# What Are Marker Interfaces In Java?

**Marker interfaces** in java are interfaces with no members declared in them. They are just an empty interfaces used to mark or identify a special operation. For example, Cloneable interface is used to mark cloning operation and Serializable interface is used to mark serialization and deserialization of an object. Marker interfaces give instructions to JVM that classes implementing them will have special behavior and must be handled with care.

Marker interfaces don’t provide any functionality. In earlier versions of Java (Before Java 5), marker interfaces are used to provide metadata to the readers. With the introduction of **annotations** from Java 5, annotations are used more instead of marker interfaces to provide metadata . But, still many use marker interfaces to mark the special behavior of a class.

## **Java’s built-in Marker Interfaces :**

These are some built-in marker interfaces in java which are used to mark some special behavior of a class.

**1) java.lang.Cloneable Interface :**

This interface is used to mark the cloning operation. An object of a class which implements Cloneable interface is eligible for field-by-field copying of an object.

**2) java.io.Serializable Interface :**

This interface is used to mark serialization and deserialization of an object. Serialization is a process in which an object state is read from memory and written into a file or a database. Deserialization is a process in which an object state is read from a file or a database and written back into memory. Any class which wants it’s object to be eligible for serialization and deserialization must implement Serializable interface.

**3) java.util.EventListener :**

This is also a marker interface which must be extended by all event listener interfaces.

**4) java.rmi.Remote :**

This is also a marker interface which is used to mark the invocation of a method remotely. Only methods of those classes which implement Remote interface are eligible for invocation by non-local virtual machine.

## **User Defined Marker Interfaces :**

You can define your own marker interfaces to indicate about any special behavior. Below is such an example. In this example, Cash and Checque are two marker interfaces which are used to indicate whether payment is done by cash or checque.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | interface Cash  {    }    interface Checque  {    }    class PaymentOne implements Cash  {      static void paymentByCash()      {          System.out.println("Payment is done by cash");      }  }    class PaymentTwo implements Checque  {      static void paymentByChecque()      {          System.out.println("Payment is done by checque");      }  }    public class MainClass  {      public static void main(String[] args)      {          PaymentOne.paymentByCash();            PaymentTwo.paymentByChecque();      }  } |

## **Alternative To Marker Interfaces :**

1)  **Internal flags** can be used instead of marker interfaces to indicate any special operation.

2)  **Annotations** are recommended instead of marker interfaces to indicate any special operation.

**Generics**

**1. Why We Need Generics In Java?**

Errors are integral part of coding. Some errors occur at compile time and some errors occur at run time. Errors which occur at compile time can be easily identified and can be removed. But, run time errors occur when an application is running in real time. If they happen, they cause abrupt termination of an application.

ClassCastException is also such an exception which happens only at run time. It occurs when data of one type can not be casted to another type. You will never get a single clue about this exception during compilation. Look at the below code which throws ClassCastException at run time. But, you will never be get notified about this exception at compile time.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class GenericsInJava  {      public static void main(String[] args)      {          ArrayList list = new ArrayList();            list.add("JAVA");            list.add(123);            for (Object object : list)          {              //Below statement throws ClassCastException at run time                String str = (String) object;       //Type casting                System.out.println(str);          }      }  } |

In this example, ‘list’ contains elements of String type as well as int type. When you try to cast it’s elements to string type in the for loop, element of string type is casted without throwing errors but element of int type throws ClassCastException.

You can avoid ClassCastException by using generics in your code. The above example can be re-written using generics like below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class GenericsInJava  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            list.add("JAVA");        //  list.add(123);       Compile time error            for (String str : list)          {              //No type casting needed. ClasscastException Never occurs                System.out.println(str);          }      }  } |

Now, ‘list’ is declared so that it can hold only string type. If you try to add elements of different type, it gives compile time error. Therefore, ClassCastException never occurs while executing the for loop.

Generics are introduced in Java 5 to provide the type checking at compile time. If you use generics, you need not to perform the type casting explicitly. Java compiler applies strong type checking if you use generics in your code and shows errors if the code violates the type safety. Thus removing the risk of ClassCastException.

Therefore, To write the type safety code and to remove the risk of ClassCastException at run time, we need generics

# 2. Defining Generic Class:

In the previous post, we have seen why we need to use generics. Generics are used to check the type compatibility at the compile time and hence removing the chances of occuring ClassCastException at run time. In this particular post, we will see how to define our own generic class.

## **Generic Class :**

The syntax for defining generic class is as follows,

|  |  |
| --- | --- |
| 1  2  3  4 | class Class\_Name<T1, T2, T3 ... Tn>  {      //Generic Type or Parameterized type  } |

Where T1, T2, T3 … Tn (T stands for Type) enclosed within angle brackets (<>) are called **type parameters** and class ‘**Class\_Name**‘ is called generic type or parameterized type.

Now, let’s try to define one generic class based on the above format.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class GenericClass<T>  {      T t;        public GenericClass(T t)      {          this.t = t;      }        public void setT(T t)      {          this.t = t;      }        public T getT()      {          return t;      }  } |

While creating an instance to the above generic class, you can pass any class type as a type parameter and that class type replaces generic ‘T’ for that object. For example, if you pass String type as a type parameter then String will be the type of variable ‘t’. If you pass Integer as type parameter than Integer will be the type of variable ‘t’.

In the other words, when you pass a type while creating an object to the generic class, that object works only with that type. For example, If you pass String type while creating an object to the above generic class then that object works only with String type. That means setT() method takes String type as an argument and getT() method returns String type. If you pass any other type to setT() method, it gives compile time error. Hence, strictly checking type casting during compilation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class GenericsInJava  {      public static void main(String[] args)      {          GenericClass<String> gen1 = new GenericClass<String>("It must be string");            gen1.setT("Value Changed");        //Passing String to setT() method            String s = gen1.getT();              //getT() method returning string            gen1.setT(new Integer(123));      //Compile time error. You can't pass Integer type to setT() method now            gen1.setT(new Double(23.56));    //Compile time error. You can't pass Double type to setT() method now      }  } |

If you create an object by using Integer type as a type parameter then that object works only with Integer type.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class GenericsInJava  {      public static void main(String[] args)      {          GenericClass<Integer> gen1 = new GenericClass<Integer>(new Integer(123));            gen1.setT(456);             //Passing Integer type to setT() method            Integer I = gen1.getT();      //getT() method returning Integer type            gen1.setT(new String("123"));      //Compile time error. You can't pass String type to setT() method now            gen1.setT(new Double(23.56));    //Compile time error. You can't pass Double type to setT() method now      }  } |

## **Generics Work Only With Derived Types :**

While creating an instance of generic class, you must pass only derived types. You can’t pass primitive types. If you pass primitive type, it gives compile time error. i.e generics works only with derived type.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class GenericsInJava  {      public static void main(String[] args)      {          GenericClass<int> gen1 = new GenericClass<int>(123);   //Error, can't use primitive type            GenericClass<float> gen2 = new GenericClass<float>(23.56);  //Error, can't use primitive type      }  } |

## **Objects Of Same Generic Class Differ Based On Their Type Parameters :**

Objects of same generic class differ depending upon their type parameters. For example, object of above generic class created using String type is not compatible with an object of same class created using Integer type.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class GenericsInJava  {      public static void main(String[] args)      {          GenericClass<String> gen1 = new GenericClass<String>("Value Of t");            GenericClass<Integer> gen2 = new GenericClass<Integer>(new Integer(20));            gen1 = gen2;        //Error : Type mismatch            gen2 = gen1;        //Error : Type mismatch      }  } |

## **Generic Class With Two Type Parameters :**

Below is an example of a generic class with two type parameters.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | class GenericClass<T1, T2>  {      T1 t1;        T2 t2;        public GenericClass(T1 t1, T2 t2)      {          this.t1 = t1;            this.t2 = t2;      }        public void setT1(T1 t1)      {          this.t1 = t1;      }        public T1 getT1()      {          return t1;      }        public void setT2(T2 t2)      {          this.t2 = t2;      }        public T2 getT2()      {          return t2;      }  }    public class GenericsInJava  {      public static void main(String[] args)      {          GenericClass<String, Integer> gen1 = new GenericClass<String, Integer>("Value of t1", new Integer(123));            GenericClass<Integer, String> gen2 = new GenericClass<Integer, String>(new Integer(123), "Value of t2");            System.out.println(gen1.getT1());       //Output : Value of t1            System.out.println(gen1.getT2());       //Output : 123            System.out.println(gen2.getT1());       //Output : 123            System.out.println(gen2.getT2());       //Output : Value of t2      }  } |

You can pass your own type while creating an instance to the generic class. Here is an example for that.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | class GenericClass<T>  {      T t;        public GenericClass(T t)      {          this.t = t;      }        public void setT(T t)      {          this.t = t;      }        public T getT()      {          return t;      }  }    class A  {      int i;        public A(int i)      {          this.i = i;      }  }    public class GenericsInJava  {      public static void main(String[] args)      {          GenericClass<A> gen1 = new GenericClass<A>(new A(10));     //Passing A-type as type parameter            GenericClass<A> gen2 = new GenericClass<A>(new A(20));     //Passing A-type as type parameter            System.out.println(gen1.getT().i);    //Output : 10            System.out.println(gen2.getT().i);    //Output : 20      }  } |

# 3. Rules To Follow While Implementing Generic Interfaces:

Like generic classes, you can also define generic interfaces. The same syntax used to define generic classes is also used to define generic interfaces. Here is an example of generic interface.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | interface GenericInterface<T>  {      void setT(T t);        T getT();  } |

While implementing generic interfaces you have to follow some rules. Below is the discussion of those rules.

## **Rules To Follow While Implementing Generic Interfaces :**

* Only generic classes can implement generic interfaces. Normal classes can’t implement generic interfaces. For example, above generic interface can be implemented as,

|  |  |
| --- | --- |
| 1  2  3  4 | class GenericClass<T> implements GenericInterface<T>  {    } |

Not like below. It gives compile time error.

|  |  |
| --- | --- |
| 1  2  3  4 | class NormalClass implements GenericInterface<T>  {       //Compile time error  } |

Here is the full implementation of above generic interface.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class GenericClass<T> implements GenericInterface<T>  {      T t;        //Implementing setT() method        @Override      public void setT(T t)      {          this.t = t;      }        //Implementing getT() method        @Override      public T getT()      {          return t;      }  } |

* A normal class can implement a generic interface if type parameter of generic interface is a wrapper class. For example, below implementation of GenericInterface is legal.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | interface GenericInterface<Integer>  {         //Generic interface with Integer as type parameter  }    class NormalClass implements GenericInterface<Integer>  {         //Normal class implementing generic interface  } |

* Class implementing generic interface at least must have same number and same type of parameters and at most can have any number and any type of parameters.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | interface GenericInterface<T>  {      //Generic interface with one type parameter  }    class GenericClass1<T> implements GenericInterface<T>  {      //Class with same type parameter  }    class GenericClass2<T, V> implements GenericInterface<T>  {      //Class with two type parameters  }    class GenericClass<T1, T2> implements GenericInterface<T>  {      //Compile time error, class having different type of parameters  } |

* You can change the type of parameter passed to generic interface while implementing it. When changed, the class which is implementing should have new type as parameter and also, you have to change old type with new type while implementing the methods.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | interface GenericInterface<T>  {      void setT(T t);        T getT();  }    //Changing the type of parameter passed to GenericInterface while implementing    class GenericClass<V> implements GenericInterface<V>  {      V t;        @Override      public void setT(V t)    //Changing the type of parameter      {          this.t = t;      }        @Override      public V getT()          //Changing the return type      {          return t;      }  } |

* Generic interface can have any number of type parameters. Class implementing generic interface at least must have  same type of parameters and at most can have any number of parameters

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | interface GenericInterface<T1, T2, T3, T4>  {      //Generic interface with 4 type parameters  }    class GenericClass1<T1, T2, T3, T4, T5> implements GenericInterface<T1, T2, T3, T4>  {      //Generic class with 5 type parameters implementing generic interface with 4 type parameters  }    class GenericClass2<T1, T2, T3> implements GenericInterface<T1, T2, T3, T4>  {      //Compile time error, must have same number of type parameters  }    class GenericClass3<T1, T2, T5, T6> implements GenericInterface<T1, T2, T3, T4>  {      //Compile time error. must have same type of parameters  } |

* Class can implement more than one generic interfaces. If implemented, class should have type parameters of both the interfaces.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | interface GenericInterface1<T1>  {      //Generic interface with one type parameter  }    interface GenericInterface2<T2, T3>  {      //Generic interface with two type parameters  }    class GenericClass<T1,T2, T3> implements GenericInterface1<T1>, GenericInterface2<T2, T3>  {      //Class having parameters of both the interfaces  } |

# 4. Can We Define Methods And Constructors As Generic?

Generics are very useful and flexible feature of Java. Generics provide safe type casting to your coding. Along with safe type casting, they also give flexibility to your coding. For example, Once you write a class or interface using generics, you can use any type to create objects to them. In simple words, You can make objects to work with any type using generics.

One more addition to generics is **Generic Methods**. If you don’t want whole class or interface to be generic, you want only some part of class as generic, then generic methods will be solution for this.

The syntax for defining generic methods is as follows,

|  |  |
| --- | --- |
| 1  2  3  4 | <type-Parameters> return\_type method\_name(parameter list)  {    } |

You can observe that type parameters are mentioned just before the return type. It is a rule you must follow while defining generic methods. The remaining parts are same as in normal method.

Generic methods can be static or non-static. There is no restriction for that. Generic class as well as non-generic class can have generic methods.

Here is an example which contains static generic method defined inside a non-generic class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class NonGenericClass  {      static <T> void genericMethod(T t1)      {          T t2 = t1;            System.out.println(t2);      }  } |

In this example, ‘genericMethod()’ is a static generic method with ‘T’ as type parameter. Notice that type parameter is mentioned just before the return type.

While calling above generic method, you can pass any type as an argument. This is the best example for generics providing the flexibility. Look at the below code, I have called the above method by passing three different types as an argument.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class GenericsInJava  {      public static void main(String[] args)      {          NonGenericClass.genericMethod(new Integer(123));     //Passing Integer type as an argument            NonGenericClass.genericMethod("I am string");        //Passing String type as an argument            NonGenericClass.genericMethod(new Double(25.89));    //Passing Double type as an argument      }  } |

## **Constructors As Generics :**

As we all know that constructors are like methods but without return types. Like methods, constructors also can be generic. Even non-generic class can have generic constructors. Here is an example in which constructor of a non-generic class is defined as generic.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class NonGenericClass  {      public <T> NonGenericClass(T t1)      {          T t2 = t1;            System.out.println(t2);      }  }    public class GenericsInJava  {      public static void main(String[] args)      {          //Creating object by passing Integer as an argument            NonGenericClass nonGen1 = new NonGenericClass(123);            //Creating object by passing String as an argument            NonGenericClass nonGen2 = new NonGenericClass("abc");            //Creating object by passing Double as an argument            NonGenericClass nonGen3 = new NonGenericClass(25.69);      }  } |

# 5. What Are Bounded Types And Why They Are Used?

In the earlier posts, We have seen that while creating objects to generic classes we can pass any derived type as type parameters. Many times it will be useful to limit the types that can be passed to type parameters. For that purpose, **bounded types** or **bounded type parameters** are introduced in generics. Using bounded types, you can make the objects of generic class to have data of specific derived types.

For example, If you want a generic class that works only with numbers (like int, double, float, long …..) then declare type parameter of that class as a bounded type to java.lang.Number class. Then while creating objects to that class you have to pass only Number types or it’s subclass types as type parameters.

Here is the syntax for declaring Bounded type parameters.

**<T extends SuperClass>**

This specifies that ‘T’ can only be replaced by ‘SuperClass’ or it’s sub classes. Remember that **extends** clause is an inclusive bound. That means bound includes ‘SuperClass’ also.

Here is an example which demonstrates the bounded type parameters.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class GenericClass<T extends Number>    //Declaring Number class as upper bound of T  {      T t;        public GenericClass(T t)      {          this.t = t;      }        public T getT()      {          return t;      }  } |

In this example, T has been declared as bounded type to Number class. So while creating objects to this class, you have to pass either Number type or it’s subclass types (Integer, Double, Float, Byte… ) as a type parameter. It wouldn’t allow other than these types to pass as a type parameter. If you try to pass, compiler will throw compile time error.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public class GenericsInJava  {      public static void main(String[] args)      {          //Creating object by passing Number as a type parameter            GenericClass<Number> gen1 = new GenericClass<Number>(123);            //Creating object by passing Integer as a type parameter            GenericClass<Integer> gen2 = new GenericClass<Integer>(new Integer(456));            //Creating object by passing Double as a type parameter            GenericClass<Double> gen3 = new GenericClass<Double>(new Double(23.589));            //Creating object by passing Long as a type parameter            GenericClass<Long> gen4 = new GenericClass<Long>(new Long(12));            //While Creating object by passing String as a type parameter, it gives compile time error            GenericClass<String> gen5 = new GenericClass<String>("I am string");   //Compile time error      }  } |

## **Bounded Type Parameters In Generic Methods :**

You can use bounded types while defining generic methods also. Here is an example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | class GenericClass  {      //Declaring T as bounded type to Number class        public static <T extends Number> void printNumbers(T[] t)      {          for (int i = 0; i < t.length; i++)          {              System.out.println(t[i]);          }      }  }    public class GenericsInJava  {      public static void main(String[] args)      {          //Passing Integer[] array while calling printNumbers()            GenericClass.printNumbers(new Integer[] {new Integer(10), new Integer(20), new Integer(30), new Integer(40)} );            //Passing Double[] array while calling printNumbers()            GenericClass.printNumbers(new Double[] {new Double(21.45), new Double(20.45), new Double(34.87), new Double(48.36)} );            //Passing String[] array while calling printNumbers(), it gives compile time error            GenericClass.printNumbers(new String[] {"one", "Two", "Three", "Four"});    //Compile time error      }  } |

## **Using Interface As An Upper Bound :**

You can also use interface type along with class type as an upper bound to type parameters. As in java, any class can extend only one class and can implement multiple interfaces, this also applies while declaring the bound to type parameters. That means a bounded parameter can extend only one class and one or more interfaces. While specifying bounded parameters that has a class and an interface or multiple interfaces, use **&** operator as a delimiter.

|  |  |
| --- | --- |
| 1  2  3  4 | class GenericClass <T extends AnyClass & FirstInterface & SecondInterface>  {    } |

# 6. What Are Wildcard Arguments In Java?

**Wildcard arguments** means unknown type arguments. They just act as placeholder for real arguments to be passed while calling method. They are denoted by question mark (?). One important thing is that the types which are used to declare wildcard arguments must be generic types. Wildcard arguments are declared in three ways.

1) Wildcard Arguments With An Unknown Type

2) Wildcard Arguments with An Upper Bound

3) Wildcard Arguments with Lower Bound

## **1) Wildcard Arguments With An Unknown Type :**

The syntax for declaring this type of wildcard arguments is,

**GenericType<?>**

The arguments which are declared like this can hold any type of objects. For example, Collection<?> or ArrayList<?> can hold any type of objects like String, Integer, Double etc.

Look at the below code. The same processElements() method is used to process the ArrayList containing strings as well as integers.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | public class GenericsInJava  {      static void processElements(ArrayList<?> a)      {          for (Object element : a)          {              System.out.println(element);          }      }        public static void main(String[] args)      {          //ArrayList Containing Integers            ArrayList<Integer> a1 = new ArrayList<>();            a1.add(10);            a1.add(20);            a1.add(30);            processElements(a1);            //Arraylist containing strings            ArrayList<String> a2 = new ArrayList<>();            a2.add("One");            a2.add("Two");            a2.add("Three");            processElements(a2);      }  } |

## **2)Wildcard Arguments With An Upper Bound :**

In the above example, if you want the processElements() method to work with only numbers, then you can specify an upper bound for wildcard argument. To specify an upper bound for wildcards, use this syntax,

**GenericType<? extends SuperClass>**

This specifies that a wildcard argument can contain ‘SuperClass’ type or it’s sub classes. Remember that extends clause is an inclusive bound. i.e ‘SuperClass’ also lies in the bound.

The above processElements() method can be modified to process only numbers like below,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | public class GenericsInJava  {      static void processElements(ArrayList<? extends Number> a)      {          for (Object element : a)          {              System.out.println(element);          }      }        public static void main(String[] args)      {          //ArrayList Containing Integers            ArrayList<Integer> a1 = new ArrayList<>();            a1.add(10);            a1.add(20);            a1.add(30);            processElements(a1);            //Arraylist containing Doubles            ArrayList<Double> a2 = new ArrayList<>();            a2.add(21.35);            a2.add(56.47);            a2.add(78.12);            processElements(a2);            //Arraylist containing Strings            ArrayList<String> a3 = new ArrayList<>();            a3.add("One");            a3.add("Two");            a3.add("Three");            //This will not work            processElements(a3);     //Compile time error      }  } |

## **3) Wildcard Arguments With Lower Bound :**

You can also specify a lower bound for wildcard argument using **super** clause. Here is the syntax,

**GenericType<? super SubClass>**

This means that a wildcard argument can contain ‘SubClass’ type or it’s super classes.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | public class GenericsInJava  {      static void processElements(ArrayList<? super Integer> a)      {          for (Object element : a)          {              System.out.println(element);          }      }        public static void main(String[] args)      {          //ArrayList Containing Integers            ArrayList<Integer> a1 = new ArrayList<>();            a1.add(10);            a1.add(20);            a1.add(30);            processElements(a1);            //Arraylist containing Doubles            ArrayList<Double> a2 = new ArrayList<>();            a2.add(21.35);            a2.add(56.47);            a2.add(78.12);            //This will not work            processElements(a2);     //Compile time error      }  } |

**Note : ‘super’ clause is used to specify the lower bound for only wildcard arguments. It does not work with bounded types.**

# 7. Generics And Their Inheritance:

You have to follow some rules while making a generic class as a super class or a sub class. Some of those rules we have already discussed while implementing [generic interfaces](https://javaconceptoftheday.com/generic-interfaces-java/). This post is an extend of that post.

