties and behavior of class B. So, here A is the parent class of B and class B is the parent class of C. So, here class C implicitly inherits the properties and behavior of class A along with Class B i.e there is a multilevel of inheritance.



**For Example:**

**Public class** Test

{

**Public void** display()

{

System.***out***.println ("Test display");

}

}

**Public class** TestOne **extends** Test

{

**Public void** display()

{

System.***out***.println ("TestOne display begining");

**super**.display();

System.***out***.println ("TestOne display end");

}

}

**public class** TestTwo **extends** TestOne

{

**public void** display()

{

System.***out***.println ("TestTwo Display Beg");

**super**.display();

System.***out***.println ("TestTwo Display end");

}

}

**public class** InheritanceTest {

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

TestTwo t = **new** TestTwo();

t.display();

}

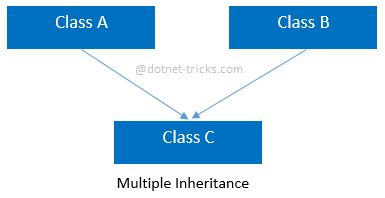
}

}

### Multiple inheritance

In this inheritance, a derived class is created from more than one base class. This inheritance is not supported by .NET Languages like C#, F# etc. and Java Language.

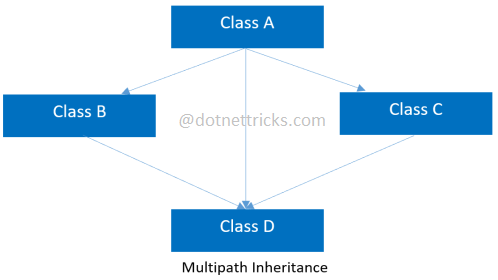
In the given example, class c inherits the properties and behavior of class B and class A at same level. So, here A and Class B both are the parent classes for Class C.



**Multipath inheritance**

In this inheritance, a derived class is created from another derived classes and the same base class of another derived classes. This inheritance is not supported by .NET Languages like C#, F# etc.

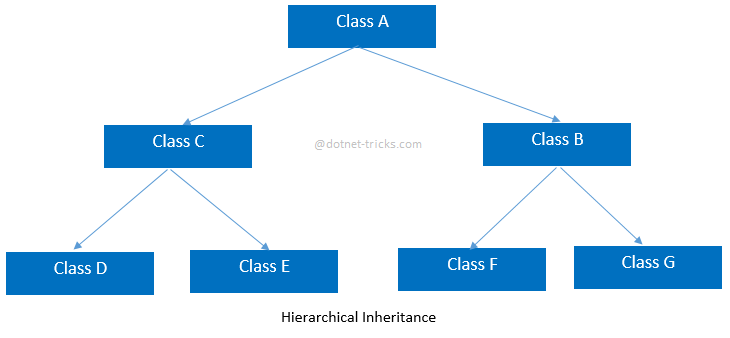
In the given example, class D inherits the properties and behavior of class C and class B as well as Class A. Both class C and class B inherits the Class A. So, Class A is the parent for Class B and Class C as well as Class D. So it's making it Multipath inheritance.



### Hierarchical Inheritance

In this inheritance, more than one derived classes are created from a single base class and futher child classes act as parent classes for more than one child classes.

In the given example, class A has two childs class B and class D. Further, class B and class C both are having two childs - class D and E; class F and G respectively.



### For Example:

**Public class** Person {

**privateint**pid;

**private** String name;

**private** String gender;

**privateint**age;

**publicvoid** setData(**int**pid, String name, String gender, **int**age) {

**this**.pid = pid;

**this**.name = name;

**this**.gender = gender;

**this**.age = age;

}

**publicvoid** display() {

System.***out***.println(pid + "\t" + name + "\t" + gender + "\t" + age);

}

}

**publicclass** Student **extends** Person {

String course;

**double**feePaid;

**double**feeDue;

**char**grade;

**publicvoid** setData(**int**pid, String name, String gender, **int**age, String course, **double**feePaid, **double**feeDue, **char**grade) {

setData(pid, name, gender, age);

**this**.course = course;

**this**.feePaid = feePaid;

**this**.feeDue = feeDue;

**this**.grade = grade;

}

**publicvoid** display() {

**super**.display();

System.***out***.println(course + "\t" + feePaid + "\t" + feeDue + "\t" + grade);

}

}

**Public class** Employee **extends** Person {

String dept;

**double**salary;

String desg;

**publicvoid** setData(**int**pid, String name, String gender, **int**age, String dept, **double**salary, String desg) {

setData(pid, name, gender, age);