In this post, We will discuss some very interesting points about generic classes and their inheritance.

* A generic class can extend a non-generic class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class NonGenericClass  {       //Non Generic Class  }    class GenericClass<T> extends NonGenericClass  {      //Generic class extending non-generic class  } |

* Generic class can also extend another generic class. When generic class extends another generic class, sub class should have at least same type and same number of type parameters and at most can have any number and any type of parameters.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class GenericSuperClass<T>  {      //Generic super class with one type parameter  }    class GenericSubClass1<T> extends GenericSuperClass<T>  {      //sub class with same type parameter  }    class GenericSubClass2<T, V> extends GenericSuperClass<T>  {      //sub class with two type parameters  }    class GenericSubClass3<T1, T2> extends GenericSuperClass<T>  {      //Compile time error, sub class having different type of parameters  } |

* When generic class extends another generic class, the type parameters are passed from sub class to super class same as in the case of constructor chaining where super class constructor is called by sub class constructor by passing required arguments. For example, in the below program  ‘T’ in ‘GenericSuperClass’ will be replaced by String.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class GenericSuperClass<T>  {      T t;        public GenericSuperClass(T t)      {          this.t = t;      }  }    class GenericSubClass<T> extends GenericSuperClass<T>  {      public GenericSubClass(T t)      {          super(t);      }  }    public class GenericsInJava  {      public static void main(String[] args)      {          GenericSubClass<String> gen = new GenericSubClass<String>("I am string");            System.out.println(gen.t);       //Output : I am string      }  } |

* A generic class can extend only one generic class and one or more generic interfaces. Then it’s type parameters should be union of type parameters of generic class and generic interface(s).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class GenericSuperClass<T1>  {      //Generic class with one type parameter  }    interface GenericInterface1<T1, T2>  {      //Generic interface with two type parameters  }    interface GenericInterface2<T2, T3>  {      //Generic interface with two type parameters  }    class GenericClass<T1,T2, T3> extends GenericSuperClass<T1> implements GenericInterface1<T1, T2>, GenericInterface2<T2, T3>  {      //Class having parameters of both the interfaces and super class  } |

* Non-generic class can’t extend generic class except of those generic classes which have already pre defined types as their type parameters.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class GenericSuperClass<T>  {      //Generic class with one type parameter  }    class NonGenericClass extends GenericSuperClass<T>  {      //Compile time error, non-generic class can't extend generic class  }    class A  {      //Pre defined class  }    class GenericSuperClass1<A>  {      //Generic class with pre defined type 'A' as type parameter  }    class NonGenericClass1 extends GenericSuperClass1<A>  {      //No compile time error, It is legal  } |

* Non-generic class can extend generic class by removing the type parameters. i.e as a **raw type**. But, it gives a warning.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class GenericClass<T>  {      T t;        public GenericClass(T t)      {          this.t = t;      }  }    class NonGenericClass extends GenericClass       //Warning  {      public NonGenericClass(String s)      {          super(s);           //Warning      }  }    public class GenericsInJava  {      public static void main(String[] args)      {          NonGenericClass nonGen = new NonGenericClass("I am String");            System.out.println(nonGen.t);    //Output : I am String      }  } |

* While extending a generic class having bounded type parameter, type parameter must be replaced by either upper bound or it’s sub classes.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class GenericSuperClass<T extends Number>  {      //Generic super class with bounded type parameter  }    class GenericSubClass1 extends GenericSuperClass<Number>  {      //type parameter replaced by upper bound  }    class GenericSubClass2 extends GenericSuperClass<Integer>  {      //type parameter replaced by sub class of upper bound  }    class GenericSubClass3 extends GenericSuperClass<T extends Number>  {      //Compile time error  } |

* Generic methods of super class can be overrided in the sub class like normal methods.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | class GenericClass  {      <T> void genericMethod(T t)      {          System.out.println(1);      }  }    class NonGenericClass extends GenericClass  {      @Override      <T> void genericMethod(T t)      {              System.out.println(2);      }  }    public class GenericsInJava  {      public static void main(String[] args)      {          new GenericClass().genericMethod("I am String");       //Output : 1            new NonGenericClass().genericMethod("I am String");    //Output : 2      }  } |

# 8. Type Erasure:

In the previous posts, we have seen how type safety can be achieved using generics. If you use generics in your code, you need not to perform explicit casting. Compiler performs strong type checking during compilation and hence removing the chances of occurring ClassCastException at run time.

One more interesting thing about generics is **type erasure**. When you compile your java code, compiler removes all generic information mentioned in your code. Compiler replaces all type parameters with their bounded type. The type parameters which don’t have bounds will be replaced with java.lang.Object class. That means all type parameters exist till compilation only. They are erased during compilation. They don’t exist at run time.

To understand how type erasure works, consider this example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class GenericClassOne<T>  {      T t;    //T will be replaced by java.lang.Object when compiled  }    class GenericClassTwo<T extends Number>  {      T t;    //T will be replaced by java.lang.Number when compiled  } |

When you compile above two classes, compiler replaces type parameter ‘T’ of GenericClassOne with java.lang.Object class as it is not bounded and type parameter ‘T’ of GenericClassTwo is replaced by java.lang.Number class as it is bounded by Number class. This is how above two classes look after compilation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class GenericClassOne extends java.lang.Object  {      java.lang.Object t;  }    class GenericClassTwo extends java.lang.Object  {      java.lang.Number t;  } |

You can notice that type parameters are erased after compilation. They don’t exist at run time. That’s why you can’t instantiate a type parameter. It gives compile time error.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class GenericClass<T>  {      T t = new T();      //Compile time error        <V> void genericMethod()      {          V v = new V();   //Compile time error      }  } |

# 9. Some Interesting Observations About Generics In Java:

In this post, I have tried to list down some interesting observations about generics in java. You may get questions about these points in the interview or any java certification exams.

* Java allows generic classes to use without type parameters i.e as a **raw type**. This is because to provide the compatibility of generic code with non-generic code. That means, non-generic code must be able to work with generic code and generic code must be able to work with non-generic code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class GenericClass<T>  {      //Generic class  }    public class GenericsInJava  {      public static void main(String[] args)      {          GenericClass rawType = new GenericClass();     //Using generic class as a raw type      }  } |

* You can’t create an instance to the type parameters. This is because, the type parameters does not exist at run time. They are erased during compilation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class GenericClass<T>  {      T t = new T();     //Compile Time error        <V> void genericMethod()      {          V v = new V();     //Compile Time error      }  } |

* In generic class with type parameter ‘T’, you can’t declare static fields of type ‘T’ and you can’t use ‘T’ in a static method. However, you can declare static generic methods with their own type parameters.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class GenericClass<T>  {      static T t;        //Compile time error        static void staticMethod()      {          System.out.println(t);    //Compile time error      }        static <V> void genericMethod()      {          //Static generic method      }  } |

* You can’t instantiate an array whose type is a type parameter.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class GenericClass<T>  {      T[] t;        public GenericClass(T[] t)      {          t = new T[5];   //Compile time error            this.t = t;     //But, This is OK      }  } |

* You can’t create an array of generic type containing specific type of data. But, you can create an array of generic type containing unknown type of data.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class GenericClass<T>  {          //Generic type  }    public class GenericsInJava  {      public static void main(String[] args)      {          GenericClass<Number> gen[] = new GenericClass<Number>[10];   //Compile time error            GenericClass<?> gen1[] = new GenericClass<?>[10];    //But, this is fine      }  } |

* You can not create generic exceptions i.e A generic class can not extend Throwable or any of it’s sub classes.

|  |  |
| --- | --- |
| 1  2  3  4 | class GenericClass<T> extends Throwable  {      //Compile time error  } |

**Collection Framework**

# 1. Collection Framework – Class Hierarchy:

## **Why Collection Framework?**

**Collections** are nothing but group of objects stored in well defined manner. Earlier, Arrays are used to represent these group of objects. But, arrays are not re-sizable. size of the arrays are fixed. Size of the arrays can not be changed once they are defined. This causes lots of problem while handling group of objects. To overcome this drawback of arrays, **Collection framework** or simply collections are introduced in java from JDK 1.2.

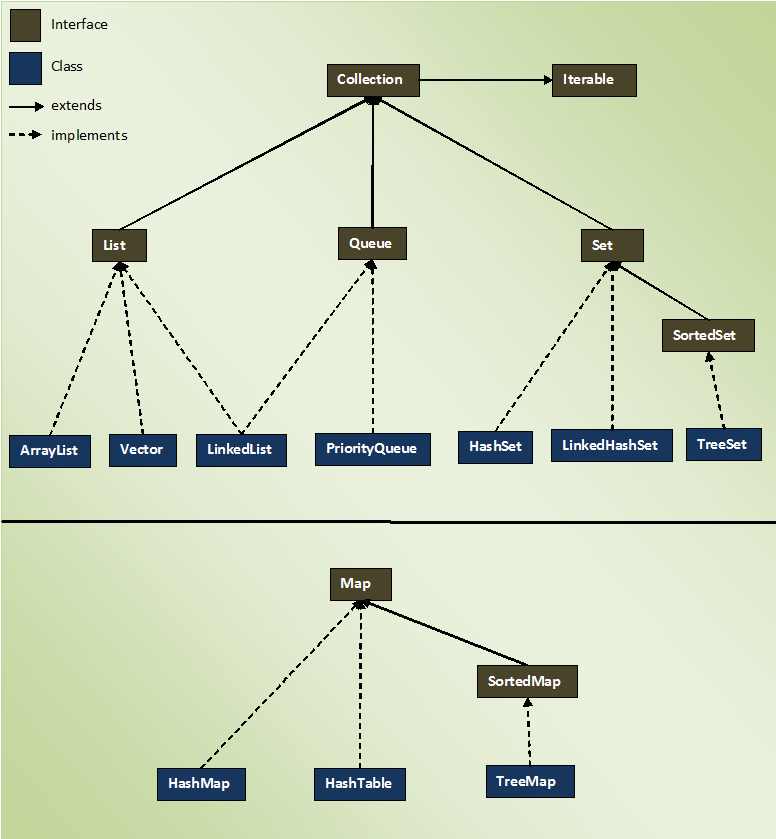
Although, there were classes like **Dictionary**, **Vector**, **Stack** and **Properties** which handle group of objects better than the arrays. But, each of them handle the objects differently. The way you use **Dictionary** class is totally different from the way you use **Stack** class and the way you use **Vector** class is different from the way you use **Properties class**. Hence, there needed a central and unifying theme to handle the group of objects. The collection framework is the answer to that.

## **What is Collection Framework In Java?**

Collection Framework in java is a centralized and unified theme to store and manipulate the group of objects. Java Collection Framework provides some pre-defined classes and interfaces to handle the group of objects. Using collection framework, you can store the objects as a **list** or as a **set** or as a **queue** or as a **map** and perform operations like adding an object or removing an object or sorting the objects without much hard work.

## **Class Hierarchy Of Collection Framework :**

All classes and interfaces related to Collection Framework are placed in **java.util** package. **java.util.Collection** interface is at the top of class hierarchy of Collection Framework. Below diagram shows the class hierarchy of collection framework.



The entire collection framework is divided into four interfaces.

**1) List**  —> It handles sequential list of objects. **ArrayList**, **Vector** and **LinkedList** classes implement this interface.

**2) Queue**  —> It handles the special group of objects in which elements are removed only from the head. **LinkedList** and **PriorityQueue** classes implement this interface.

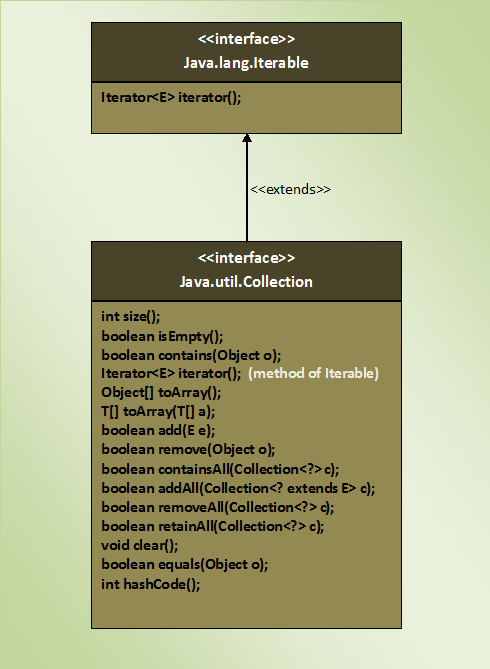
**3) Set**  —> It handles the group of objects which must contain only unique elements. This interface is implemented by **HashSet** and **LinkedHashSet** classes and extended by **SortedSet** interface which in turn, is implemented by **TreeSet**.

**4) Map**  —> This is the one interface in Collection Framework which is not inherited from Collection interface. It handles the group of objects as Key/Value pairs. It is implemented by **HashMap** and **HashTable** classes and extended by **SortedMap** interface which in turn is implemented by **TreeMap**.

Three of above interfaces (List, Queue and Set) inherit from Collection interface. Although, Map is included in collection framework it does not inherit from Collection interface.

# 2. Collection Framework – Collection Interface:

**Collection interface** is the root level interface in the collection framework. List, Queue and Set are all sub interfaces of Collection interface. JDK does not provide any direct implementations of this interface. But, JDK provides direct implementations of it’s sub interfaces.

Collection interface extends **Iterable interface** which is a member of java.lang package. Iterable interface has only one method called iterator(). It returns an Iterator object, using that object you can iterate over the elements of Collection. Here is the class diagram of Collection interface.  


Collection interface contains total 15 abstract methods. 14 of it’s own and one is inherited from Iterable interface. Here is the list and descriptions of those methods.

|  |  |  |
| --- | --- | --- |
| **SL No.** | **Method** | **Description** |
| **1** | int size() | Returns the number of elements in this collection |
| **2** | boolean isEmpty() | Checks whether this collection is empty or not. If collection is empty, it returns true otherwise it returns false |
| **3** | boolean contains(Object o) | Checks whether this collection has specified element. |
| **4** | Iterator<E> iterator() | Returns an iterator over the collection. |
| **5** | Object[] toArray() | It returns an array containing all elements of this collection. |
| **6** | <T> T[] toArray(T[] a) | It returns an array of specified type containing all elements of this collection. |
| **7** | boolean add(E e) | This method adds specified element to this collection. It returns true if element is added successfully to the collection otherwise it returns false. |
| **8** | boolean remove(Object o) | Removes the specified element from this collection. |
| **9** | boolean containsAll(Collection<?> c) | It checks whether this collection contains all elements of passed collection. |
| **10** | boolean addAll(Collection<? extends E> c) | Adds all elements of the passed collection to this collection. |
| **11** | boolean removeAll(Collection<?> c) | Removes all elements of this collection which are also elements of passed collection. |
| **12** | boolean retainAll(Collection<?> c) | Retains only those elements in this collection which are also elements of passed collection. |
| **13** | void clear() | Removes all elements in this collection. |
| **14** | boolean equals(Object o) | Compares the specified object with this collection for equality. |
| **15** | int hashCode() | Returns the hash code value of this collection. |

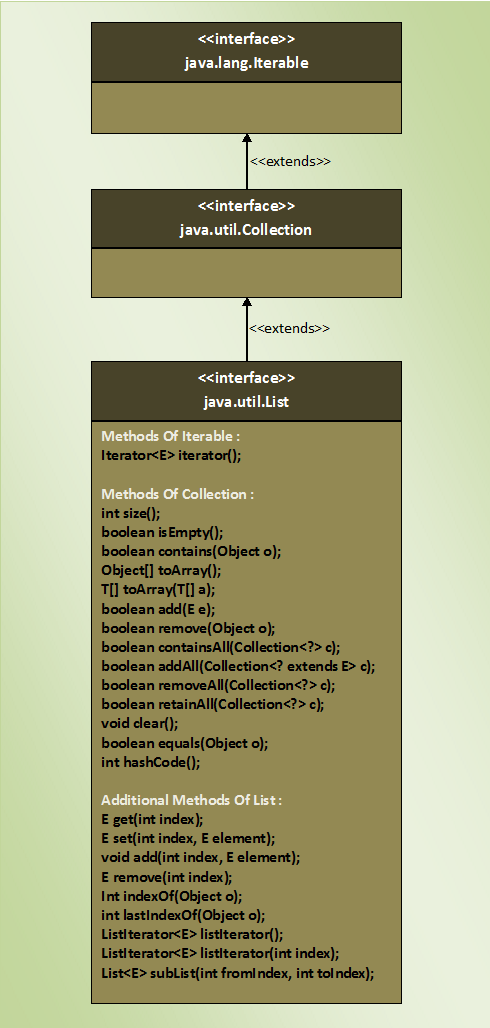
**Note :**  
*equals()* and *hashcode()* methods in the *Collection* interface are not the methods of *java.lang.Object* class. Because, interfaces does not inherit from *Object* class. Only classes in java are inherited from *Object* class. Any classes implementing *Collection* interface must provide their own version of *equals()* and *hashcode()* methods or they can retain default version inherited from *Object* class.

# 3. Collection Framework – List Interface:

**List** **Interface** represents an ordered or sequential collection of objects. This interface has some methods which can be used to store and manipulate the ordered collection of objects. The classes which implement the List interface are called as **Lists**. ArrayList, Vector and LinkedList are some examples of lists. You have the control over where to insert an element and from where to remove an element in the list.

Here are some properties of lists.

* Elements of the lists are ordered using Zero based index.
* You can access the elements of lists using an integer index.
* Elements can be inserted at a specific position using integer index. Any pre-existing elements at or beyond that position are shifted right.
* Elements can be removed from a specific position. The elements beyond that position are shifted left.
* A list may contain duplicate elements.
* A list may contain multiple null elements.

List interface extends Collection interface. So, All 15 methods of Collection interface are inherited to List interface. Along with these methods, another 9 methods are included in the List interface to support the properties of lists. Here is the class diagram of List interface.  


## **Additional Methods Of List Interface :**

There are 9 additional methods included in List interface along with the methods inherited from Collection Interface. Here is the list and the details of those methods.

|  |  |  |
| --- | --- | --- |
| SL NO | Methods | Descriptions |
| 1 | E get(int index) | Returns element at the specified position. |
| 2 | E set(int index, E element) | Replaces an element at the specified position with the passed element. |
| 3 | void add(int index, E element) | Inserts passed element at a specified index. |
| 4 | E remove(int index) | Removes an element at specified index. |
| 5 | int indexOf(Object o) | It returns an index of first occurrence of passed object. |
| 6 | int lastIndexOf(Object o) | It returns an index of last occurrence of passed object. |
| 7 | ListIterator<E> listIterator() | It returns a list iterator over the elements of this list. |
| 8 | ListIterator<E> listIterator(int index) | Returns a list iterator over the elements of this list starting from the specified index. |
| 9 | List<E> subList(int fromIndex, int toIndex) |  |

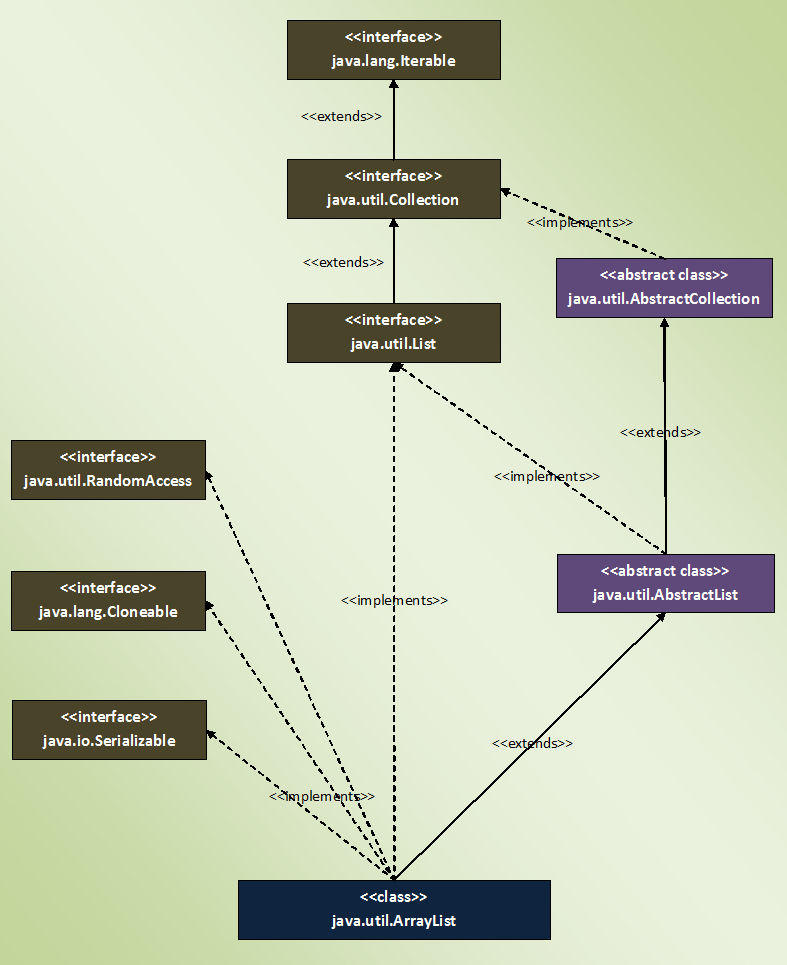
# 3.1. Collection Framework – The ArrayList Class:

In java, normal arrays are of fixed length. You can not change the size of arrays once they are defined. That means, you must know in advance how large an array you want. But sometimes, you may not know how large an array you want. To overcome this situation, ArrayList is introduced in Collection framework.

**ArrayList**, in simple terms, can be defined as re-sizable array. ArrayList is same like normal array but it can grow and shrink dynamically to hold any number of elements. ArrayList is a sequential collection of objects which increases or decreases in size as we add or delete the elements.

In ArrayList, elements are positioned according to **Zero-based index**. That means, elements are inserted from index 0. **Default initial capacity** of an ArrayList is 10. This capacity increases automatically as we add more elements to arraylist. You can also specify initial capacity of an ArrayList while creating it.

ArrayList class implements **List interface** and extends **AbstractList**. It also implements 3 marker interfaces – **RandomAccess**, **Cloneable** and **Serializable**. Here is hierarchy diagram of ArrayList class.



## **Properties Of ArrayList :**

* Size of the ArrayList is not fixed. It can increase and decrease dynamically as we add or delete the elements.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public class MainClass  {      public static void main(String[] args)      {          //ArrayList with no size defined            ArrayList<Integer> list = new ArrayList<>();            //Adding elements to ArrayList            list.add(10);            list.add(20);            list.add(30);            list.add(40);            System.out.println(list.size());     //Output : 4            //Removing an element at index 0            list.remove(0);            System.out.println(list.size());    //Output : 3      }  } |

* ArrayList can have any number of null elements.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<Integer> list = new ArrayList<>();            //Adding elements to ArrayList            list.add(100);            list.add(null);            list.add(2000);            list.add(null);            list.add(null);            //ArrayList having 3 null elements            System.out.println(list);     //Output : [100, null, 2000, null, null]      }  } |

* ArrayList can have duplicate elements.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<Integer> list = new ArrayList<>();            //Adding elements to ArrayList            list.add(100);            list.add(100);            list.add(100);            list.add(100);            //ArrayList having 4 duplicate elements            System.out.println(list);     //Output : [100, 100, 100, 100]      }  } |

* As ArrayList implements RandomAccess, you can get, set, insert and remove elements of the ArrayList from  any arbitrary position.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<Integer> list = new ArrayList<>();            //Adding elements to ArrayList            list.add(10);            list.add(20);            list.add(30);            list.add(40);            System.out.println(list);     //Output : [10, 20, 30, 40]            //Retrieving element at index 2            System.out.println(list.get(2));     //Output : 30            //Setting value of element at index 2            list.set(2, 2222);            System.out.println(list);      //Output : [10, 20, 2222, 40]            //Inserting element at index 1            list.add(1, 1111);            System.out.println(list);     //Output : [10, 1111, 20, 2222, 40]            //Removing element from index 3            list.remove(3);            System.out.println(list);    //Output : [10, 1111, 20, 40]      }  } |

* When you insert an element in the middle of the ArrayList, the elements at the right side of that position are shifted one position right and when you delete an element, they will be shifted one position left. This feature of the ArrayList causes some performance issues as shifting of elements is time consuming if ArrayList has lots of elements.
* Elements are placed according to **Zero-based index**. That means, first element will be placed at index 0 and last element at index n-1, where ‘n’ is the size of the ArrayList.

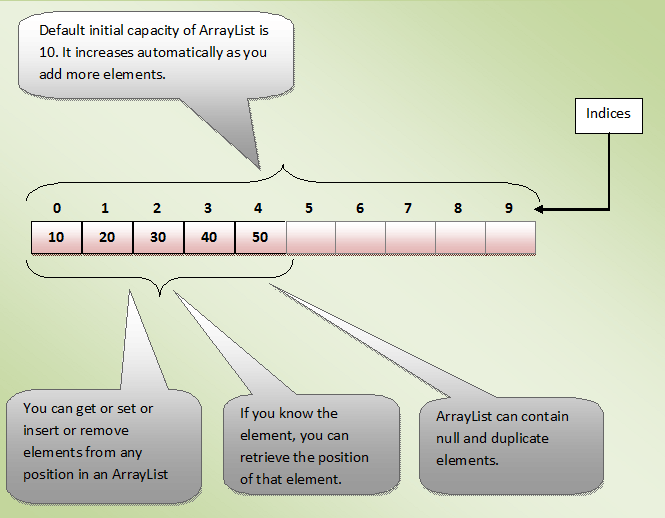
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            list.add("First");            list.add("Second");            list.add("Third");            list.add("Fourth");            System.out.println(list);    //Output : [First, Second, Third, Fourth]      }  } |

* ArrayList is not synchronized. That means, multiple threads can use same ArrayList simultaneously.
* If you know the element, you can retrieve the position of that element.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            //Adding elements to ArrayList            list.add("First");            list.add("Second");            list.add("Third");            list.add("Fourth");            System.out.println(list);    //Output : [First, Second, Third, Fourth]            //Retrieving position of "Second" element            System.out.println(list.indexOf("Second"));     //Output : 1      }  } |

## **Quick Overview Of ArrayList  Class:**

Below diagram shows quick overview of ArrayList.



# 3.2. Differences Between Array Vs ArrayList In Java:

Array and ArrayList are two important and most used data structures in Java. Array is a basic data structure which is the part of Java from the beginning. ArrayList is a class in Java Collection Framework which is introduced from JDK 1.2. You can describe ArrayList as an advanced version of Array. Because, array is a fixed length data structure. You can’t change its size once it is created. To overcome this drawback of array, ArrayList is introduced in Java. ArrayList automatically resizes itself when you add elements more than its capacity. Let’s discuss the differences between Array Vs ArrayList in Java in detail.

### **Differences Between Array And ArrayList In Java :**

**1) Static Vs Dynamic**

Array is static in nature i.e its length is fixed. You can’t change its size once it is created. Where as ArrayList is dynamic in nature. ArrayList is also called as dynamic array or re-sizable array. Because, it automatically resizes itself if you try to add elements beyond its capacity.

**2) How they are implemented?**

ArrayList internally uses an array to store its elements. So, internal implementation of both array and ArrayList is almost the same. The only difference is that when you try to add elements to ArrayList beyond its capacity, it creates the new array with increased size and copies the elements from old array to new array.

**3) How they Perform?**

Both, array and ArrayList, give constant time performance for both add and get operations. But, in case of ArrayList, if adding an element requires resizing of an ArrayList, then it gets slightly slower as it involves creating a new array in the background and copying all elements from old array to new array.

**4) What they can hold?**

Array can hold both primitive data types (int, float….) as well as objects. Where as ArrayList can hold only objects. If you try to insert primitive data into ArrayList, data is automatically boxed into corresponding wrapper classes.

**5) How they can be iterated?**

ArrayList provides iterators to iterate through its elements. You can also use for loop or for-each loop to iterate an ArrayList. But to iterate an array, you have to use either for loop or for-each loop.

**6) How you can check their size?**

The size of an array can be checked using its attribute called length. The ArrayList provides method called size() to check its size.

**7) Type Safety : Compile Time Type Checking Vs Run Time Type Checking**

As ArrayList supports generics, it ensures the type safety during compilation itself. i.e the type of each element is checked at compile time. If you try to add an incompatible element, compiler will show an error. But, array does not support generics. If you add an incompatible element into an array, compiler doesn’t show any error but you will get ArrayStoreException at run time. That means it checks the type of each element at run time.

**8) Are they multi-dimensional?**

Array is multi-dimensional. You can have one, two or three dimensional arrays. But, ArrayList is one dimensional.

**9) How do you add elements into them?**

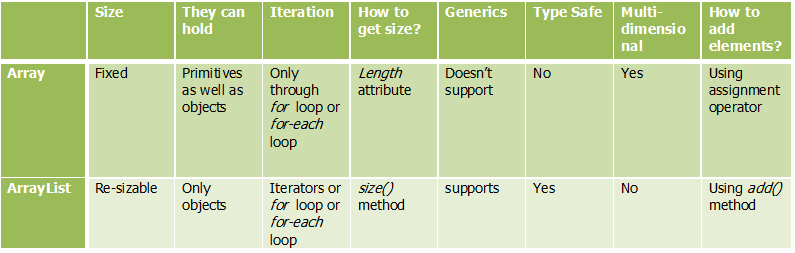
The elements are inserted into ArrayList using add() method. To insert elements into an array, we use assignment operator.

**10) How do you manipulate their elements?**

ArrayList provides methods like get(), isEmpty(), contains(), indexOf(), replaceAll()…… to manipulate its elements. Where as array doesn’t provide such methods.

### **Array Vs ArrayList In Java :**

|  |  |
| --- | --- |
| **Array** | **ArrayList** |
| Arrays are static in nature. Arrays are fixed length data structures. You can’t change their size once they are created. | ArrayList is dynamic in nature. Its size is automatically increased if you add elements beyond its capacity. |
| Arrays can hold both primitives as well as objects. | ArrayList can hold only objects. |
| Arrays can be iterated only through for loop or for-each loop. | ArrayList provides iterators to iterate through their elements. |
| The size of an array is checked using length attribute. | The size of an ArrayList can be checked using size() method. |
| Array gives constant time performance for both add and get operations. | ArrayList also gives constant time performance for both add and get operations provided adding an element doesn’t trigger resize. |
| Arrays don’t support generics. | ArrayList supports generics. |
| Arrays are not type safe. | ArrayList are type safe. |
| Arrays can be multi-dimensional. | ArrayList can’t be multi-dimensional. |
| Elements are added using assignment operator. | Elements are added using add() method. |



# 3.3 Advantages Of Using ArrayList Over Arrays:

Array and ArrayList are most used data types while developing any java applications. Both are used to store group of objects. In this post I have tried to list down the advantages of using **ArrayList** over **Arrays**. Before discussing the advantages of ArrayList, let’s see what are the drawbacks of arrays.

* Arrays are of fixed length. You can not change the size of the arrays once they are created.
* You can not accommodate an extra element in an array after they are created.
* Memory is allocated to an array during it’s creation only, much before the actual elements are added to it.

Because of these drawbacks, use of arrays are less preferred. Instead of arrays, you can use ArrayList class which addresses all these drawbacks. Here are some advantages of using ArrayList over arrays.

1) You can define ArrayList as **re-sizable array**. Size of the ArrayList is not fixed. ArrayList can grow and shrink dynamically.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class ArrayListDemo  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            list.add("ONE");            list.add("TWO");            list.add("THREE");            System.out.println(list.size());     //Output : 3            //Inserting some more elements          list.add("FOUR");            list.add("FIVE");            System.out.println(list.size());    //Output : 5            //Removing an element          list.remove("TWO");            System.out.println(list.size());    //Output : 4      }  } |

2) Elements can be inserted at or deleted from a particular position.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | class ArrayListDemo  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            list.add("ZERO");            list.add("TWO");            list.add("FOUR");            System.out.println(list);     //Output : [ZERO, TWO, FOUR]            list.add(2, "THREE");       //Inserting an element at index 2            list.add(1, "ONE");     //Inserting an element at index 1            System.out.println(list);    //Output : [ZERO, ONE, TWO, THREE, FOUR]            list.remove(3);       //Removing an element from index 3            System.out.println(list);    //Output : [ZERO, ONE, TWO, FOUR]      }  } |

3) ArrayList class has many methods to manipulate the stored objects.

ArrayList class has methods to perform solo modifications ( add(), remove()… ), bulk modifications ( addAll(), removeAll(), retainAll()… ), searching( indexOf(), lasIndexOf() ) and iterations( iterator() ).

4) If generics are not used, ArrayList can hold any type of objects.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class ArrayListDemo  {      public static void main(String[] args)      {          ArrayList list = new ArrayList();     //ArrayList without generics            list.add("ZERO");    //adding string type object            list.add(1);        //adding primitive int type            list.add(20.24);    //adding primitive double type            list.add(new Float(23.56));   //Adding Float wrapper type object            list.add(new Long(25));      //Adding Long wrapper type object            System.out.println(list);     //Output : [ZERO, 1, 20.24, 23.56, 25]      }  } |

5) Many are of the assumption that multiple insertion and removal operations on ArrayList will decrease the performance of an application. But, there will be no significant change in the performance of an application if you use ArrayList instead of arrays. Below example shows time taken to add 1000 string elements to ArrayList and array.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class ArrayListDemo  {      public static void main(String[] args)      {          String[] namesArray = new String[1000];            long startTime = System.currentTimeMillis();            for (int i = 0; i < namesArray.length; i++)          {              namesArray[i] = "Name"+i;          }            long endTime = System.currentTimeMillis();            System.out.println("Time taken by Array : "+(endTime - startTime)+"ms");            ArrayList<String> nameList = new ArrayList<String>();            startTime = System.currentTimeMillis();            for (int i = 0; i <= 1000; i++)          {              nameList.add("Name"+i);          }            endTime = System.currentTimeMillis();            System.out.println("Time taken by ArrayList : "+(endTime-startTime)+"ms");      }  } |

Output :  
Time taken by Array : 6ms  
Time taken by ArrayList : 6ms

6) You can traverse an ArrayList in both the directions – forward and backward using ListIterator.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class ArrayListDemo  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            list.add("ONE");            list.add("TWO");            list.add("THREE");            list.add("FOUR");            ListIterator iterator = list.listIterator();            System.out.println("Elements in forward direction");            while (iterator.hasNext())          {              System.out.println(iterator.next());          }            System.out.println("Elements in backward direction");            while (iterator.hasPrevious())          {              System.out.println(iterator.previous());          }      }  } |

7) ArrayList can hold multiple null elements.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class ArrayListDemo  {      public static void main(String[] args)      {          ArrayList<Integer> list = new ArrayList<Integer>();            list.add(100);            list.add(null);            list.add(null);            System.out.println(list);     //Output : [100, null, null]      }  } |

8) ArrayList can hold duplicate elements.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class ArrayListDemo  {      public static void main(String[] args)      {          ArrayList<Integer> list = new ArrayList<Integer>();            list.add(100);            list.add(100);            list.add(100);            System.out.println(list);     //Output : [100, 100, 100]      }  } |

(Above two advantages(7 and 8) are also applicable to arrays. But, you can treat them as bonus with all above advantages of ArrayList.)

# 3.4 18 Java ArrayList Programming Examples:

## **18 Java ArrayList Programming Examples**

**1) Explain the different ways of constructing an ArrayList?**

ArrayList can be created in 3 ways.

a) **ArrayList()** —> It creates an empty ArrayList with initial capacity of 10.

b) **ArrayList(int initialCapacity)** —> It creates an empty ArrayList with supplied initial capacity.

c) **ArrayList(Collection c)** —> It creates an ArrayList containing the elements of the supplied collection.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<Integer> list1 = new ArrayList<Integer>();            //First Method            ArrayList<String> list2 = new ArrayList<String>(20);         //Second Method            ArrayList<Integer> list3 = new ArrayList<Integer>(list1);      //Third Method      }  } |

**2) How do you increase the current capacity of an ArrayList?**

**ensureCapacity() method** is used to increase the current capacity of an ArrayList. However, capacity of an ArrayList is automatically increased when we try to add more elements than the current capacity. To manually increase the current capacity, ensureCapacity() method is used.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            //Now 'list' can hold 10 elements (Default Initial Capacity)            list.ensureCapacity(20);            //Now 'list' can hold 20 elements.      }  } |

**3) How do you decrease the current capacity of an ArrayList to the current size?**

**trimToSize() method** is used to trim the capacity of arrayList to the current size of ArrayList. Developers use this method to minimize the storage area of an ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            //Now 'list' can hold 10 elements (Default Initial Capacity)            list.ensureCapacity(20);            //Now 'list' can hold 20 elements.            list.add("ONE");            list.add("TWO");            list.add("THREE");            list.add("FOUR");            //reducing the current capacity to current size of an ArrayList.            list.trimToSize();      }  } |

**4) How do you find the number of elements present in an ArrayList?**

Using **size()** method. size() method returns number of elements present in an ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<Double> list = new ArrayList<Double>();            list.add(1.1);            list.add(2.2);            list.add(3.3);            list.add(4.4);            list.add(5.5);            System.out.println(list);     //Output : [1.1, 2.2, 3.3, 4.4, 5.5]            System.out.println("Size Of ArrayList = "+list.size());   //Output : Size Of ArrayList = 5      }  } |

**5) How do you find out whether the given ArrayList is empty or not?**

**isEmpty()** method of ArrayList is used to check whether the given ArrayList is empty or not. This method returns true if an ArrayList contains no elements otherwise returns false.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<Double> list = new ArrayList<Double>();            System.out.println(list.isEmpty());    //Output : true      }  } |

Note : You can also use size() method to check whether the given ArrayList is empty or not. size() method returns ‘0’ if an ArrayList is empty.

**6) How do you check whether the given element is present in an ArrayList or not?**

Using **contains()** method of ArrayList, we can examine whether the ArrayList contains the given element or not. This method returns true if ArrayList has that element otherwise returns false.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<Double> list = new ArrayList<Double>();            list.add(1.1);            list.add(11.11);            list.add(111.111);            list.add(1111.1111);            //Checking whether list conatins '111.1111'            System.out.println(list.contains(111.1111));    //Output : false      }  } |

**7) How do you get the position of a particular element in an ArrayList?**

We can use **indexOf()** and **lastIndexOf()** methods to find out the position of a given element in an ArrayList.**indexOf()** method returns index of first occurrence of a specified element where as **lastIndexOf()** method returns index of last occurrence of a specified element in an ArrayList. If element is not found, they will return -1.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            list.add("JAVA");            list.add("J2EE");            list.add("JSP");            list.add("JAVA");            list.add("SERVLETS");            list.add("JAVA");            list.add("STRUTS");            System.out.println(list);     //Output : [JAVA, J2EE, JSP, JAVA, SERVLETS, JAVA, STRUTS]            //Getting the index of first occurrence of "JAVA"            System.out.println(list.indexOf("JAVA"));     //Output : 0            //Getting the index of last occurrence of "JAVA"            System.out.println(list.lastIndexOf("JAVA"));    //Output : 5      }  } |

**8) How do you convert an ArrayList to Array?**

Using **toArray()** method of ArrayList class. toArray() method returns an array containing all elements of the ArrayList. This method acts as a bridge between normal arrays and collection framework in java.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            list.add("JAVA");            list.add("J2EE");            list.add("JSP");            list.add("SERVLETS");            list.add("STRUTS");            System.out.println(list);      //Output : [JAVA, J2EE, JSP, SERVLETS, STRUTS]            //getting an array containing all elements of the list.            Object[] array = list.toArray();            //Printing the elements of the returned array.            for (Object object : array)          {              System.out.println(object);          }    //      Output :    //      JAVA  //      J2EE  //      JSP  //      SERVLETS  //      STRUTS      }  } |

**9) How do you retrieve an element from a particular position of an ArrayList?**

**get()** method returns an element from a specified position of an ArrayList. This method takes index of the element as an argument.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<Integer> list = new ArrayList<Integer>();            list.add(111);            list.add(222);            list.add(333);            list.add(444);            System.out.println(list);     //Output : [111, 222, 333, 444]            //Getting element at index 3            System.out.println(list.get(3));    //Output : 444            //Getting element at index 1            System.out.println(list.get(1));    //Output : 222      }  } |

**10) How do you replace a particular element in an ArrayList with the given element?**

**set()** method replaces a particular element in an Arraylist with the given element. This method takes two arguments. One is the index of the element to be replaced and another one is the element to be placed at that position.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<Integer> list = new ArrayList<Integer>();            list.add(111);            list.add(222);            list.add(333);            list.add(444);            System.out.println(list);     //Output : [111, 222, 333, 444]            //Replacing the element at index 1 with '000'            list.set(1, 000);            //Replacing the element at index 3 with '000'            list.set(3, 000);            System.out.println(list);   //Output : [111, 0, 333, 0]      }  } |

**11) How do you append an element at the end of an ArrayList?**

**add()** method appends an element at the end of an ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            list.add("ONE");            list.add("TWO");            list.add("THREE");            list.add("FOUR");            System.out.println(list);     //Output : [ONE, TWO, THREE, FOUR]      }  } |

**12) How do you insert an element at a particular position of an ArrayList?**

**add()** method which takes index and an element as arguments can be used to insert an element at a particular position of an ArrayList. The elements at the right side of that position are shifted one position right i.e indices of right side elements of that position are increased by 1.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            list.add("ONE");            list.add("TWO");            list.add("THREE");            list.add("FOUR");            System.out.println(list);     //Output : [ONE, TWO, THREE, FOUR]            //Inserting "AAA" at index 1            list.add(1, "AAA");            //Inserting "BBB" at index 3            list.add(3, "BBB");            System.out.println(list);    //Output : [ONE, AAA, TWO, BBB, THREE, FOUR]      }  } |

**13) How do you remove an element from a particular position of an ArrayList?**

remove() method which takes int type as an argument is used to remove an element from a particular position of an ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            list.add("AAA");            list.add("BBB");            list.add("ccc");            list.add("DDD");            list.add("e");            System.out.println(list);     //Output : [AAA, BBB, ccc, DDD, e]            //Removing an element from position 2            list.remove(2);            System.out.println(list);    //Output : [AAA, BBB, DDD, e]            //Removing an element from position 3            list.remove(3);            System.out.println(list);   //Output : [AAA, BBB, DDD]      }  } |

**14) How do you remove the given element from an ArrayList?**

**remove(Object obj)** method removes the first occurrence of the specified element ‘**obj**‘. If that element doesn’t exist, ArrayList will be unchanged.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            list.add("AAA");            list.add("BBB");            list.add("AAA");            list.add("CCC");            list.add("BBB");            System.out.println(list);     //Output : [AAA, BBB, AAA, CCC, BBB]            //Removing first occurrence of "AAA"            list.remove("AAA");            System.out.println(list);    //Output : [BBB, AAA, CCC, BBB]            //Removing first occurrence of "BBB"            list.remove("BBB");            System.out.println(list);   //Output : [AAA, CCC, BBB]      }  } |

**15) How do you remove all elements of an ArrayList at a time?**

**clear()** method removes all elements of an ArrayList. ArrayList will be empty after this method is executed.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            list.add("AAA");            list.add("BBB");            list.add("AAA");            list.add("CCC");            list.add("BBB");            System.out.println(list);     //Output : [AAA, BBB, AAA, CCC, BBB]            //Removing all elements of the list            list.clear();            System.out.println(list);    //Output : []      }  } |

**16) How do you retrieve a portion of an ArrayList?**

Using **subList()** method of ArrayList, we can retrieve a portion of an ArrayList. subList() method returns a view of a portion of an ArrayList in the given range. The returned subList is backed by original ArrayList. That means any changes made to subList will be reflected in original ArrayList or Vice-Versa.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<Integer> list = new ArrayList<Integer>();            list.add(111);            list.add(222);            list.add(333);            list.add(444);            list.add(555);            list.add(666);            System.out.println(list);     //Output : [111, 222, 333, 444, 555, 666]            //Retrieving a SubList            List<Integer> subList = list.subList(1, 4);            System.out.println(subList);    //Output : [222, 333, 444]            //Modifying the list            list.set(2, 000);            //Changes will be reflected in subList            System.out.println(subList);    //Output : [222, 0, 444]            //Modifying the subList            subList.set(2, 000);            //Changes will be reflected in list            System.out.println(list);    //Output : [111, 222, 0, 0, 555, 666]      }  } |

**17) How do you join two ArrayLists?**

We can use addAll() method which takes Collection type as an argument to join two ArrayLists. This method appends all elements of the passed collection to the end of the invoking collection.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<Integer> list1 = new ArrayList<Integer>();            list1.add(111);            list1.add(222);            list1.add(333);            list1.add(444);            System.out.println(list1);     //Output : [111, 222, 333, 444]            ArrayList<Integer> list2 = new ArrayList<Integer>();            list2.add(555);            list2.add(666);            list2.add(777);            list2.add(888);            System.out.println(list2);    //Output : [555, 666, 777, 888]            //Joining list1 and list2            list1.addAll(list2);            System.out.println(list1);    //Output : [111, 222, 333, 444, 555, 666, 777, 888]      }  } |

**18) How do you insert more than one element at a particular position of an ArrayList?**

Another version of addAll() method which takes two arguments, one is index and another one is Collection type, can be used for this requirement. This method inserts all of the elements of passed collection at a specified position of an ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<Integer> list1 = new ArrayList<Integer>();            list1.add(111);            list1.add(222);            list1.add(333);            list1.add(444);            System.out.println(list1);     //Output : [111, 222, 333, 444]            ArrayList<Integer> list2 = new ArrayList<Integer>();            list2.add(555);            list2.add(666);            list2.add(777);            list2.add(888);            System.out.println(list2);    //Output : [555, 666, 777, 888]            //Inserting all elements of list2 at index 2 of list1            list1.addAll(2, list2);            System.out.println(list1);    //Output : [111, 222, 555, 666, 777, 888, 333, 444]      }  } |

# 3.5 Array To ArrayList And ArrayList To Array In Java With Examples:

Array and ArrayList are two most frequently used data types in java. Array is old derived type which can hold primitive types as well as objects where as ArrayList is a part of Java Collection Framework which holds only objects. One more difference is that array is of fixed size. You can’t change its size once it is created. But, ArrayList is re-sizable. It grows and shrinks as you insert or delete the elements. While coding, you often need to perform the conversion between these two i.e Array to ArrayList and ArrayList to Array. In this post, we will see how to convert Array to ArrayList and ArrayList to Array in java with some simple examples.