**this**.dept = dept;

**this**.salary = salary;

**this**.desg = desg;

}

**publicvoid** display() {

**super**.display();

System.***out***.println(dept + "\t" + salary + "\t" + desg);

}

}

**Public class** HierachialTest {

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

Student student = **new** Student();

student.setData(1001, "abc", "male", 25, "java", 10000, 5000, 'A');

student.display();

Employee employee = **new** Employee();

employee.setData(101, "Mohan", "Male", 40, "IT", 25000, "admin");

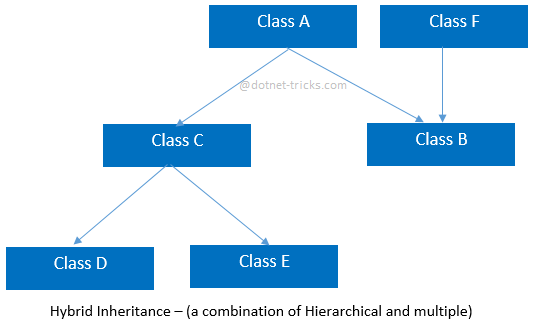
employee.display();

}

}

### Hybrid inheritance

This is combination of more than one inheritance. Hence, it may be a combination of Multilevel and Multiple inheritance or Hierarchical and Multilevel inheritance or Hierarchical and Multipath inheritance or Hierarchical, Multilevel and Multiple inheritance.



**Advantages of Inheritance:**

* Reduce code redundancy.
* Provides code reusability.
* Reduces source code size and improves code readability.
* The code is easy to manage and divided into parent and child classes.
* Supports code extensibility by overriding the base class functionality within child classes.

### Disadvantages of Inheritance:

* In Inheritance base class and child class, both are tightly coupled. Hence If you change the code of parent class, it will affect all the child classes.
* In a class hierarchy, many data members remain unused and the memory allocated to them is not utilized. Hence it affects the performance of your program if you have not implemented inheritance correctly.

Inheritance is to provide the reusability of code so that a class has to write only the unique data and rest of the common properties and functionalities can be extended from the another class.

### Polymorphism

**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 form. In Java it works by using a reference to a parent class to affect an object in the child class. Two more examples of polymorphism in Java are method overriding and method overloading.

In **method** **overriding**, the child class can use the OOP polymorphism concept to override a method of its parent class. That allows a programmer to use one method in different ways depending on whether it’s invoked by an object of the parent class or an object of the child class.

In **method overloading,** a single method may perform different functions depending on the context in which it’s called. That is, a single method name might work in different ways depending on what arguments are passed to it.

**Method Overloading**

Method overloading takes place between the same method with different signature

Method signature defines,

name of the method, no.of args, type of args and return type of the method.

Method overloading is a compile time process.

First compiler checks no.of arguments, if no.of args are matched then cheks for type of arguments, if type is also matched then ambiguity b/w methods and compiler shows error massage.

Method over loading is compile time binding (or) static binding.

**Method overloading with data :**

**public** **class** Sample

{

**private** inta;

**private** intb;

**private** intc;

**public** **void** setData() {

a=0;

b=0;

c=0;

}

**public** **void** setData(intx) {

a = x;

b = 0;

c = 0;

}

**public** **void** setData(intx, inty) {

a = x;

b = y;

c = 0;

}

**public** **void** setData(intx, inty, intz) {

a = x;

b = y;

c = z;

}

**public** **void** display() {

System.***out***.println ("a="+a+"\tb="+b+"\tc="+c);

}

}

**public** **class** Test {

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

Sample s1 = **new** Sample();

Sample s2 = **new** Sample();

Sample s3 = **new** Sample();

Sample s4 = **new** Sample();

s1.setData();

s2.setData(15);

s3.setData(10,20);

s4.setData(15,25,30);

System.***out***.println ("First Object");

s1.display();

System.***out***.println ("Second Object");

s2.display();

System.***out***.println ("Third Object");

s3.display();

System.***out***.println ("Fourth Object");

s4.display();

}

}

**Example of Java Runtime Polymorphism**

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

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

**Example:**

**class** Bike{

**void** run(){System.out.println("running");

}

**class** Splendor **extends** Bike{

**void** run(){System.out.println("60km");}

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

Bike b = **new** Splendor();//upcasting

b.run();

 }

}

**Output:** 60km.

Polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.The only possible way to access an object is through a reference variable. A reference variable can be of only one type. once declared, the type of a reference variable cannot be changed.