### **Array To ArrayList In Java :**

**a) Using Arrays.asList() Method :**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.util.ArrayList;  import java.util.Arrays;    public class ArrayToArrayListExample  {      public static void main(String[] args)      {          String[] array = new String[] {"ANDROID", "JSP", "JAVA", "STRUTS", "HADOOP", "JSF"};            ArrayList<String> list = new ArrayList<String>(Arrays.asList(array));            System.out.println(list);      }  } |

**Output :**

[ANDROID, JSP, JAVA, STRUTS, HADOOP, JSF]

**b) Using Collections.addAll() Method**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.util.ArrayList;  import java.util.Collections;    public class ArrayToArrayListExample  {      public static void main(String[] args)      {          String[] array = new String[] {"ANDROID", "JSP", "JAVA", "STRUTS", "HADOOP", "JSF"};            ArrayList<String> list = new ArrayList<String>();            Collections.addAll(list, array);            System.out.println(list);      }  } |

**Output :**

[ANDROID, JSP, JAVA, STRUTS, HADOOP, JSF]

**c) Using ArrayList.addAll() Method :**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.util.ArrayList;  import java.util.Arrays;    public class ArrayToArrayListExample  {      public static void main(String[] args)      {          String[] array = new String[] {"ANDROID", "JSP", "JAVA", "STRUTS", "HADOOP", "JSF"};            ArrayList<String> list = new ArrayList<String>();            list.addAll(Arrays.asList(array));            System.out.println(list);      }  } |

**Output :**

[ANDROID, JSP, JAVA, STRUTS, HADOOP, JSF]

**d) Using Streams from Java 8**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.util.Arrays;  import java.util.List;  import java.util.stream.Collectors;    public class ArrayToArrayListExample  {      public static void main(String[] args)      {          String[] array = new String[] {"ANDROID", "JSP", "JAVA", "STRUTS", "HADOOP", "JSF"};            List<Object> list = Arrays.stream(array).collect(Collectors.toList());            System.out.println(list);      }  } |

**Output :**

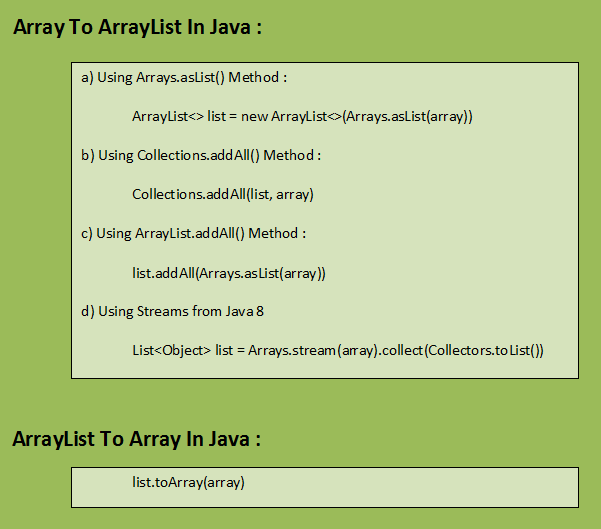
[ANDROID, JSP, JAVA, STRUTS, HADOOP, JSF]

### **ArrayList To Array In Java :**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | import java.util.ArrayList;    public class ArrayListToArrayExample  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            list.add("JAVA");            list.add("JSP");            list.add("ANDROID");            list.add("STRUTS");            list.add("HADOOP");            list.add("JSF");            String[] array = new String[list.size()];            list.toArray(array);            for (String string : array)          {              System.out.println(string);          }      }  } |

**Output :**

JAVA  
JSP  
ANDROID  
STRUTS  
HADOOP  
JSF



# 3.5 How To Reverse An ArrayList In Java?:

The given ArrayList can be reversed using Collections.reverse() method. Collections class is an utility class in java.util package which provides many useful methods to operate on Collection classes. Collections.reverse() method reverses the elements of the given ArrayList in linear time i.e it has the time complexity of O(n). Collections.reverse() method takes List type as an argument. So you can use this method to reverse any List type like ArrayList, LinkedList or Vector. Below is the program to reverse an ArrayList in Java.

**Also Read :**

[How to reverse an array in java?](https://javaconceptoftheday.com/reverse-an-array-in-java/)

[How to sort an arraylist in java?](https://javaconceptoftheday.com/how-to-sort-an-arraylist-in-java/)

[How to remove duplicate elements from an ArrayList?](https://javaconceptoftheday.com/how-to-remove-duplicate-elements-from-arraylist-in-java/)

### **Java Program To Reverse An ArrayList :**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | import java.util.ArrayList;  import java.util.Collections;    public class ReverseArrayListExample  {      public static void main(String[] args)      {          //Constructing an ArrayList            ArrayList<String> list = new ArrayList<String>();            list.add("Gold");            list.add("Iron");            list.add("Copper");            list.add("Silver");            list.add("Nickel");            list.add("Cobalt");            list.add("Zinc");            //Printing list before reverse            System.out.println("ArrayList Before Reverse :");            System.out.println(list);            //Reversing the list using Collections.reverse() method            Collections.reverse(list);            //Printing list after reverse            System.out.println("ArrayList After Reverse :");            System.out.println(list);      }  } |

**Output :**

ArrayList Before Reverse :  
[Gold, Iron, Copper, Silver, Nickel, Cobalt, Zinc]  
ArrayList After Reverse :  
[Zinc, Cobalt, Nickel, Silver, Copper, Iron, Gold]

### **Java Program To Reverse LinkedList :**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | import java.util.Collections;  import java.util.LinkedList;    public class ReverseLinkedListExample  {      public static void main(String[] args)      {          //Constructing a LinkedList            LinkedList<Integer> list = new LinkedList<Integer>();            list.add(56);            list.add(67);            list.add(81);            list.add(41);            list.add(63);            list.add(21);            list.add(96);            //Printing list before reverse            System.out.println("LinkedList Before Reverse :");            System.out.println(list);            //Reversing the list using Collections.reverse() method            Collections.reverse(list);            //Printing list after reverse            System.out.println("LinkedList After Reverse :");            System.out.println(list);      }  } |

**Output :**

LinkedList Before Reverse :  
[56, 67, 81, 41, 63, 21, 96]  
LinkedList After Reverse :  
[96, 21, 63, 41, 81, 67, 56]

### **Some Useful Methods Of java.Util.Collections :**

|  |  |
| --- | --- |
| Method | Description |
| Collections.copy() | This method is used to copy all elements from one list to another list. |
| Collections.frequency() | This method checks the number of occurrences of a specified element in the given Collection. |
| Collections.max() | It returns the maximum element in the given Collection. |
| Collections.min() | It returns the minimum element in the given Collection. |
| Collections.replaceAll() | It replaces all occurrences of old value with new value in the given list. |
| Collections.sort() | This method sorts the specified list in the ascending order. |
| Collections.synchronizedCollection() | This method returns the synchronized version of the specified Collection. |
| Collections.synchronizedList() | This method returns the synchronized i.e thread safe list backed by the specified list. |
| Collections.synchronizedMap() | This method returns the synchronized map backed by the specified map. |
| Collections.synchronizedSet() | This method returns the synchronized Set backed by the specified Set. |

# 3.5 How To Sort An ArrayList In Java With Examples?:

ArrayList is one of the widely used data structure in java. In ArrayList, elements are placed as they are inserted. But while coding, you often need them in some order. In this post, we will see how to sort an ArrayList with some simple examples.

### **How To Sort An ArrayList In Java?**

To sort an ArrayList, we use sort() method of Collections class. Collections class is an utility class in java.util package consisting of many useful methods. (Collections and Collection are two different entities. Check [this](https://javaconceptoftheday.com/difference-between-collection-and-collections-in-java/) for more info). Collections.sort() method has two overloaded forms. They are,

1) sort(List<T> list)  :  This method sorts the specified list according to natural ordering of its elements.

2) sort(List<T> list, Comparator<? super T> c)  : This method sorts the specified list according to supplied Comparator.

These two methods are used not only just to sort the ArrayList, but also other list types like LinkedList and Vector. Let’s see how to sort an ArrayList with some simple examples.

### **How To Sort An ArrayList Of Strings?**

In this example, we are sorting an ArrayList of strings using first form of Collections.sort() method. This example sorts the string elements while considering the case of the elements.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | import java.util.ArrayList;  import java.util.Collections;    public class ListSorting  {      public static void main(String[] args)      {          //Creating an ArrayList of strings            ArrayList<String> list = new ArrayList<String>();            //Adding elements to list            list.add("Virat");            list.add("rohit");            list.add("Shikar");            list.add("ashwin");            list.add("ravindra");            list.add("Bhargav");            System.out.println("ArrayList Before Sorting :");            System.out.println(list);            //Sorting the list            Collections.sort(list);            System.out.println("ArrayList After Sorting :");            System.out.println(list);      }  } |

**Output :**

ArrayList Before Sorting :  
[Virat, rohit, Shikar, ashwin, ravindra, Bhargav]  
ArrayList After Sorting :  
[Bhargav, Shikar, Virat, ashwin, ravindra, rohit]

### **How To Sort An ArrayList Of Strings While Ignoring The Case?**

To sort an ArrayList of strings while ignoring the case of the elements, we use second form of the Collections.sort() method which takes two arguments. One is the list to be sorted and another one is the Comparator. We pass String.CASE\_INSENSITIVE\_ORDER as Comparator here. This Comparator ignores the case of the string elements.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | import java.util.ArrayList;  import java.util.Collections;    public class ListSorting  {      public static void main(String[] args)      {          //Creating an ArrayList of strings            ArrayList<String> list = new ArrayList<String>();            //Adding elements to list            list.add("Virat");            list.add("rohit");            list.add("Shikar");            list.add("ashwin");            list.add("ravindra");            list.add("Bhargav");            System.out.println("ArrayList Before Sorting :");            System.out.println(list);            //Sorting the list by ignoring the case            Collections.sort(list, String.CASE\_INSENSITIVE\_ORDER);            System.out.println("ArrayList After Sorting :");            System.out.println(list);      }  } |

**Output :**

ArrayList Before Sorting :  
[Virat, rohit, Shikar, ashwin, ravindra, Bhargav]  
ArrayList After Sorting :  
[ashwin, Bhargav, ravindra, rohit, Shikar, Virat]

### **How To Sort An ArrayList Of Custom Objects?**

In this example, we sort an ArrayList of Student objects. To do this, Student class must implement Comparable interface and override compareTo() method like below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70 | import java.util.ArrayList;  import java.util.Collections;    //Student class implementing Comparable interface    class Student implements Comparable<Student>  {      int id;        String name;        int percentage;        public Student(int id, String name, int percentage)      {          this.id = id;            this.name = name;            this.percentage = percentage;      }        @Override      public int compareTo(Student s)      {          return this.id - s.id;     //Sorts the objects in ascending order            // return s.id - this.id;    //Sorts the objects in descending order      }        @Override      public String toString()      {          return "{ID : "+id+", Name : "+name+", Percentage : "+percentage+"}";      }  }    public class MainClass  {      public static void main(String[] args)      {          //Creating an ArrayList of Student objects            ArrayList<Student> listOfStudents = new ArrayList<Student>();            //Adding students to listOfStudents            listOfStudents.add(new Student(123, "Student1", 62));            listOfStudents.add(new Student(231, "Student2", 81));            listOfStudents.add(new Student(85, "Student3", 79));            listOfStudents.add(new Student(478, "Student4", 94));            listOfStudents.add(new Student(365, "Student5", 62));            System.out.println("listOfStudents Before Sorting :");            System.out.println(listOfStudents);            //Sorting the listOfStudents            Collections.sort(listOfStudents);            System.out.println("listOfStudents After Sorting :");            System.out.println(listOfStudents);      }  } |

**Output :**

listOfStudents Before Sorting :  
[{ID : 123, Name : Student1, Percentage : 62}, {ID : 231, Name : Student2, Percentage : 81}, {ID : 85, Name : Student3, Percentage : 79}, {ID : 478, Name : Student4, Percentage : 94}, {ID : 365, Name : Student5, Percentage : 62}]  
listOfStudents After Sorting :  
[{ID : 85, Name : Student3, Percentage : 79}, {ID : 123, Name : Student1, Percentage : 62}, {ID : 231, Name : Student2, Percentage : 81}, {ID : 365, Name : Student5, Percentage : 62}, {ID : 478, Name : Student4, Percentage : 94}]

### **How To Sort An ArrayList Of Custom Objects Using Comparator?**

In this example, we sort an ArrayList of Student objects by using our own Comparator. This Comparator sorts the Student objects based on their percentage.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82 | import java.util.ArrayList;  import java.util.Collections;  import java.util.Comparator;    //Student class implementing Comparable interface    class Student implements Comparable<Student>  {      int id;        String name;        int percentage;        public Student(int id, String name, int percentage)      {          this.id = id;            this.name = name;            this.percentage = percentage;      }        @Override      public int compareTo(Student s)      {          return this.id - s.id;     //Sorts the objects in ascending order            // return s.id - this.id;    //Sorts the objects in descending order      }        @Override      public String toString()      {          return "{ID : "+id+", Name : "+name+", Percentage : "+percentage+"}";      }  }    //Defining our own Comparator    class OrderByPercentage implements Comparator<Student>  {      @Override      public int compare(Student s1, Student s2)      {          return s1.percentage - s2.percentage;      }  }    public class MainClass  {      public static void main(String[] args)      {          //Creating an ArrayList of Student objects            ArrayList<Student> listOfStudents = new ArrayList<Student>();            //Adding students to listOfStudents            listOfStudents.add(new Student(123, "Student1", 62));            listOfStudents.add(new Student(231, "Student2", 81));            listOfStudents.add(new Student(85, "Student3", 79));            listOfStudents.add(new Student(478, "Student4", 94));            listOfStudents.add(new Student(365, "Student5", 62));            System.out.println("listOfStudents Before Sorting :");            System.out.println(listOfStudents);            //Sorting the listOfStudents by percentage            Collections.sort(listOfStudents, new OrderByPercentage());            System.out.println("listOfStudents After Sorting :");            System.out.println(listOfStudents);      }  } |

**Output :**

listOfStudents Before Sorting :  
[{ID : 123, Name : Student1, Percentage : 62}, {ID : 231, Name : Student2, Percentage : 81}, {ID : 85, Name : Student3, Percentage : 79}, {ID : 478, Name : Student4, Percentage : 94}, {ID : 365, Name : Student5, Percentage : 62}]  
listOfStudents After Sorting :  
[{ID : 123, Name : Student1, Percentage : 62}, {ID : 365, Name : Student5, Percentage : 62}, {ID : 85, Name : Student3, Percentage : 79}, {ID : 231, Name : Student2, Percentage : 81}, {ID : 478, Name : Student4, Percentage : 94}]

### **How To Sort An ArrayList In The Reverse Order?**

You can sort the list in the reverse order also by passing the Comparator returned by Collections.reverseOrder() as Comparator to Collections.sort() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.util.ArrayList;  import java.util.Collections;    public class MainClass  {      public static void main(String[] args)      {          //Creating an ArrayList of integers            ArrayList<Integer> list = new ArrayList<Integer>();            //Adding elements to list            list.add(1452);            list.add(6854);            list.add(8741);            list.add(6542);            list.add(3845);            System.out.println("ArrayList Before Sorting :");            System.out.println(list);            //Sorting the list in the reverse order            Collections.sort(list, Collections.reverseOrder());            System.out.println("ArrayList Sorted In The Reverse Order :");            System.out.println(list);      }  } |

**Output :**

ArrayList Before Sorting :  
[1452, 6854, 8741, 6542, 3845]  
ArrayList Sorted In The Reverse Order :  
[8741, 6854, 6542, 3845, 1452]

# 3.5 How To Remove Duplicate Elements From ArrayList In Java?

**ArrayList** is one of the most used Collection type in java. It gives the flexibility of adding multiple null elements, duplicate elements and also maintains the insertion order of elements. While coding, you often come across the requirement where you have to remove duplicate elements from already constructed ArrayList. In this post, we will see two methods to remove duplicate elements from an ArrayList.

## **Method 1 : Removing Duplicate Elements From ArrayList Using HashSet**

In this method, we use **HashSet** to remove duplicate elements from an ArrayList. As you know, HashSet doesn’t allow duplicate elements. We use this property of HashSet to remove duplicate elements from already constructed ArrayList. But, there is one disadvantage of this method. That is, it erases the insertion order of ArrayList elements. That means, after removing the duplicate elements, elements will not be in the insertion order. Let’s see one example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | import java.util.ArrayList;  import java.util.HashSet;    public class MainClass  {      public static void main(String[] args)      {          //Constructing An ArrayList            ArrayList<String> listWithDuplicateElements = new ArrayList<String>();            listWithDuplicateElements.add("JAVA");            listWithDuplicateElements.add("J2EE");            listWithDuplicateElements.add("JSP");            listWithDuplicateElements.add("SERVLETS");            listWithDuplicateElements.add("JAVA");            listWithDuplicateElements.add("STRUTS");            listWithDuplicateElements.add("JSP");            //Printing listWithDuplicateElements            System.out.print("ArrayList With Duplicate Elements :");            System.out.println(listWithDuplicateElements);            //Constructing HashSet using listWithDuplicateElements            HashSet<String> set = new HashSet<String>(listWithDuplicateElements);            //Constructing listWithoutDuplicateElements using set            ArrayList<String> listWithoutDuplicateElements = new ArrayList<String>(set);            //Printing listWithoutDuplicateElements            System.out.print("ArrayList After Removing Duplicate Elements :");            System.out.println(listWithoutDuplicateElements);      }  } |

**Output :**

|  |  |
| --- | --- |
| 1  2 | ArrayList With Duplicate Elements :[JAVA, J2EE, JSP, SERVLETS, JAVA, STRUTS, JSP]  ArrayList After Removing Duplicate Elements :[JAVA, SERVLETS, JSP, J2EE, STRUTS] |

You notice the output of the above example. Elements are shuffled after duplicate elements are removed. They are not in the insertion order. If you want insertion order of elements to be maintained even after removing the duplicate elements, then this method is not recommended. There is another method exist which doesn’t alter the insertion order of elements even after removing the duplicate elements. That is using **LinkedHashSet**.

## **Method 2 : Removing Duplicate Elements From ArrayList Using LinkedHashSet**

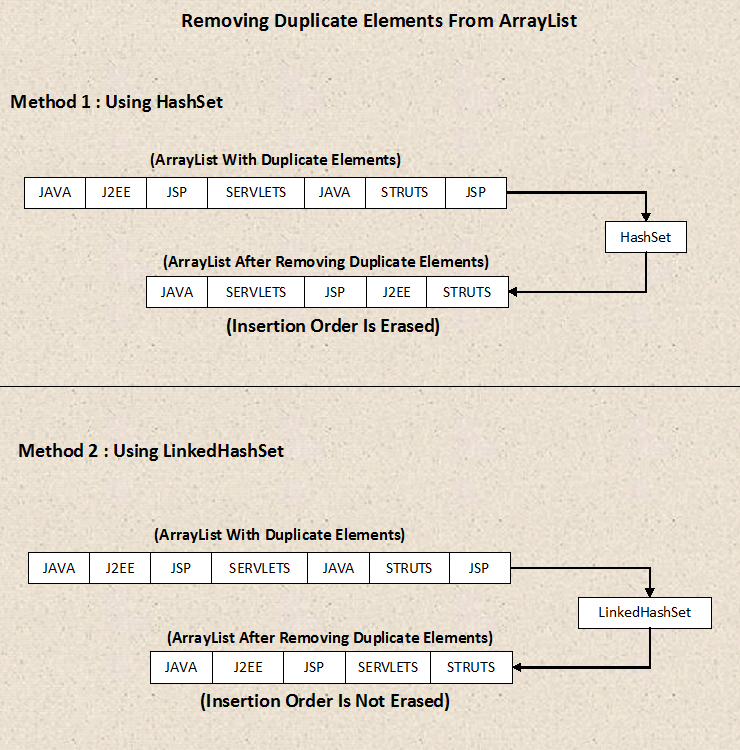
In this method, we use **LinkedHashSet** to remove duplicate elements from ArrayList. As you know that LinkedHashSet doesn’t allow duplicate elements and also maintains the insertion order of elements. Both these properties of LinkedHashSet is used here in order to remove duplicate elements from ArrayList and also maintain the insertion order of elements. See the below example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | import java.util.ArrayList;  import java.util.LinkedHashSet;    public class MainClass  {      public static void main(String[] args)      {          //Constructing An ArrayList            ArrayList<String> listWithDuplicateElements = new ArrayList<String>();            listWithDuplicateElements.add("JAVA");            listWithDuplicateElements.add("J2EE");            listWithDuplicateElements.add("JSP");            listWithDuplicateElements.add("SERVLETS");            listWithDuplicateElements.add("JAVA");            listWithDuplicateElements.add("STRUTS");            listWithDuplicateElements.add("JSP");            //Printing listWithDuplicateElements            System.out.print("ArrayList With Duplicate Elements :");            System.out.println(listWithDuplicateElements);            //Constructing LinkedHashSet using listWithDuplicateElements            LinkedHashSet<String> set = new LinkedHashSet<String>(listWithDuplicateElements);            //Constructing listWithoutDuplicateElements using set            ArrayList<String> listWithoutDuplicateElements = new ArrayList<String>(set);            //Printing listWithoutDuplicateElements            System.out.print("ArrayList After Removing Duplicate Elements :");            System.out.println(listWithoutDuplicateElements);      }  } |

**Output :**

|  |  |
| --- | --- |
| 1  2 | ArrayList With Duplicate Elements :[JAVA, J2EE, JSP, SERVLETS, JAVA, STRUTS, JSP]  ArrayList After Removing Duplicate Elements :[JAVA, J2EE, JSP, SERVLETS, STRUTS] |

Notice the output. Insertion order of elements is maintained even after the duplicate elements are removed from ArrayList.



# 3.5 How To Modify An ArrayList?

You can modify an ArrayList elementarily (Only one element is added or removed or updated) or in bulk (More than one elements are added or removed or updated).

## **1) Elementary Modification Operations On ArrayList :**

The following methods are used to perform elementary modification operations on ArrayList.

**boolean add(E e) :**

This method appends an element at the end of this List. If the ArrayList is empty then there will be exactly one element after this operation.

**void add(int index, E element) :**

This method inserts an element at the specified position.

**boolean remove(Object o) :**

It removes first occurrence of specified element from the list.

**E remove(int index) :**

This method removes an element from the specified position.

**E set(int index, E element) :**

This method replaces an element at the specified position with the passed element.

Here is an example to show elementary modifications on ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | class ArrayListDemo  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            list.add("ONE");     //Adds "ONE" at the end of the list            list.add("TWO");     //Adds "TWO" at the end of the list            list.add("THREE");   //Adds "THREE" at the end of the list            list.add("FOUR");    //Adds "FOUR" at the end of the list            System.out.println(list);    //Output : [ONE, TWO, THREE, FOUR]            list.add(3, "INSERTED");   //Inserts "INSERTED" at position 3            System.out.println(list);   //Output : [ONE, TWO, THREE, INSERTED, FOUR]            list.add(1, "INSERTED");   //Inserts "INSERTED" at position 1            System.out.println(list);     //Output : [ONE, INSERTED, TWO, THREE, INSERTED, FOUR]            list.remove("INSERTED");    //Removes first occurence of "INSERTED"            System.out.println(list);     //Output : [ONE, TWO, THREE, INSERTED, FOUR]            list.remove(3);           //Removes an element at position 3            System.out.println(list);     //Output : [ONE, TWO, THREE, FOUR]            list.set(3, "REPLACED");    //Replaces an element at position 3 with "REPLACED"            System.out.println(list);     //Output : [ONE, TWO, THREE, REPLACED]      }  } |

## **2) Bulk Modification Operations On ArrayList :**

The following methods are used to perform bulk modifications on ArrayList.

**boolean addAll(Collection<? extends E> c) :**

This method appends all elements of the passed collection at the end of this list.

**boolean addAll(int index, Collection<? extends E> c) :**

This method inserts all elements of the passed collection at the specified position in this list.

**boolean removeAll(Collection<?> c) :**

This method removes all elements of this list which are also elements of the passed collection.

**boolean retainAll(Collection<?> c) :**

This method retains only those elements in this list which are also elements of the passed collection.

**void clear() :**

This method removes all elements of the list.

Here is an example to perform bulk modifications on ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | class ArrayListDemo  {      public static void main(String[] args)      {          ArrayList<String> list1 = new ArrayList<String>();            list1.add("ONE");            list1.add("TWO");            list1.add("THREE");            list1.add("FOUR");            System.out.println(list1);     //Output : [ONE, TWO, THREE, FOUR]            ArrayList<String> list2 = new ArrayList<String>();            list2.add("THREE");            list2.add("FOUR");            list2.add("FIVE");            list2.add("SIX");            System.out.println(list2);     //Output : [THREE, FOUR, FIVE, SIX]            list1.addAll(list2);   //Appends list2 at the end of list1            System.out.println(list1);    //Output : [ONE, TWO, THREE, FOUR, THREE, FOUR, FIVE, SIX]            list1.removeAll(list2);    //Removes the elements of list1 which are also elements of list2            System.out.println(list1);    //Output : [ONE, TWO]            list1.addAll(2, list2);    //Inserts all elements of list2 into list1 at position 2            System.out.println(list1);    //Output : [ONE, TWO, THREE, FOUR, FIVE, SIX]            list1.retainAll(list2);    //Retains all elements of list1 which are also elements of list2            System.out.println(list1);    //Output : [THREE, FOUR, FIVE, SIX]            list1.clear();      //Removes all elements of list1            System.out.println(list1);   //Output : []      }  } |

# 3.7 Different Ways Of Iterating An ArrayList In Java:

## **Different Ways Of Iterating An ArrayList In Java :**

You can iterate a given ArrayList in 4 different ways. They are,

a) Iteration Using Normal for loop.

b) Iteration Using Iterator Object.

c) Iteration Using ListIterator Object.

d) Iteration Using Enhanced for loop.

Below is the detail description of all of the above methods.

## **Iteration Using Normal for loop :**

This method is useful when you also need index of the elements along with the elements itself. Using this method, you can also iterate a part of an ArrayList. Here is the template for this method.

|  |  |
| --- | --- |
| 1  2  3  4 | for (int i = 0; i < list.size(); i++)  {          type\_of\_element element = list.get(i);  } |

## **Iteration Using Iterator :**

This method is useful when you don’t want index of an element, but you want to remove the elements as you iterate through an ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | while (iterator.hasNext())  {      System.out.println(iterator.next());        //Removing an element from ArrayList      iterator.remove();  } |

## **Iteration Using ListIterator :**

If you want to iterate the list in both the directions – forward and backward, then use the ListIterator method. One more advantage of this method is, you can start iteration from a specific element in an ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | while (listIterator.hasNext() or listIterator.hasPrevious())  {      System.out.println(listIterator.next());            System.out.println(listIterator.previous());  } |

Click [here](https://javaconceptoftheday.com/difference-between-iterator-and-listiterator-in-java/) for more info on Iterator and ListIterator.

## **Iteration Using Enhanced for loop :**

This method is useful when you don’t need indexes of elements and you just want to access the elements without removing them or modifying them (it is the most common case). This method is also short and very easy to write.

|  |  |
| --- | --- |
| 1  2  3  4 | for (type\_of\_element element : list)  {      System.out.println(element);  } |

Here is the program which implements all of the above four methods.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | class ArrayListIteration  {      public static void main(String[] args)      {          ArrayList<String> list = new ArrayList<String>();            list.add("FIRST");            list.add("SECOND");            list.add("THIRD");            list.add("FOURTH");            list.add("FIFTH");            //1. Using for loop            for (int i = 0; i < list.size(); i++)          {              System.out.println(list.get(i));          }            //2. Using Iterator            Iterator<String> it = list.iterator();            while (it.hasNext())          {              System.out.println(it.next());                //Removing an element from list              it.remove();          }            //3. Using ListIterator            ListIterator<String> listIt = list.listIterator();            while (listIt.hasNext())          {              System.out.println(listIt.next());          }            //4. Using enhanced for loop            for (String element : list)          {              System.out.println(element);          }      }  } |

# 3.8 Difference Between Iterator And ListIterator In Java:

## **Difference Between Iterator And ListIterator In Java.**

**Iterator** and **ListIterator** are two interfaces in Java collection framework which are used to traverse the collections. Although ListIterator extends Iterator, there are some differences in the way they traverse the collections.

**1)** Using Iterator, you can traverse List, Set and Queue type of objects. But using ListIterator, you can traverse only List objects. In Set and Queue types, there is no method to get the ListIterator object. But, In List types, there is a method called listIterator() which returns ListIterator object.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | class IteratorAndListIterator  {      public static void main(String[] args)      {          List list = new ArrayList();            list.add("ONE");            list.add("TWO");            list.add("THREE");            //Traversing list elements using Iterator          Iterator iterator1 = list.iterator();            while (iterator1.hasNext())          {              System.out.println(iterator1.next());          }            Queue queue = new PriorityQueue(list);            //Traversing queue elements using Iterator          Iterator iterator2 = queue.iterator();            while (iterator2.hasNext())          {              System.out.println(iterator2.next());          }            Set set = new HashSet(list);            //Traversing set elements using Iterator          Iterator iterator3 = set.iterator();            while (iterator3.hasNext())          {              System.out.println(iterator3.next());          }            //Traversing list elements using ListIterator          ListIterator listIterator1 = list.listIterator();            while (listIterator1.hasNext())          {              System.out.println(listIterator1.next());          }            //Traversing queue and set elements using ListIterator is not possible            ListIterator listIterator2 = queue.listIterator();    //Compile time error, there is no such method in Queue            ListIterator listIterator3 = set.listIterator();     //Compile time error, there is no such method in Set      }  } |

**2)** Using Iterator, we can traverse the elements only in forward direction. But, using ListIterator you can traverse the elements in both the directions – forward and backward. ListIterator has those methods to support the traversing of elements in both the directions.

## **Iterator Methods :**

**boolean hasNext()** –> Checks whether collection has more elements.

**E next()**  –> Returns the next element in the collection.

**void remove()**  –> Removes the current element in the collection i.e element returned by next().

## **ListIterator Methods :**

**boolean hasNext()** –> Checks whether the list has more elements when traversing the list in forward direction.

**boolean hasPrevious()** –> Checks whether list has more elements when traversing the list in backward direction.

**E next()**  –> Returns the next element in the list and moves the cursor forward.

**E previous()**  –> Returns the previous element in the list and moves the cursor backward.

**int nextIndex()** –> Returns index of the next element in the list.

**int previousIndex()** –> Returns index of the previous element in the list.

**void remove()**  –> Removes the current element in the collection i.e element returned by next() or previous().

**void set(E e)** –> Replaces the current element i.e element returned by next() or previous() with the specified element.

**void add(E e)** –> Inserts the specified element in the list.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | class IteratorAndListIterator  {      public static void main(String[] args)      {          List<String> list = new ArrayList<String>();            list.add("FIRST");            list.add("SECOND");            list.add("THIRD");            //Traversing list elements in forward direction using Iterator            Iterator iterator = list.iterator();            while (iterator.hasNext())          {              System.out.println(iterator.next());          }            //      OUTPUT :          //      FIRST          //      SECOND          //      THIRD            //Traversing list elements in forward direction using ListIterator            ListIterator listIterator = list.listIterator();            while (listIterator.hasNext())          {              System.out.println(listIterator.next());          }            //      OUTPUT :          //      FIRST          //      SECOND          //      THIRD            //Traversing list elements in backward direction using ListIterator            while (listIterator.hasPrevious())          {              System.out.println(listIterator.previous());          }            //      OUTPUT :          //      THIRD          //      SECOND          //      FIRST      }  } |

**3)** Using ListIterator, you can obtain index of next and previous elements. But, it is not possible with Iterator interface.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | import java.util.ArrayList;  import java.util.List;  import java.util.ListIterator;    class IteratorAndListIterator  {      public static void main(String[] args)      {          List<String> list = new ArrayList<String>();            list.add("FIRST");            list.add("SECOND");            list.add("THIRD");            ListIterator listIterator = list.listIterator();            while (listIterator.hasNext())          {              //Getting index of next element                System.out.println(listIterator.nextIndex()+" : "+listIterator.next());          }            //      OUTPUT :          //      0 : FIRST          //      1 : SECOND          //      2 : THIRD            while (listIterator.hasPrevious())          {              //Getting index of previous element                System.out.println(listIterator.previousIndex()+" : "+listIterator.previous());          }            //      OUTPUT :          //      2 : THIRD          //      1 : SECOND          //      0 : FIRST      }  } |

**4)** Using ListIterator, you can perform modifications(insert, replace, remove) on the list. But, using Iterator you can only remove the elements from the collection.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | class IteratorAndListIterator  {      public static void main(String[] args)      {          List<String> list = new ArrayList<String>();            list.add("FIRST");            list.add("SECOND");            list.add("THIRD");            ListIterator<String> listIterator = list.listIterator();            System.out.println(list);       //Output :  [FIRST, SECOND, THIRD]            while (listIterator.hasNext())          {              listIterator.next();                //Modifying an element returned by next()              listIterator.set("MODIFIED");          }            System.out.println(list);       //Output :  [MODIFIED, MODIFIED, MODIFIED]            Iterator<String> iterator = list.iterator();            while (iterator.hasNext())          {              iterator.next();                //Removing an element              iterator.remove();          }            System.out.println(list);    //Output : []      }  } |

**5)** Using ListIterator, you can iterate a list from the specified index. It is not possible with Iterator.

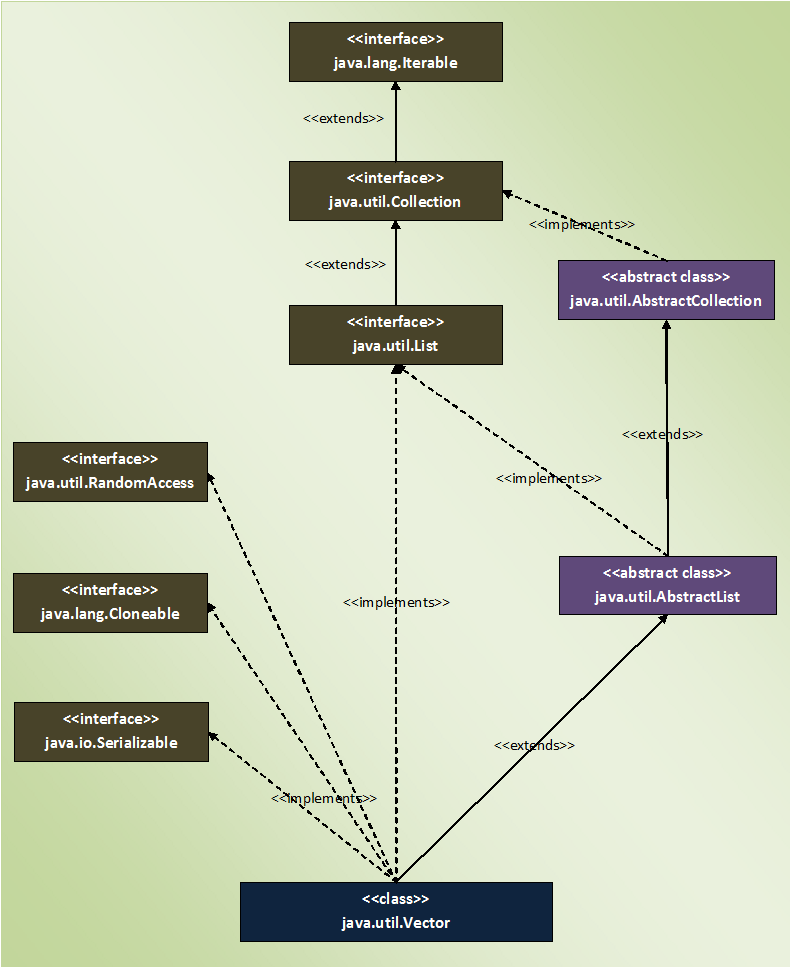
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class IteratorAndListIterator  {      public static void main(String[] args)      {          List<String> list = new ArrayList<String>();            list.add("FIRST");            list.add("SECOND");            list.add("THIRD");            list.add("FOURTH");            list.add("FIFTH");            //Iterating list from index 2 using ListIterator            ListIterator<String> listIterator = list.listIterator(2);            while (listIterator.hasNext())          {              System.out.println(listIterator.next());          }            //      OUTPUT :          //      THIRD          //      FOURTH          //      FIFTH      }  } |

# 4. Collection Framework – The Vector Class:

**The Vector Class** is also dynamically grow-able and shrink-able collection of objects like an [ArrayList](https://javaconceptoftheday.com/collection-framework-arraylist-class/) class. But, the main difference between ArrayList and Vector is that **Vector class is synchronized**. That means, only one thread can enter into vector object at any moment of time.

Vector class is preferred over ArrayList class when you are developing a multi threaded application. But, precautions need to be taken because vector may reduce the performance of your application as it is thread safety and only one thread is allowed to have object lock at any moment of time and remaining threads have to wait until a thread releases the object lock which is held by it. So, it is always recommended that if you don’t need thread safety environment, it is better to use ArrayList class than the Vector class.

Vector class has same features as ArrayList. Vector class also extends **AbstractList** class and implements **List interface**. It also implements 3 marker interfaces – **RandomAccess**, **Cloneable** and **Serializable**. Below is the hierarchy diagram of Vector class.



## **Properties Of Vector Class :**

* The main feature of Vector class is that it is thread safety. All methods of Vector class are synchronized so that only one thread can execute them at any given time. This feature of Vector class is useful when you need thread safety code.
* Thread safety property of Vector class effects the performance of an application as it makes threads to wait for object lock.
* **Capacity Increment :** Capacity increment is an amount by which the capacity of the vector is automatically incremented whenever size of the vector exceeds it’s capacity. You can pass this capacity increment while creating a vector. If you don’t pass, capacity increment will be treated as zero and capacity of the vector will be doubled whenever size exceeds capacity.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public class MainClass  {      public static void main(String[] args)      {          //Creating vector object with capacity of 3 and with default capacity increment i.e 0            Vector<Integer> vector = new Vector<Integer>(3);            //Printing Current Capacity of Vector            System.out.println(vector.capacity());      //Output : 3            //Adding 4 elements (greater than the capacity) to vector            vector.add(10);            vector.add(20);            vector.add(30);            vector.add(40);            //again Printing Current Capacity of Vector            System.out.println(vector.capacity());     //Output : 6      }  } |

* Unlike an ArrayList, you can set the size of the Vector manually. If the new size is greater than the current size, the new slots will be filled with null elements. If the new size is smaller than current size, then the extra elements will be discarded.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | public class MainClass  {      public static void main(String[] args)      {          //Creating Vector with default initial capacity of 10            Vector<Integer> vector = new Vector<Integer>();            //Adding elements to vector            vector.add(10);            vector.add(20);            vector.add(30);            vector.add(40);            //Retrieving the current size of vector            System.out.println(vector.size());      //Output : 4            //Setting the size of vector as 10.            vector.setSize(10);            //Now retrieving the current size of vector            System.out.println(vector.size());    //Output : 10            //Printing the elements of vector. notice that 6 null elements are inserted            System.out.println(vector);     //Output : [10, 20, 30, 40, null, null, null, null, null, null]            //Again changing the size of vector to 3            vector.setSize(3);            //Printing the elements of vector. notice that extra elements are removed.            System.out.println(vector);    //Output : [10, 20, 30]      }  } |

* You can traverse the vector using **Enumeration** object. Vector class has a method called **elements()** which returns an Enumeration object consisting of all elements of Vector.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | public class MainClass  {      public static void main(String[] args)      {          //Creating Vector with default initial capacity of 10            Vector<Integer> vector = new Vector<Integer>();            //Adding elements to vector            vector.add(10);            vector.add(20);            vector.add(30);            vector.add(40);            //Getting Enumeration object            Enumeration<Integer> en = vector.elements();            //traversing elements of Vector using Enumeration            while (en.hasMoreElements())          {              System.out.println(en.nextElement());          }    //      Output :    //      10  //      20  //      30  //      40      }  } |

* Vector class has separate methods to retrieve first and last element of vector object. You will not find these methods in ArrayList class. **firstElement()** retrieves first element and **lastElement()** method retrieves last element of the vector.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public class MainClass  {      public static void main(String[] args)      {          //Creating Vector with default initial capacity of 10            Vector<Integer> vector = new Vector<Integer>();            //Adding elements to vector            vector.add(10);            vector.add(20);            vector.add(30);            vector.add(40);            //Getting first element            System.out.println(vector.firstElement());     //Output : 10            //Getting last element            System.out.println(vector.lastElement());      //Output : 40      }  } |

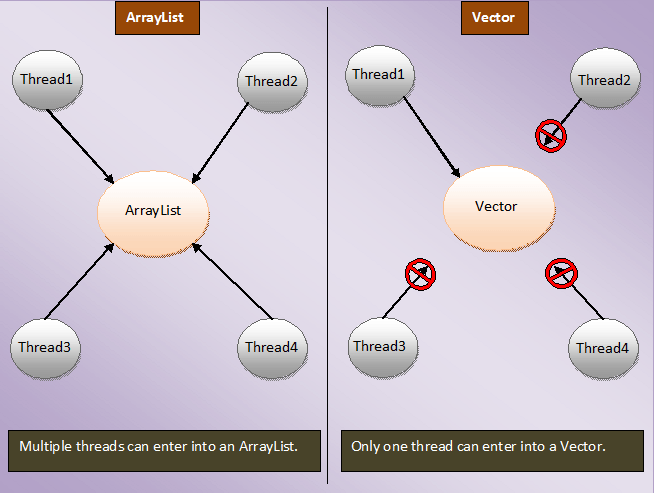
# 4.1 Difference Between ArrayList And Vector Class:

## **What is the difference between ArrayList and Vector Class?**

This is one of the most asked Java interview question on Collections for 0-2 years experienced Java professionals. In this post, I have tried to list out the differences between ArrayList and Vector classes. I hope It will be helpful for you guys.