### Difference between Abstraction and Encapsulation.

|  |  |
| --- | --- |
| **Abstraction** | **Encapsulation** |
| Abstraction solves the issues at the design level. | Encapsulation solves it implementation level. |
| Abstraction is about hiding unwanted details while showing most essential information. | Encapsulation means binding the code and data into a single unit. |
| Abstraction allows focussing on what the information object must contain | Encapsulation means hiding the internal details or mechanics of how an object does something for security reasons. |

### Difference between Abstract Class and Interface.

|  |  |
| --- | --- |
| **Abstract Class** | **Interface** |
| An abstract class can have both abstract and non-abstract methods. | The interface can have only abstract methods. |
| It does not support multiple inheritances. | It supports multiple inheritances. |
| It can provide the implementation of the interface. | It can not provide the implementation of the abstract class. |
| An abstract class can have protected and abstract public methods. | An interface can have only have public abstract methods. |
| An abstract class can have final, static, or static final variable with any access specifier. | The interface can only have a public static final variable. |

**Collections**

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects. Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion. Java Collection means a single unit of objects.



# Array List

ArrayList is a part of collection framework and is present in java.util package. It provides us dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.

ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases.

**Syntax**: ArrayList al = new ArrayList();//creating old non-generic arraylist

In generic collection, we specify the type in angular braces. Now ArrayList is forced to have only specified type of objects in it. If you try to add another type of object, it gives compile time error.

**Example of Java ArrayList class**

**public** **class** Array {

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

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

al.add(**new** Integer(10));

al.add(215);

al.add(95);

al.add(65);

al.add(55);

al.add(45);

System.***out***.println("array elements"+al);

System.***out***.println();

//using for loop

**for**(**int** i = 0; i<al.size(); i++)

{

System.***out***.println(al.get(i));

}

//for each loop

**for**(Integer i:al)

{

System.***out***.println(i);

}

// using interator

Iterator <Integer> iterator = al.iterator();

**while**(iterator.hasNext())

{

Integer nextInteger = iterator.next();

System.***out***.println(nextInteger);

}

// no. of elements

System.***out***.println("no. of elements"+al.size());

// to display first element

System.***out***.println("First element"+al.get(0));

// last element

System.***out***.println("last element"+al.get(al.size()-1));

// adding a no. in the string

al.add(0, 144);

// removing element

Integer i1= al.remove(4);

{

System.***out***.println("after removing"+al);

}

Integer i2 = al.get(0);// read only, will not remove physically

System.***out***.println(i2);

System.***out***.println(al);

//clear the array list

al.clear();

{

System.***out***.println(al);

}

}

}

**Output:**array elements[10, 215, 95, 65, 55, 45]

no. of elements6

First element10

last element45

after removing[144, 10, 215, 95, 55, 45]

144

[144, 10, 215, 95, 55, 45]

[]

**Linked List class**

Java LinkedList class uses doubly linked list to store the elements. It extends the AbstractList class and implements List and Deque interfaces.

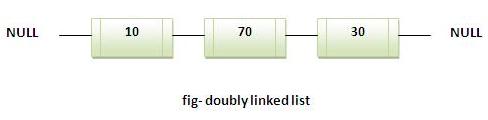
Java LinkedList class can contain duplicate elements.

Java LinkedList class maintains insertion order.

Java LinkedList class is non synchronized.

In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.

Java LinkedList class can be used as list, stack or queue.



**For Example:**

**import** java.util.\*;

**public** **class** Test

{

**public** **static** **void** main(String [] args)

{

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

al.add("R");

al.add("V");

al.add("R");

al.add("A");

Iterator<String> itr=al.iterator();

**while**(itr.hasNext())

{

System.***out***.println (itr.next());

}

}

}

**Output**:R V R A

**Vector class**

The Vector class implements a growable array of objects. Vectors basically fall in legacy classes but now it is fully compatible with collections.

* Vector implements a dynamic array that 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 synchronised and have some legacy method which collection framework does not contain.
* It extends **AbstractList** and implements **List** interfaces.

**Constructor:**

* **Vector()**: Creates a default vector of initial capacity is 10.
* **Vector(int size):** Creates a vector whose initial capacity is specified by size.
* **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 that a vector is resized upward.
* **Vector(Collection c):** Creates a vector that contains the elements of collection c.

# Program: How to copy vector to array.

**import** java.util.Vector;

**publicclass** MyVectorArrayCopy {

**publicstaticvoid** main(String a[]) {

Vector<String>vct = **new** Vector<String>();

vct.add("First");

vct.add("Second");

vct.add("Third");

vct.add("Random");

System.***out***.println("Actual vector:" + vct);

String []copyArr = **new** String[vct.size()];

vct.copyInto(copyArr);

System.***out***.println("Copied Array content:");

**for** (String str : copyArr) {

System.***out***.println(str);

}

}

}

**Stack Class**

**Stack** is a subclass of **Vector** that implements a standard last-in, first-out stack. **Stack** only defines the default constructor, which creates an empty stack. **Stack** includes all the methods defined by **Vector**, and adds several of its own.