## **1) Thread Safety**

This is the main difference between ArrayList and Vector class. ArrayList class is not thread safety where as Vector class is thread safety. Vector class is a synchronized class. Only one thread can enter into Vector object at any moment of time during execution. Where as ArrayList class is not synchronized. Multiple threads can access ArrayList object simultaneously. Below diagram clearly shows that.



## **2) Performance**

ArrayList has better performance compared to Vector. It is because, Vector class is synchronized. It makes the threads to wait for object lock to enter into vector object. Where as ArrayList class is not synchronized. Threads need not to wait for object lock to access ArrayList object. This makes ArrayList faster than the Vector class.

## **3) Capacity Increment**

Whenever the size of the ArrayList exceeds it’s capacity, the capacity is increased by half of the current capacity. Where as in case of Vector, the capacity is increased by **Capacity Increment** passed while creating the Vector object. If Capacity increment is not passed, capacity will be doubled automatically when the size exceeds it’s capacity. In ArrayList, there is no provision to pass Capacity increment while creating it. It’s capacity is automatically increased by half of the current capacity whenever size exceeds capacity.

## **4) Size**

You can manually change the current size of the vector. Vector class has a method called **setSize().** Using this method, you can change the current size of the vector. If the new size is greater than the current size, new slots will be filled with null elements and if the new size is smaller than the current size, extra elements will be discarded. But in case of ArrayList, you can’t change the current size manually. It doesn’t have methods which alter it’s size. The size of the ArrayList will be changed only when you add or delete it’s elements.

## **5) Traversing The Elements.**

ArrayList elements can be traversed using Iterator, ListIterator and using either normal or advanced for loop. But, vector elements can be traversed using Enumeration also along with these methods. Vector class has a method called **elements()** which returns Enumeration object containing all elements of the vector. Where as ArrayList does not have such methods.

## **6) Searching The Elements.**

In ArrayList, you have to start searching for a particular element from the beginning of an Arralist. But in the Vector, you can start searching for a particular element from a particular position in a vector. This makes the search operation in Vector faster than in ArrayList.

## **7) Legacy Code**

Vector class is considered as Legacy code. Because, it exist in Java before the introduction of Collection Framework. Earlier it was not a part of Collections. Later it has been included in Collections. But, the older methods of vector class have been retained as it is.

# 4.2 Why Not To Use Vector Class In Your Code?

Vector class is often considered as obsolete or “**Due for Deprecation**” by many experienced Java developers. They always recommend and advise not to use Vector class in your code. They prefer using ArrayList over Vector class. In this article, I have tried to list out some points regarding why not to use Vector class in your code.

## **1) You can achieve Thread Safety without Vector.**

Vector class has only one advantage over ArrayList i.e it is thread safety. But, you can achieve thread safe ArrayList by using **synchronizedList()** method of Collections class. Below is the sample code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class MainClass  {      public static void main(String[] args)      {          ArrayList<Integer> list = new ArrayList<Integer>();            Collections.synchronizedList(list);            //It returns Synchronized list backed by original list.      }  } |

## **2) Thread Safeness of Vector class is time consuming.**

All methods of Vector class are synchronized. This makes each and every operation on Vector object thread safe. But, it is time consuming. Because, you need to acquire object lock for each operation you want to perform on vector object. Usually, you need set of operations to be synchronized not each and every operation.  Isn’t make sense to take the object lock once, perform the operations you want and then release the lock when you are done. Why acquire the lock again and again for each operations?. This is the time consuming process and decreases the performance of your application.

## **3) Enumeration Vs Iterator**

Vector class has a method which return Enumeration over the elements of Vector object. Although, Enumerations are faster than the Iterator, but it is not backed by the original collection. That means, any changes made to original collection does not reflect in Enumeration object. They ignore the modifications done during iteration. This may cause issues.

## **4) Is Vector class poorly designed?**

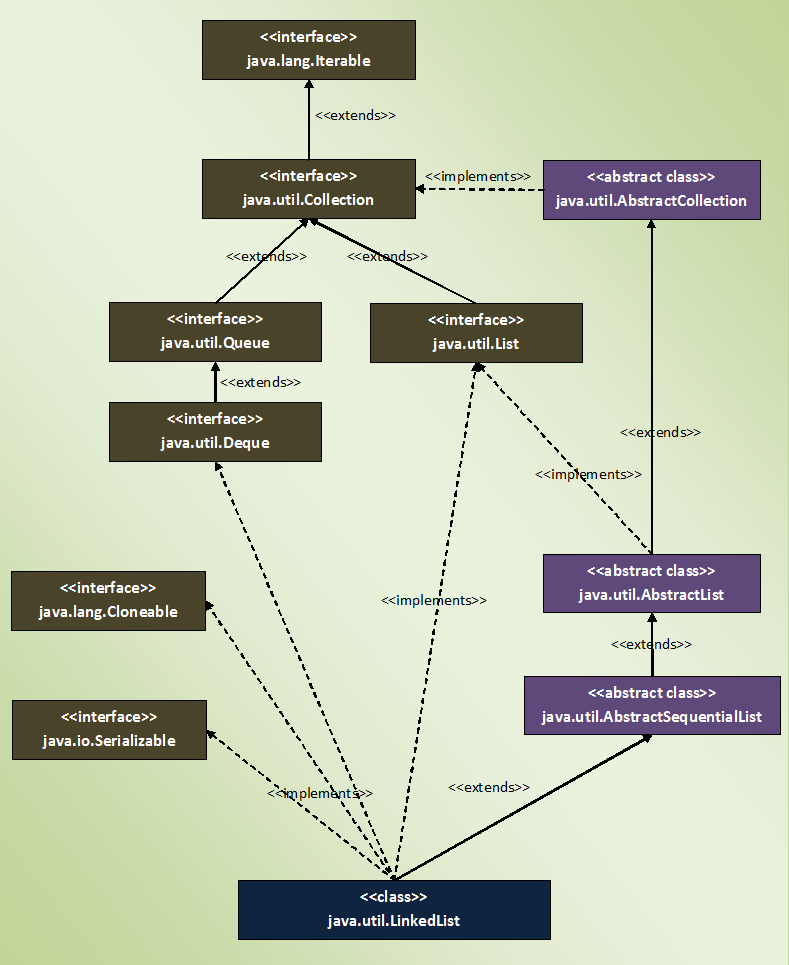
Vector class combines two features – “**Re-sizable Array**” and “**Synchronization**“. This makes poor design. Because, if you need just “Re-sizable Array” and you use Vector class for that, you will get “synchronized Resizable Array” not just re-sizable array. This may reduce the performance of your application. Therefore, instead of using Vector class, always use ArrayList class. You will have re-sizable array and whenever you want to make it synchronized, use Collections.SynchronizedList().

# 5. Java Collection Framework – The LinkedList Class:

In general terms, LinkedList is a data structure where each element consist of three things. First one is the reference to previous element, second one is the actual value of the element and last one is the reference to next element.

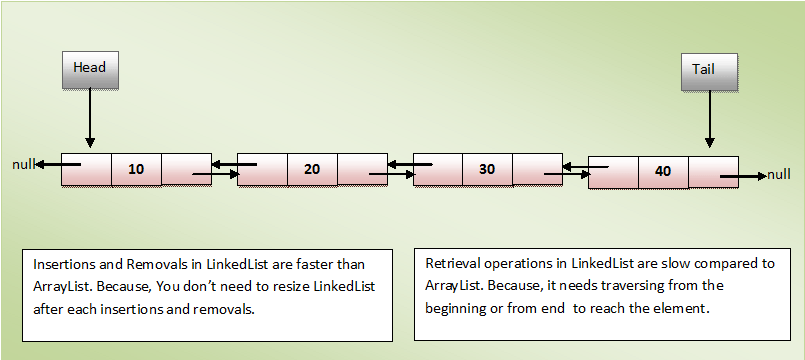
The **LinkedList** class in Java is an implementation of doubly linked list which can be used both as a **List** as well as **Queue**. The LinkedList in java can have any type of elements including null and duplicates. Elements can be inserted and can be removed from both the ends and can be retrieved from any arbitrary position.

The LinkedList class extends **AbstractSequentialList** and implements **List** and **Deque** interfaces. It also implements 2 marker interfaces – **Cloneable** and **Serializable**. Here is the hierarchy diagram of LinkedList class in Java.



## **Properties Of LinkedList Class In Java:**

* Elements in the LinkedList are called as **Nodes**. Where each node consist of three parts – Reference To Previous Element, Value Of The Element and Reference To Next Element. Below diagram shows how LinkedList looks like.



* Reference To Previous Element of first node and Reference To Next Element of last node are null as there will be no elements before the first node and after the last node.
* You can insert the elements at both the ends and also in the middle of the LinkedList. Below is the list of methods for insertion operations.

|  |  |  |
| --- | --- | --- |
| Insertion At Head | Insertion In The Middle | Insertion At Tail |
| addFirst(E e) | add(int index, E e) | add(E e) |
| offerFirst(E e) | addAll(int index, Collection c) | addAll(Collection c) |
|  |  | offer(E e) |
|  |  | offerLast(E e) |

* You can remove the elements from the head, from the tail and also from the middle of the LinkedList.

|  |  |  |
| --- | --- | --- |
| Removing from head | Removing from the middle | Removing from the tail |
| poll() | Remove(int index) | pollLast() |
| pollFirst() |  | removeLast() |
| remove() |  |  |
| removeFirst() |  |  |

* You can retrieve the elements form the head, from the middle and from the tail of the LinkedList. Below is the list of retrieval methods.

|  |  |  |
| --- | --- | --- |
| Retrieving from the head | Retrieving from the middle | Retrieving from the tail |
| element() | get(int index) | getLast() |
| getFirst() |  | peekLast() |
| peek() |  |  |
| peekFirst() |  |  |

* Insertion and removal operations in LinkedList are faster than the ArrayList. Because in LinkedList, there is no need to shift the elements after each insertion and removal. only references of next and previous elements need to be changed.
* Retrieval of the elements is very slow in LinkedList as compared to ArrayList. Becaues in LinkedList, you have to traverse from beginning or end (whichever is closer to the element) to reach the element.
* The LinkedList can be used as **stack**. It has the methods pop() and push() which make it to function as Stack.
* The LinkedList can also be used as ArrayList, Queue, SIngle linked list and doubly linked list.
* LinkedList can have multiple **null** elements.
* LinkedList can have **duplicate** elements.
* LinkedList class in Java is not of type **Random Access**. i.e the elements can not be accessed randomly. To access the given element, you have to traverse the LinkedList from beginning or end (whichever is closer to the element) to reach the given element.

# 5.1 ArrayList Vs LinkedList In Java:

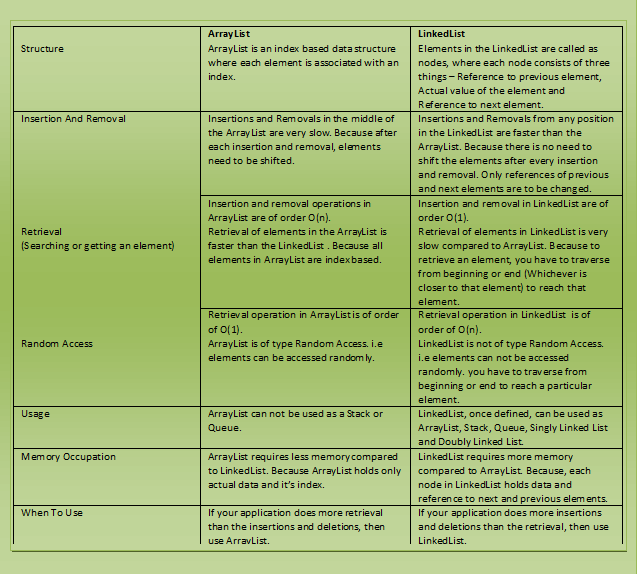
## **ArrayList Vs LinkedList In Java :**

Although both **ArrayList** and **LinkedList** implement List interface, they have some differences between them. The performance and internal working nature of both varies significantly. There are also some similarities between them. In this article, we will see both differences and similarities between ArrayList and LinkedList in Java.

## **Differences Between ArrayList And LinkedList In Java:**

|  |  |  |
| --- | --- | --- |
|  | **ArrayList** | **LinkedList** |
| Structure | ArrayList is an index based data structure where each element is associated with an index. | Elements in the LinkedList are called as nodes, where each node consists of three things – Reference to previous element, Actual value of the element and Reference to next element. |
| Insertion And Removal | Insertions and Removals in the middle of the ArrayList are very slow. Because after each insertion and removal, elements need to be shifted. | Insertions and Removals from any position in the LinkedList are faster than the ArrayList. Because there is no need to shift the elements after every insertion and removal. Only references of previous and next elements are to be changed. |
| Insertion and removal operations in ArrayList are of order O(n). | Insertion and removal in LinkedList are of order O(1). |
| Retrieval(Searching or getting an element) | Retrieval of elements in the ArrayList is faster than the LinkedList . Because all elements in ArrayList are index based. | Retrieval of elements in LinkedList is very slow compared to ArrayList. Because to retrieve an element, you have to traverse from beginning or end (Whichever is closer to that element) to reach that element. |
| Retrieval operation in ArrayList is of order of O(1). | Retrieval operation in LinkedList is of order of O(n). |
| Random Access | ArrayList is of type Random Access. i.e elements can be accessed randomly. | LinkedList is not of type Random Access. i.e elements can not be accessed randomly. you have to traverse from beginning or end to reach a particular element. |
| Usage | ArrayList can not be used as a Stack or Queue. | LinkedList, once defined, can be used as ArrayList, Stack, Queue, Singly Linked List and Doubly Linked List. |
| Memory Occupation | ArrayList requires less memory compared to LinkedList. Because ArrayList holds only actual data and it’s index. | LinkedList requires more memory compared to ArrayList. Because, each node in LinkedList holds data and reference to next and previous elements. |
| When To Use | If your application does more retrieval than the insertions and deletions, then use ArrayList. | If your application does more insertions and deletions than the retrieval, then use LinkedList. |

Below is the pictorial representation of the above table.



## **Similarities Between ArrayList And LinkedList In Java :**

* Both ArrayList and LinkedList implement **List interface**.
* Both ArrayList and LinkedList are **Cloneable** and **Serializable**.
* Both ArrayList and LinkedList maintain **insertion order**.
* Both are **non synchronized**.

# 5.2 16 Java LinkedList Programming Examples:

## **Java LinkedList Programming Examples**

**1) Given an element, how do you find out whether that element exist in a LinkedList or not. If it exist retrieve the position of that element?**

First use contains() method to check whether LinkedList contains the given element or not. If it contains, retrieve it’s position using indexOf() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | public class LinkedListExample  {      public static void main(String[] args)      {          LinkedList<String> list = new LinkedList<String>();            //Adding elements at the end of the list            list.add("JAVA");            list.add("J2EE");            list.add("JSP");            list.add("SERVLETS");            list.add("JDBC");            //Printing the elements of list            System.out.println(list);      //Output : [JAVA, J2EE, JSP, SERVLETS, JDBC]            String s = "JSP";            //Checking whether list contains "JSP"            boolean contains = list.contains(s);            if(contains)          {              //If list contains "JSP", printing it's index                System.out.println(list.indexOf(s));      //Output : 2          }            s = "STRUTS";            //Checking whether list contains "STRUTS"            contains = list.contains("STRUTS");            if(contains)          {              //If list contains "STRUTS", printing it's index                System.out.println(list.indexOf(s));          }      }  } |

**2) Write a Java program to traverse the elements of a LinkedList in reverse direction?**

This can be done using **descendingIterator()** method which returns an Iterator object containing all elements of a LinkedList in the reverse order i.e from tail to head.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | public class LinkedListExample  {      public static void main(String[] args)      {          LinkedList<String> list = new LinkedList<String>();            //Adding elements at the end of the list            list.add("JAVA");            list.add("J2EE");            list.add("JSP");            list.add("SERVLETS");            list.add("JDBC");            //Printing the elements of list            System.out.println(list);      //Output : [JAVA, J2EE, JSP, SERVLETS, JDBC]            //Getting the Iterator object using descendingIterator() method            Iterator<String> it = list.descendingIterator();            //printing the elements of list in reverse order            while (it.hasNext())          {              System.out.println(it.next());          }      }  } |

**3) How do you join an ArrayList at the end of a LinkedList?**

Using addAll() method, we can append an ArrayList or any other Collection type at the end of a LinkedList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | public class LinkedListExample  {      public static void main(String[] args)      {          //Creating a LinkedList            LinkedList<String> linkedList = new LinkedList<String>();            //Adding elements at the end of the linkedList            linkedList.add("ONE");            linkedList.add("TWO");            linkedList.add("THREE");            linkedList.add("FOUR");            linkedList.add("FIVE");            //Printing the elements of linkedList            System.out.println(linkedList);      //Output : [ONE, TWO, THREE, FOUR, FIVE]            //Creating an ArrayList            ArrayList<String> arrayList = new ArrayList<String>();            arrayList.add("SIX");            arrayList.add("SEVEN");            arrayList.add("EIGHT");            arrayList.add("NINE");            //Printing the elements of ArrayList            System.out.println(arrayList);      //Output : [SIX, SEVEN, EIGHT, NINE]            //Appending arrayList at the end of linkedList            linkedList.addAll(arrayList);            //Printing the elements of linkedList            System.out.println(linkedList);     //Output : [ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE]      }  } |

**4) Write a Java program which implements LinkedList as a Queue (FIFO)?**

Use the offer() and poll() methods which make LinkedList to work as a Queue.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public class LinkedListExamples  {      public static void main(String[] args)      {          LinkedList<Integer> queue = new LinkedList<Integer>();            //adding the elements into the queue            queue.offer(10);            queue.offer(20);            queue.offer(30);            queue.offer(40);            //Printing the elements of queue            System.out.println(queue);      //Output : [10, 20, 30, 40]            //Removing the elements from the queue            System.out.println(queue.poll());    //Output : 10            System.out.println(queue.poll());    //Output : 20      }  } |

**5) How do you insert an element at the head and tail of a LinkedList?**

You can use add() or addLast() or offer() or offerLast() to add the elements at the tail of a LinkedList and to add the elements at the head of a LinkedList, use either addFirst() or offerFirst().

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | public class LinkedListExample  {      public static void main(String[] args)      {          LinkedList<Integer> list = new LinkedList<Integer>();            //Adding elements at the end of the list            list.add(10);            list.addLast(20);            list.offer(30);            list.offerLast(40);            //Printing the elements of list            System.out.println(list);      //Output : [10, 20, 30, 40]            //Adding elements at the beginning of the list            list.offerFirst(1);            list.addFirst(2);            //Printing the elements of list            System.out.println(list);     //Output : [2, 1, 10, 20, 30, 40]      }  } |

**6) How do you add an element or collection of elements at a specific position of a LinkedList?**

You have to use add(int index, E element) to add an element at specific position of a LinkedList and to add collection of elements at specific position, use addAll(int index, Collection c) where ‘c’ is a collection of elements to add.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | public class LinkedListExample  {      public static void main(String[] args)      {          LinkedList<Integer> list = new LinkedList<Integer>();            //Adding elements at the end of the list            list.add(10);            list.add(20);            list.add(30);            list.add(40);            //Printing the elements of list            System.out.println(list);      //Output : [10, 20, 30, 40]            //Adding an element at index 2            list.add(2, 9999);            //Printing the elements of list            System.out.println(list);     //Output : [10, 20, 9999, 30, 40]            //Creating another LinkedList with elements to add            LinkedList<Integer> list1 = new LinkedList<Integer>();            //Adding elements at the beginning of the list1            list1.addFirst(111);            list1.addFirst(222);            list1.addFirst(333);            //Printing the elements of list1            System.out.println(list1);     //Output : [333, 222, 111]            //Adding all elements of list1 at index 3 of list            list.addAll(3, list1);            //Printing the elements of list            System.out.println(list);    //Output : [10, 20, 9999, 333, 222, 111, 30, 40]      }  } |

**7) How do you remove the elements of a LinkedList from both the ends?**

You can use pollLast() and removeLast() to remove the elements from the tail of a LinkedList and to remove the elements from the head of a LinkedList, use poll() or pollFirst() or remove() or removeFirst().

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | public class LinkedListExample  {      public static void main(String[] args)      {          LinkedList<String> list = new LinkedList<String>();            //Adding elements at the end of the list            list.add("ONE");            list.add("TWO");            list.add("THREE");            list.add("FOUR");            list.add("FIVE");            list.add("SIX");            list.add("SEVEN");            //Printing the elements of list            System.out.println(list);      //Output : [ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN]            //Removing the elements from the head of the LinkedList            list.poll();            list.pollFirst();            list.remove();            list.removeFirst();            //Printing the elements of list            System.out.println(list);     //Output : [FIVE, SIX, SEVEN]            //Removing elements from the end of the LinkedList            list.pollLast();            list.removeLast();            //Printing the elements of list            System.out.println(list);     //Output : [FIVE]      }  } |

**8) How do you replace an element at a specific position of a LinkedList with the given element?**

To replace an element at specific position of a LinkedList , use set() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public class LinkedListExample  {      public static void main(String[] args)      {          LinkedList<String> list = new LinkedList<String>();            //Adding elements at the end of the list            list.add("ONE");            list.add("TWO");            list.add("THREE");            list.add("FOUR");            //Printing the elements of list            System.out.println(list);      //Output : [ONE, TWO, THREE, FOUR]            //Replacing an element at index 2 with "ZERO"            list.set(2, "ZERO");            System.out.println(list);     //Output : [ONE, TWO, ZERO, FOUR]      }  } |

**9) How do you retrieve but not remove the elements of a LinkedList from both the ends?**

To retrieve but not remove an element from the head of a LinkedList, use element() or getFirst() or peek() or peekFirst() methods and to retrieve the elements from the tail of a LinkedList, use getLast() or peekLast() methods.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | public class LinkedListExample  {      public static void main(String[] args)      {          LinkedList<String> list = new LinkedList<String>();            //Adding elements at the end of the list            list.add("FIRST");            list.add("SECOND");            list.add("THIRD");            list.add("FOURTH");            list.add("FIFTH");            //Printing the elements of list            System.out.println(list);      //Output : [FIRST, SECOND, THIRD, FOURTH, FIFTH]            //Retrieving the elements from the head            System.out.println(list.element());      //Output : FIRST            System.out.println(list.getFirst());     //Output : FIRST            System.out.println(list.peek());        //Output : FIRST            System.out.println(list.peekFirst());     //Output : FIRST            //Retrieving the elements from the tail            System.out.println(list.peekLast());     //Output : FIFTH            System.out.println(list.getLast());      //Output : FIFTH      }  } |

**10) How do you retrieve and remove and only retrieve an element from specific position of a LinkedList?**

You can use remove(int index) to retrieve and remove an element from specific position of a LinkedList. To only retrieve an element, ust get(int index) method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | public class LinkedListExample  {      public static void main(String[] args)      {          LinkedList<String> list = new LinkedList<String>();            //Adding elements at the end of the list            list.add("JAVA");            list.add("J2EE");            list.add("JSP");            list.add("SERVLETS");            list.add("JDBC");            //Printing the elements of list            System.out.println(list);      //Output : [JAVA, J2EE, JSP, SERVLETS, JDBC]            //Retrieving and removing an element at index 2 of the list            System.out.println(list.remove(2));     //Output : JSP            //Printing the elements of list            System.out.println(list);     //Output : [JAVA, J2EE, SERVLETS, JDBC]            //Only retrieving an element at index 2 of the list            System.out.println(list.get(2));     //Output : SERVLETS      }  } |

**11) How do you get the number of elements in a LinkedList?**

Using size() method. This method returns the number of elements in a LinkedList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public class LinkedListExample  {      public static void main(String[] args)      {          LinkedList<String> list = new LinkedList<String>();            //Adding elements at the end of the list            list.add("JAVA");            list.add("J2EE");            list.add("JSP");            list.add("SERVLETS");            list.add("JDBC");            //Printing the elements of list            System.out.println(list);      //Output : [JAVA, J2EE, JSP, SERVLETS, JDBC]            //Getting the number of elements in list            System.out.println(list.size());     //Output : 5      }  } |

**12) How do you remove the first occurrence and last occurrence of a given element in a LinkedList?**

Use the **removeFirstOccurrence(Object 0)** to remove the first occurrence of a given element and use **removeLastOccurrence(Object 0)** to remove last occurrence of a given element in a LinkedList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | public class LinkedListExample  {      public static void main(String[] args)      {          LinkedList<String> list = new LinkedList<String>();            //Adding elements at the end of the list            list.add("JAVA");            list.add("J2EE");            list.add("JSP");            list.add("J2EE");            list.add("JDBC");            //Printing the elements of list            System.out.println(list);      //Output : [JAVA, J2EE, JSP, J2EE, JDBC]            //Removing the first occurrence of "J2EE"            list.removeFirstOccurrence("J2EE");            //Printing the elements of list            System.out.println(list);      //Output : [JAVA, JSP, J2EE, JDBC]            //Removing the last occurrence of "J2EE"            list.removeLastOccurrence("J2EE");            //Printing the elements of list            System.out.println(list);      //Output : [JAVA, JSP, JDBC]      }  } |

**13) How do you use LinkedList as Stack (LIFO)?**

LinkedList has pop() and push() methods which make LinkedList to function as a Stack.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public class LinkedListExample  {      public static void main(String[] args)      {          LinkedList<Integer> stack = new LinkedList<Integer>();            //pushing the elements into the stack            stack.push(10);            stack.push(20);            stack.push(30);            stack.push(40);            //Printing the elements of stack            System.out.println(stack);      //Output : [40, 30, 20, 10]            //Poping out the elements from the stack            System.out.println(stack.pop());    //Output : 40            System.out.println(stack.pop());    //Output : 30      }  } |

**14) How do you remove all elements of a LinkedList?**

Using clear() method. This method removes all elements of a LinkedList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | public class LinkedListExample  {      public static void main(String[] args)      {          LinkedList<Integer> linkedList = new LinkedList<Integer>();            //adding the elements to LinkedList            linkedList.add(10);            linkedList.add(20);            linkedList.add(30);            linkedList.add(40);            linkedList.add(50);            //Printing the elements of LinkedList            System.out.println(linkedList);       //Output : [10, 20, 30, 40, 50]            //Removing all elements of linkedList            linkedList.clear();            //Printing the elements of LinkedList            System.out.println(linkedList);      //Output : []      }  } |

**15) How do you create clone of a LinkedList?**

Using clone() method of LinkedList class. This method creates shallow copy of the original LinkedList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | public class LinkedListExample  {      public static void main(String[] args)      {          LinkedList<Integer> linkedList1 = new LinkedList<Integer>();            //adding the elements to linkedList1            linkedList1.add(10);            linkedList1.add(20);            linkedList1.add(30);            linkedList1.add(40);            linkedList1.add(50);            //Printing the elements of linkedList1            System.out.println(linkedList1);       //Output : [10, 20, 30, 40, 50]            //Creating another LinkedList            LinkedList<Integer> linkedList2 = new LinkedList<Integer>();            //Cloning the linkedList1 into linkedList2            linkedList2 = (LinkedList<Integer>) linkedList1.clone();            //Printing the elements of linkedList2            System.out.println(linkedList2);     //Output : [10, 20, 30, 40, 50]      }  } |

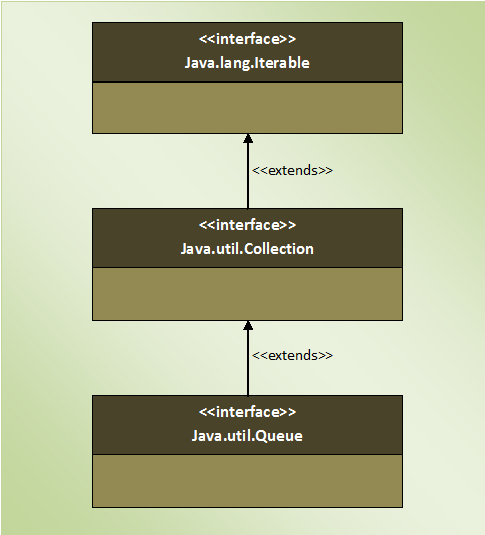
**16) How do you get the position of last occurrence of a given element in a LinkedList?**

Using lastIndexOf() method, we can retrieve the position of last occurrence of a given element in a LinkedList.

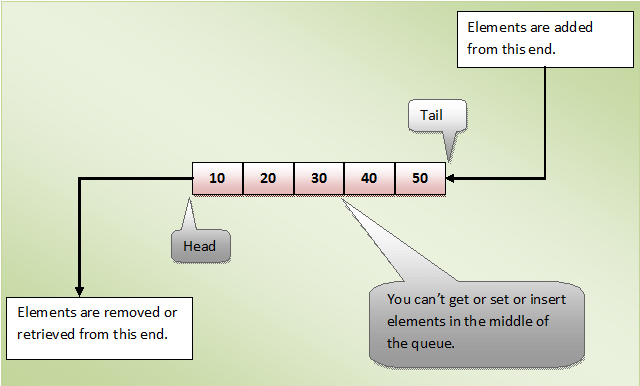
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class LinkedListExample  {      public static void main(String[] args)      {          LinkedList<String> linkedList = new LinkedList<String>();            //adding the elements to linkedList            linkedList.add("AAA");            linkedList.add("BBB");            linkedList.add("CCC");            linkedList.add("BBB");            linkedList.add("FFF");            linkedList.add("BBB");            //Printing the elements of linkedList            System.out.println(linkedList);       //Output : [AAA, BBB, CCC, BBB, FFF, BBB]            //Getting the position of last occurrence of "BBB"            System.out.println(linkedList.lastIndexOf("BBB"));    //Output : 5      }  } |

# 6. Collection Framework – The Queue Interface:

The Queue Interface extends Collection interface. It defines queue data structure which is normally **First-In-First-Out**. Queue is a data structure in which elements are added from one end and elements are deleted from another end. But, exception being the Priority Queue in which elements are removed from one end, but elements are added according to the order defined by the supplied comparator. Here is the hierarchy diagram of Queue interface.



## **How Typical Queue Works?**

Queue is a data structure where elements are added from one end called **tail** of the queue and elements are removed from another end called **head** of the queue. Queue is also **first-in-first-out** type of data structure (except priority queue). That means an element which is inserted first will be the first element to be removed from the queue. You can’t add or get or set elements at an arbitrary position in the queues. Here is the diagram which shows how typical queue works.  


## **Properties Of Queue :**

* **Null** elements are not allowed in the queue. If you try to insert null object into the queue, it throws NullPointerException.
* Queue can have **duplicate** elements.
* Unlike a normal list, queue is **not random access**. i.e you can’t set or insert or get elements at an arbitrary positions.
* In most of cases, elements are inserted at one end called **tail** of the queue and elements are removed or retrieved from another end called **head** of the queue.
* In the Queue Interface, there are two methods to obtain and remove the elements from the head of the queue. They are **poll()** and **remove()**. The difference between them is, poll() returns null if the queue is empty and remove() throws an exception if the queue is empty.
* There are two methods in the Queue interface to obtain the elements but don’t remove. They are **peek()** and **element()**. peek() returns null if the queue is empty and element() throws an exception if the queue is empty.

## **Methods Of Queue Interface:**

Here are the methods of Queue interface. Some of the methods throw an exception if operation is not possible and some methods return a value (null or false) if operation is not possible.

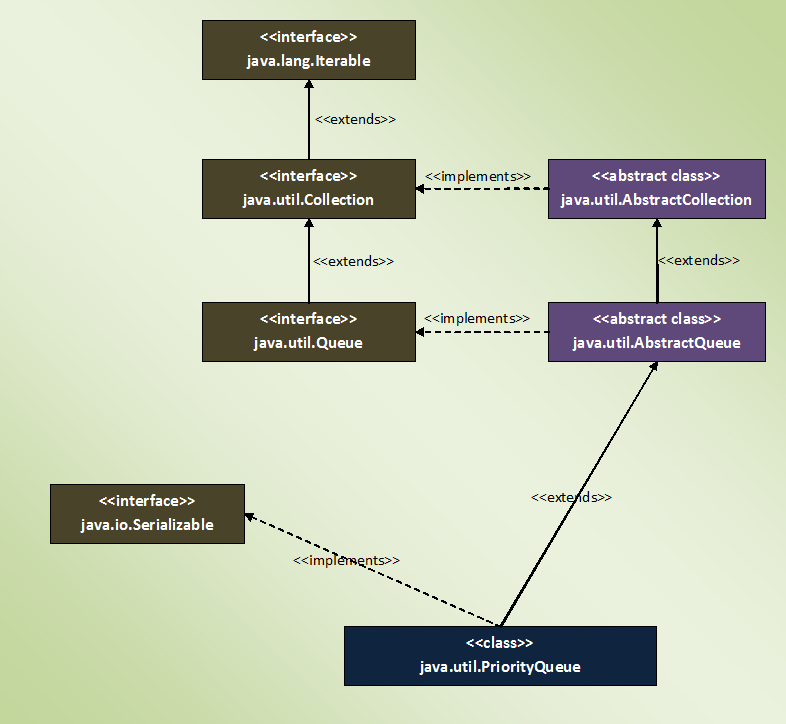
|  |  |  |
| --- | --- | --- |
| Operation | Throws An Exception If operation is not possible | Returns null or false if operation is not possible |
| Add an element to the queue. | add() | offer() |
| Retrieve an element from the head of the queue. | element() | peek() |
| Retrieve And Remove an element from the head of the queue. | remove() | poll() |

# 7. Collection Framework – The PriorityQueue:

The **PriorityQueue** is a queue in which elements are ordered according to specified Comparator. You have to specify this Comparator while creating a PriorityQueue itsel. If no Comparator is specified, elements will be placed in their natural order. The PriorityQueue is a special type of queue because it is not a **First-In-First-Out** (FIFO) as in the normal queues. But, elements are placed according to supplied Comaparator.

The PriorityQueue does not allow **null** elements. Elements in the PriorityQueue must be of **Comparable** type, If you insert the elements which are not Comparable, you will get ClassCastException at run time.

PriorityQueue class extends **AbstractQueue** class which in turn implements **Queue** interface. PriorityQueue also implements one marker interface – **java.io.Serializable** interface. Below is the hierarchy diagram of PriorityQueue class.



## **Properties Of PriorityQueue Class :**

* Elements in the PriorityQueue are ordered according to supplied **Comparator**. If Comparator is not supplied, elements will be placed in their natural order.
* The PriorityQueue is **unbounded**. That means the capacity of the PriorityQueue increases automatically if the size exceeds capacity. But, how it grows is not specified.
* The PriorityQueue can have **duplicate** elements but can not have **null** elements.
* All elements of the PriorityQueue must be of **Comparable type**. Otherwise ClassCastException will be thrown at run time.
* The head element of the PriorityQueue is always the least element and tail element is always the largest element according to specified Comparator.
* The default initial capacity of PriorityQueue is **11**.
* You can retrieve the Comparator used to order the elements of the PriorityQueue using **comparator()** method.
* PriorityQueue is not a thread safe.

# 7.1 Java PriorityQueue Example:

If you are a Java developer, you wouldn’t have developed a single application without using collections in your code. Java collections are of great use when you are dealing with a group of objects. Some times you need to process the group of objects on a priority basis. You may need to order them on a particular criteria. For example, you may want to order employee records in the ascending order of their salaries or you may want to order the customers on their id’s. In such scenarios, **PriorityQueue** is of great help.

In this particular article, we will discuss two examples of PriorityQueue – One with the default Comparator and another one with the customized comparator. You can go through the some basic definitions and properties of PriorityQueue [here](https://javaconceptoftheday.com/java-collection-framework-priorityqueue-class/).

## **Java PriorityQueue Example With Default Comparator :**

You already know that if you don’t supply the Comparator while creating a PriorityQueue, elements will be ordered in natural ascending order. In this example, we create a PriorityQueue of Integers without supplying a Comparator like this,

|  |  |
| --- | --- |
| 1 | PriorityQueue<Integer> pQueue = new PriorityQueue<Integer>(); |

As we are not passing any Comparator, elements of ‘**pQueue**‘ will be placed in the ascending order. Let’s add some elements to this PriorityQueue.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | pQueue.offer(21);    pQueue.offer(17);    pQueue.offer(37);    pQueue.offer(41);    pQueue.offer(9);    pQueue.offer(67);    pQueue.offer(31); |

You know that head element of the PriorityQueue always will be the least element. Let’s remove the elements of **‘pQueue’** one by one using poll() method ( poll() method removes the head of the queue ).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | System.out.println(pQueue.poll());     //Output : 9    System.out.println(pQueue.poll());     //Output : 17    System.out.println(pQueue.poll());     //Output : 21    System.out.println(pQueue.poll());     //Output : 31    System.out.println(pQueue.poll());     //Output : 37    System.out.println(pQueue.poll());     //Output : 41    System.out.println(pQueue.poll());     //Output : 67 |

You can notice that always the least element is removed from the **‘pQueue’**. That means elements in the **‘pQueue’** are placed in the ascending order. The whole example can be written like this,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | public class PriorityQueueExample  {      public static void main(String[] args)      {          //Creating a PriorityQueue with default Comparator.            PriorityQueue<Integer> pQueue = new PriorityQueue<Integer>();            //Inserting elements into pQueue.            pQueue.offer(21);            pQueue.offer(17);            pQueue.offer(37);            pQueue.offer(41);            pQueue.offer(9);            pQueue.offer(67);            pQueue.offer(31);            //Removing the head elements            System.out.println(pQueue.poll());     //Output : 9            System.out.println(pQueue.poll());     //Output : 17            System.out.println(pQueue.poll());     //Output : 21            System.out.println(pQueue.poll());     //Output : 31            System.out.println(pQueue.poll());     //Output : 37            System.out.println(pQueue.poll());     //Output : 41            System.out.println(pQueue.poll());     //Output : 67      }  } |

## **Java PriorityQueue Example With Customized Comparator :**

In this example, we create a PriorityQueue with our own Comparator. We try to create a PriorityQueue of ‘**Employee**‘ objects ordered in the ascending order of their salaries. That means head element always will be an ‘**Employee**‘ object with lowest salary.

Let’s define **‘Employee’** class with two attributes –  **‘name’** and **‘salary’**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class Employee  {      String name;        int salary;        //Constructor Of Employee        public Employee(String name, int salary)      {          this.name = name;            this.salary = salary;      }        @Override      public String toString()      {          return name+" : "+salary;      }  } |

In the above class, toString() method is overrided so that it returns the contents of the object.

Let’s define our own Comparator class ‘**MyComparator**‘ which compares the salary of two Employees.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class MyComparator implements Comparator<Employee>  {      @Override      public int compare(Employee e1, Employee e2)      {          return e1.salary - e2.salary;      }  } |

Let’s create a PriorityQueue of **‘Employee’** objects with ‘**MyComparator**‘ as a Comparator.

|  |  |
| --- | --- |
| 1  2  3 | MyComparator comparator = new MyComparator();    PriorityQueue<Employee> pQueue = new PriorityQueue<Employee>(7, comparator); |

Let’s insert some **‘Employee’** objects into **‘pQueue’**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | pQueue.offer(new Employee("AAA", 15000));    pQueue.offer(new Employee("BBB", 12000));    pQueue.offer(new Employee("CCC", 7500));    pQueue.offer(new Employee("DDD", 17500));    pQueue.offer(new Employee("EEE", 21500));    pQueue.offer(new Employee("FFF", 29000));    pQueue.offer(new Employee("GGG", 14300)); |

Let’s remove the head elements of the **‘pQueue’** one by one.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | System.out.println(pQueue.poll());       //Output --> CCC : 7500    System.out.println(pQueue.poll());       //Output --> BBB : 12000    System.out.println(pQueue.poll());       //Output --> GGG : 14300    System.out.println(pQueue.poll());       //Output --> AAA : 15000    System.out.println(pQueue.poll());       //Output --> DDD : 17500    System.out.println(pQueue.poll());       //Output --> EEE : 21500    System.out.println(pQueue.poll());       //Output --> FFF : 29000 |

You can notice that always an **Employee** of lowest salary is removed. That means, head element always contains **Employee** object with lowest salary. ‘**Employee**‘ objects in ‘**pQueue**‘ are placed in the ascending order of their salary.

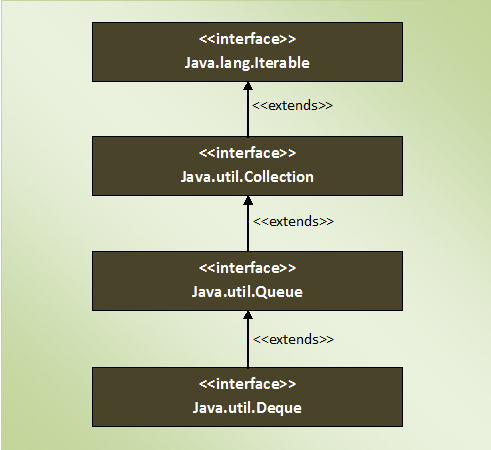
Here is the whole code of the above example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80 | //Employee Class    class Employee  {      String name;        int salary;        //Constructor Of Employee        public Employee(String name, int salary)      {          this.name = name;            this.salary = salary;      }        @Override      public String toString()      {          return name+" : "+salary;      }  }    //MyComparator Class    class MyComparator implements Comparator<Employee>  {      @Override      public int compare(Employee e1, Employee e2)      {          return e1.salary - e2.salary;      }  }    public class PriorityQueueExample  {      public static void main(String[] args)      {          //Instantiating MyComaparator            MyComparator comparator = new MyComparator();            //Creating PriorityQueue of Employee objects with MyComparator as Comparator            PriorityQueue<Employee> pQueue = new PriorityQueue<Employee>(7, comparator);            //Adding Employee objects to pQueue            pQueue.offer(new Employee("AAA", 15000));            pQueue.offer(new Employee("BBB", 12000));            pQueue.offer(new Employee("CCC", 7500));            pQueue.offer(new Employee("DDD", 17500));            pQueue.offer(new Employee("EEE", 21500));            pQueue.offer(new Employee("FFF", 29000));            pQueue.offer(new Employee("GGG", 14300));            //Removing the head elements            System.out.println(pQueue.poll());       //Output --> CCC : 7500            System.out.println(pQueue.poll());       //Output --> BBB : 12000            System.out.println(pQueue.poll());       //Output --> GGG : 14300            System.out.println(pQueue.poll());       //Output --> AAA : 15000            System.out.println(pQueue.poll());       //Output --> DDD : 17500            System.out.println(pQueue.poll());       //Output --> EEE : 21500            System.out.println(pQueue.poll());       //Output --> FFF : 29000      }  } |

# 8. Collection Framework – The Deque Interface:

The Deque is the short name for “**Double Ended Queue**“. As the name suggest, Deque is a linear collection of objects which supports insertion and removal of elements from both the ends. The Deque interface defines the methods needed to insert, retrieve and remove the elements from both the ends.

The Deque interface is introduced in Java SE 6. It extends Queue interface. Here is the hierarchy diagram of Deque interface.