To put an object on the top of the stack, call **push()**. To remove and return the top element, call **pop()**. An EmptyStackExceptionis thrown if you call **pop( )** when the invoking stack is empty. You can use **peek( )** to return, but not remove, the top object.

The **empty()** method returns **true** if nothing is on the stack. The **search()** method determines whether an object exists on the stack, and returns the number of pops that are required to bring it to the top of the stack. Here is an example that creates a stack, pushes several **Integer** objects onto it, and then pops them off again:

**For Example:**

**import** java.util.Stack;

**public** **class** Test {

**public** staticvoid main(String[] args) {

Stack<String>bookRack = **new** Stack<String>();

bookRack.push("HADOOP");

bookRack.push("JAVA");

bookRack.push("ORACLE");

bookRack.push("C");

bookRack.push("LINUX");

**while**(! bookRack.empty())

{

String book = bookRack.pop();

System.***out***.println(book);

}

}

**Set**

List can contain duplicate elements whereas Set contains unique elements only.

# HashSet

The HashSet class implements the Set interface, backed by a hash table which is actually a HashMap instance. This class permits the null element. The class also offers constant time performance for the basic operations like add, remove, contains and size assuming the hash function disperses the elements properly among the buckets, which we shall see further in the article.

Few important features of HashSet are:

* Implements [Set Interface](https://www.geeksforgeeks.org/set-in-java/).
* Underlying data structure for HashSet is hashtable.
* As it implements the Set Interface, duplicate values are not allowed.
* Objects that you insert in HashSet are not guaranteed to be inserted in same order. Objects are inserted based on their hash code.
* NULL elements are allowed in HashSet.
* HashSet also implements Searlizable and Cloneable interfaces.

**For Example:**

**public** **static** **void** main(String[]args)

{

HashSet<String> h = **new** HashSet<String>();

 // Adding elements into HashSet usind add()

h.add("India");

h.add("Australia");

h.add("South Africa");

h.add("India");// adding duplicate elements

 // Displaying the HashSet

System.out.println(h);

System.out.println("List contains India or not:" +

                           h.contains("India"));

 // Removing items from HashSet using remove()

 h.remove("Australia");

System.out.println("List after removing Australia:"+h);

// Iterating over hash set items

System.out.println("Iterating over list:");

Iterator<String> i = h.iterator();

**while** (i.hasNext())

System.out.println(i.next());

}

}

**Output:** [South Africa, Australia, India]

True

South Africa, India]

South Africa

India

**Internal working of a HashSet**  
All the classes of Set interface internally backed up by Map. HashSet uses HashMap for storing its object internally. You must be wondering that to enter a value in HashMap we need a key-value pair, but in HashSet we are passing only one value.

# LinkedHashSet

A 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.

**Syntax:** LinkedHashSet<String> hs = new LinkedHashSet<String>();

**LinkedHashSet():** This constructor is used to create a default HashSet.

**LinkedHashSet(Collection C):** Used in initializing the HashSet with the eleements of the collection C

**LinkedHashSet(int size):** Used to initialize the size of the LinkedHashSet with the integer mentioned in the parameter.

**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.

**For Example:**

**import** java.util.LinkedHashSet;

**public** **static** **void** main(String[] args)

   {

LinkedHashSet<String> linkedset = **new** LinkedHashSet<String>();

//Adding element to LinkedHashSet

linkedset.add("A");

linkedset.add("B");

linkedset.add("C");

linkedset.add("D");

// This will not add new element as A already exists

linkedset.add("A");

linkedset.add("E");

System.out.println("Size of LinkedHashSet = " +  linkedset.size());

System.out.println("Updated LinkedHashSet: " + linkedset);

}

}

**Output:**Size of LinkedHashSet=5

Updated LinkedHashSet: [A, B, C, E]

**TreeSet**

TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements NavigableSet interface. The objects of TreeSet class are stored in ascending order.

* Contains unique elements only like HashSet.
* Access and retrieval times are quiet fast.
* Maintains ascending order.

**For Example:**

**import** java.util.Iterator;

**import** java.util.TreeSet;

**public** **class** Test

{

**public** **static** **void** main(String [] args)

{

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

al.add("S");

al.add("V");

al.add("K");

al.add("A");

// Traversing elements

Iterator<String>itr = al.iterator();

**while** (itr.hasNext())

{

System.***out***.println (itr.next());

}

}

}

**Output:**SVKA

Collections is an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. It reduces programming effort, and usability of data in different classes.