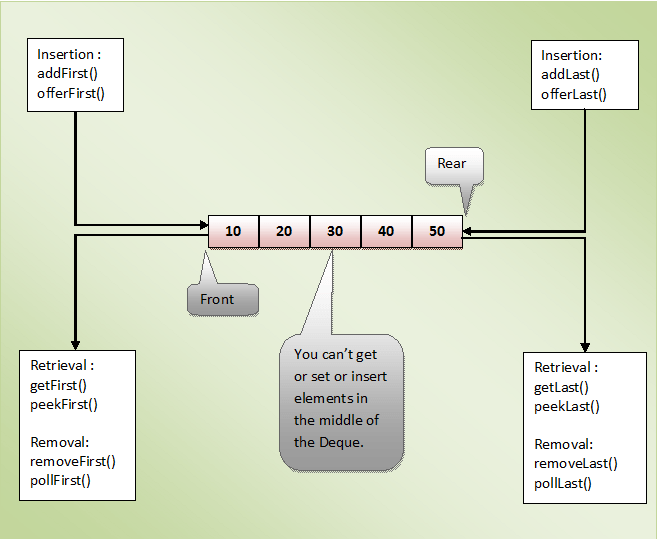


The main advantage of Deque is that you can use it as both **Queue** (FIFO) as well as **Stack** (LIFO). The Deque interface has all those methods required for FIFO and LIFO operations. Some of those methods throw an exception if operation is not possible and some methods return a special value (null or false) if operation fails. Here is the list of Deque Methods.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | | Throws an exception if operation fails. | Returns null or false if operation fails. |
| Insertion | Front End | addFirst() | offerFirst() |
| Rear End | addLast() | offerLast() |
| Retrieval | Front End | getFirst() | peekFirst() |
| Rear End | getLast() | peekLast() |
| Retrieval And Removal | Front End | removeFirst() | pollFirst() |
| Rear End | removeLast() | pollLast() |

## **How Deque – Double Ended Queue Works?**

As already said, Deque is nothing but the double ended queue. That means, you can insert, retrieve and remove the elements from both the ends. Below diagram shows how Deque works.



## **Deque As Queue :**

As Deque interface extends Queue interface, it inherits all methods of Queue interface. So, you can use all those inherited methods to perform Queue operations. Along with them, methods defined in the Deque interface can also be used for Queue operations. Below is the list of Queue methods and their equivalent Deque methods.

|  |  |
| --- | --- |
| Queue Methods | Equivalent Deque Methods |
| add() | addLast() |
| offer() | OfferLast() |
| element() | getFirst() |
| peek() | peekFirst() |
| remove() | removeFirst() |
| poll() | pollFirst() |

## **Deque As Stack :**

Deque interface has two more methods – **pop()** and **push()**. These two methods make Deque to function as a stack (Last-In-First-Out). Along with these two methods, you can also use addFirst(), peekFirst() and removeFirst() for stack operations. Below is the list of Stack methods and their equivalent methods of Deque.

|  |  |
| --- | --- |
| Stack Methods | Equivalent Deque Methods |
| push() | addFirst() |
| pop() | removeFirst() |
| peek() | peekFirst() |

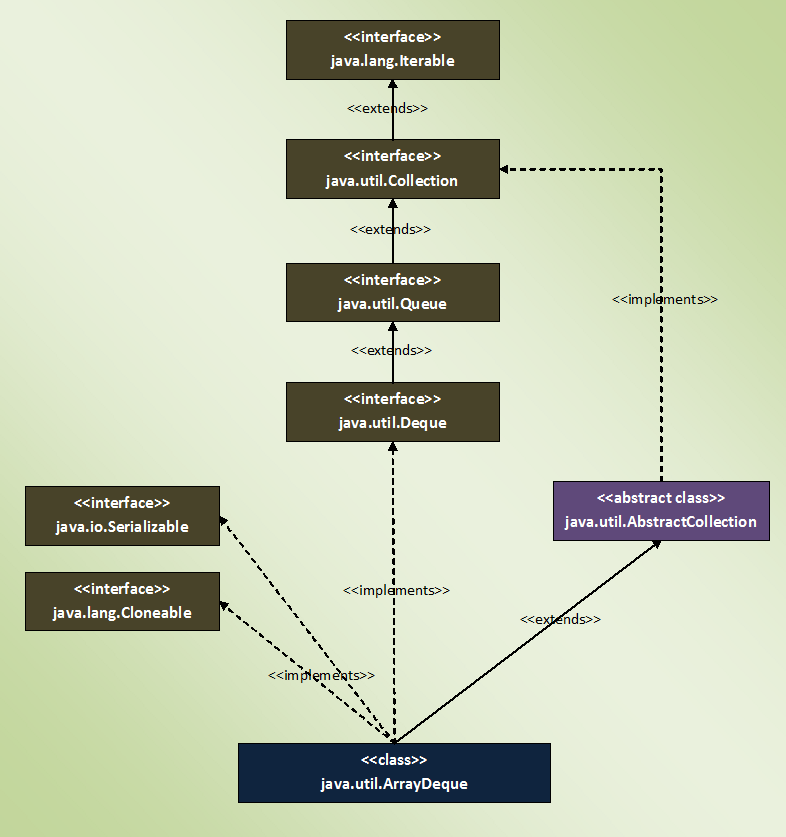
## **Properties Of Deque :**

* Unlike Queue, Deque can have **null** elements. But, it is recommended not to insert null elements as many methods return null to indicate Deque is empty.
* Deque can have **duplicate** elements.
* You can’t set or get or insert the elements at an arbitrary position of Deque. i.e **Random access** is not possible with the Deque.
* You can use removeFirstOccurrenec(E e), removeLastOccurrence(E e) and remove(E e) methods to delete the elements from the Deque.

# 8.1 Java Collection Framework – The ArrayDeque Class:

**The ArrayDeque class**in Javais introduced from JDK 1.6. It is an implementation of **Deque Interface** which allows insertion of elements at both the ends. It does not have any restrictions on capacity. It expands automatically as we add more elements. The ArrayDeque class extends **AbstractCollection** class and implements **Deque** interface. It also implements **Cloneable** and **Serializable** marker interfaces.

Below is the hierarchy diagram of ArrayDeque class.



## **Properties Of ArrayDeque Class :**

* ArrayDeque is a r**esizable-array** implementation of Deque interface like ArrayList class which is a resizable-array implementation of List interface. But, **ArrayDeque is not a List.**
* ArrayDeque does not have any capacity limit. It will grow automatically as we add elements.
* Default initial capacity of ArrayDeque is **16**. It will increase at a power of 2 (24, 25, 26 and so on) when size exceeds capacity.
* ArrayDeque can be used as a **stack** (LIFO) as well as a **queue**(FIFO). ArrayDeque is faster than the Stack class when used as a stack and faster than the LinkedList class when used as a queue.
* Performance of ArrayDeque is sometimes considered as the best among the collection framework. It gives performance of **O(1)** for insertion, removal and retrieval operations. ArrayDeque class is recommended instead of Stack class (when you want stack data structure) and instead of LinkedList class (when you want queue data structure).
* You can’t perform **indexed operations** on ArrayDeque. ArrayDeque doesn’t have the methods to support those operations.
* ArrayDeque is not a thread safe.

## **Examples Of ArrayDeque class :**

**1) ArrayDeque As Queue**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | public class ArrayDequeExample  {      public static void main(String[] args)      {          //Creating an array deque            ArrayDeque<String> arrayDeque = new ArrayDeque<String>();            //Adding elements at the tail of arrayDeque            arrayDeque.offer("One");            arrayDeque.offer("Two");            arrayDeque.offer("Three");            arrayDeque.offer("Four");            arrayDeque.offer("Five");            //Printing the elements of arrayDeque            System.out.println(arrayDeque);     //Output : [One, Two, Three, Four, Five]            //Removing the elements from the head of arrayDeque            System.out.println(arrayDeque.poll());    //Output : One            System.out.println(arrayDeque.poll());    //Output : Two      }  } |

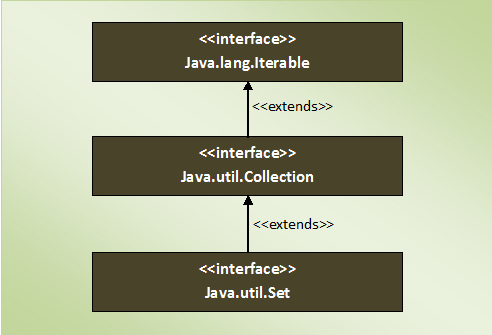
**2) ArrayDeque As Stack**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | public class ArrayDequeExample  {      public static void main(String[] args)      {          //Creating an array deque            ArrayDeque<String> arrayDeque = new ArrayDeque<String>();            //pushing elements into arrayDeque            arrayDeque.push("One");            arrayDeque.push("Two");            arrayDeque.push("Three");            arrayDeque.push("Four");            arrayDeque.push("Five");            //Printing the elements of arrayDeque            System.out.println(arrayDeque);     //Output : [Five, Four, Three, Two, One]            //popping up the elements from arrayDeque            System.out.println(arrayDeque.pop());    //Output : Five            System.out.println(arrayDeque.pop());    //Output : Four      }  } |

# 9. Collection Framework – The Set Interface:

The Set interface defines a set. The **set** is a linear collection of objects with no duplicates. Duplicate elements are not allowed in a set. The **Set interface** extends Collection interface. Set interface does not have it’s own methods. All it’s methods are inherited from Collection interface. The only change that has been made to Set interface is that add() method will return false if you try to insert an element which is already present in the set.

Below is the hierarchy diagram of Set interface.



## **Properties Of Set :**

* Set contains only unique elements. It does not allow **duplicates**.
* Set can contain only one **null** element.
* **Random access** of elements is not possible.
* **Order of elements** in a set is implementation dependent. **HashSet** elements are ordered on hash code of elements. **TreeSet** elements are ordered according to supplied Comparator (If no Comparator is supplied, elements will be placed in ascending order) and **LinkedHashSet** maintains insertion order.
* Set interface contains only methods inherited from Collection interface. It does not have it’s own methods. But, applies restriction on methods so that duplicate elements are always avoided.
* One more good thing about Set interface is that the **stronger contract** between equals() and hashCode() methods. According to this contract, you can compare two set instances of different implementation types (HashSet, TreeSet and LinkedHashSet).
* Two set instances, irrespective of their implementation types, are said to be equal if they contain same elements.

## **Methods Of Set Interface :**

Here is the list of Set interface methods. All methods are inherited from Collection interface.

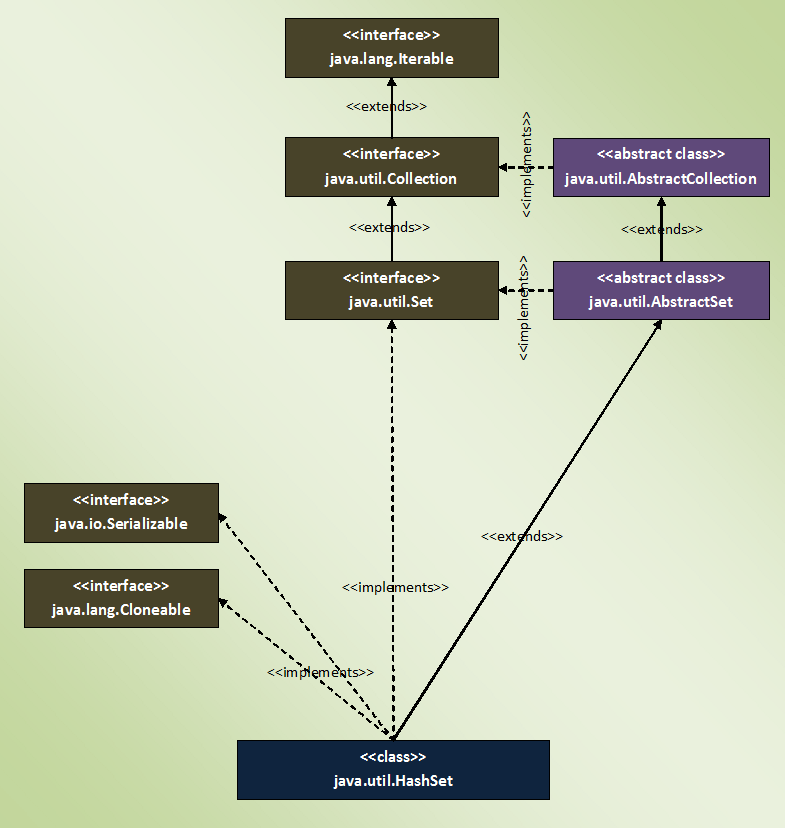
|  |  |  |
| --- | --- | --- |
| **SL No.** | **Method** | **Description** |
| **1** | int size() | Returns the number of elements in the set. |
| **2** | boolean isEmpty() | Checks whether the set is empty or not. |
| **3** | boolean contains(Object o) | Checks whether this set has specified element. |
| **4** | Iterator<E> iterator() | Returns an iterator over the set. |
| **5** | Object[] toArray() | It returns an array containing all elements of the set. |
| **6** | <T> T[] toArray(T[] a) | It returns an array of specified type containing all elements of this set. |
| **7** | boolean add(E e) | This method adds specified element to this set only if that element doesn’t present already. It returns true if element is added successfully otherwise returns false. |
| **8** | boolean remove(Object o) | Removes the specified element from this set. |
| **9** | boolean containsAll(Collection<?> c) | It checks whether this set contains all elements of passed collection. |
| **10** | boolean addAll(Collection<? extends E> c) | Adds all elements of the passed collection to this set if they are not already present. |
| **11** | boolean removeAll(Collection<?> c) | Removes all elements of this set which are also elements of passed collection. |
| **12** | boolean retainAll(Collection<?> c) | Retains only those elements in this set which are also elements of passed collection. |
| **13** | void clear() | Removes all elements in this set. |
| **14** | boolean equals(Object o) | Compares the specified object with this set for equality. |
| **15** | int hashCode() | Returns the hash code value of this set. |

# 9.1. Java Collection Framework – The HashSet Class:

**The HashSet class** in Java is an implementation of **Set interface**. HashSet is a collection of objects which contains only unique elements. Duplicates are not allowed in HashSet. HashSet gives constant time performance for insertion, removal and retrieval operations. It allows only one null element.

The HashSet internally uses **HashMap** to store the objects. The elements you insert in HashSet will be stored as keys of that HashMap object and their values will be a constant called **PRESENT**. This constant is defined as **private static final Object PRESENT = new Object()**in the source code of HashSet class.

HashSet class extends **AbstractSet class** and implements **Set interface**. It also implements **Cloneable** and **Serializable** marker interfaces. Below is the hierarchy diagram of HashSet class.



## **Properties Of HashSet Class In Java :**

1) HashSet class internally uses **HashMap** to store the objects. The elements you enter into HashSet will be stored as keys of HashMap and their values will be a constant.

2) HashSet does not allow **duplicate** elements. If you try to insert a duplicate element, older element will be overwritten.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class HashSetExample  {      public static void main(String[] args)      {          //Creating HashSet object            HashSet<String> set = new HashSet<String>();            //Adding elements to HashSet            set.add("JAVA");            set.add("JSP");            set.add("STRUTS");            set.add("HIBERNATE");            set.add("JSP");            set.add("JAVA");            //Printing the elements of HashSet            System.out.println(set);     //Output : [STRUTS, HIBERNATE, JSP, JAVA]            //You can notice that duplicate elements are not added to HashSet      }  } |

3) HashSet can have maximum one **null** element.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | public class HashSetExample  {      public static void main(String[] args)      {          //Creating HashSet object            HashSet<String> set = new HashSet<String>();            //Adding elements to HashSet            set.add("ONE");            set.add("TWO");            set.add("THREE");            set.add("FOUR");            //Adding 3 null elements to hashSet            set.add(null);            set.add(null);            set.add(null);            //Printing the elements of HashSet            System.out.println(set);     //Output : [null, ONE, TWO, THREE, FOUR]            //You can notice that HashSet contains only one null element      }  } |

4) HashSet doesn’t maintain any order. The order of the elements will be largely unpredictable. And it also doesn’t guarantee that order will remain constant over time.

5) HashSet offers constant time performance for insertion, removal and retrieval operations.

6) HashSet class is not synchronized. If you want synchronized HashSet, use **Collections.synchronizedSet()** method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public class HashSetExample  {      public static void main(String[] args)      {          //Creating HashSet object            HashSet<String> set = new HashSet<String>();            //Adding elements to HashSet            set.add("BANGALORE");            set.add("DELHI");            set.add("CHENNAI");            set.add("MUMBAI");            set.add("AHMEDABAD");            //getting synchronized set            Set<String> syncSet = Collections.synchronizedSet(set);      }  } |

# 9.2. Java HashSet Internal Working:

**HashSet** internally uses HashMap to store it’s elements. Whenever you create a HashSet object, one **HashMap** object associated with it is also created. This HashMap object is used to store the elements you enter in the HashSet. The elements you add into HashSet are stored as **keys** of this HashMap object. The value associated with those keys will be a **constant**. In this post, we will see Java HashSet internal working with an example.

Every constructor of HashSet class internally creates one HashMap object. You can check this in the source code of HashSet class. Below is the some sample code of the constructors of HashSet class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | private transient HashMap<E,Object> map;    //Constructor - 1    public HashSet()  {          map = new HashMap<>();          //Creating internally backing HashMap object  }    //Constructor - 2    public HashSet(Collection<? extends E> c)  {          map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));     //Creating internally backing HashMap object          addAll(c);  }    //Constructor - 3    public HashSet(int initialCapacity, float loadFactor)  {          map = new HashMap<>(initialCapacity, loadFactor);        //Creating internally backing HashMap object  }    //Constructor - 4    public HashSet(int initialCapacity)  {          map = new HashMap<>(initialCapacity);          //Creating internally backing HashMap object  } |

You can notice that each and every constructor internally creates one new HashMap object.

## **Java HashSet Internal Working :**

Whenever you insert an element into HashSet using **add()** method, it actually creates an entry in the internally backing HashMap object with element you have specified as it’s key and constant called “**PRESENT**” as it’s value. This “PRESENT” is defined in the HashSet class as below.

|  |  |
| --- | --- |
| 1  2 | // Dummy value to associate with an Object in the backing Map  private static final Object PRESENT = new Object(); |

Let’s have a look at add() method of HashSet class.

|  |  |
| --- | --- |
| 1  2  3  4 | public boolean add(E e)  {          return map.put(e, PRESENT)==null;  } |

You can notice that, add() method of HashSet class internally calls put() method of backing HashMap object by passing the element you have specified as a key and constant “PRESENT” as it’s value.

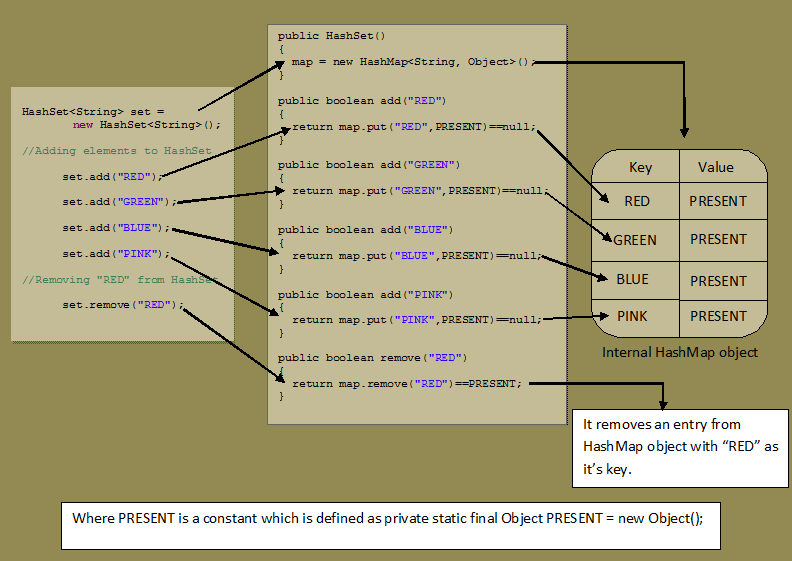
remove() method also works in the same manner.

|  |  |
| --- | --- |
| 1  2  3  4 | public boolean remove(Object o)  {          return map.remove(o)==PRESENT;  } |

Let’s see one example of HashSet and how it maintains HashMap internally.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class HashSetExample  {      public static void main(String[] args)      {          //Creating One HashSet object            HashSet<String> set = new HashSet<String>();            //Adding elements to HashSet            set.add("RED");            set.add("GREEN");            set.add("BLUE");            set.add("PINK");            //Removing "RED" from HashSet            set.remove("RED");      }  } |

See the below picture how above program works internally. You can observe that internal HashMap object contains elements of HashSet as keys and constant “PRESENT” as their value.



In the same manner, all methods of HashSet class process internally backing HashMap object to get the desired result. If you know how HashMap works, it will be easy for you to understand how HashSet works. You go through the source code of HashSet class once, you will get a clear picture about how HashSet works internally in Java.

# 9.3. Java HashSet Example:

Java **HashSet** is very powerful Collection type when you want a collection of unique objects. HashSet doesn’t allow duplicate elements. HashSet also gives constant time performance for insertion, removal and retrieval operations. It is also important to note that HashSet doesn’t maintain any order. So, It is recommended to use HashSet if you want a collection of unique elements and order of elements is not so important. If you want your elements to be ordered in some way, you can use **LinkedHashSet** or **TreeSet**.

In this Java article, we will see one real time example of HashSet.

Let’s create one **HashSet of Student records** where each Student record contains three fields – **name**, **rollNo** and **department**. In these, rollNo will be unique for all students.

Let’s create **Student** class with three fields – **name**, **rollNo** and **department**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | class Student  {      String name;        int rollNo;        String department;        public Student(String name, int rollNo, String department)      {          this.name = name;            this.rollNo = rollNo;            this.department = department;      }        @Override      public int hashCode()      {          return rollNo;      }        @Override      public boolean equals(Object obj)      {          Student student = (Student) obj;            return (rollNo == student.rollNo);      }        @Override      public String toString()      {          return rollNo+", "+name+", "+department;      }  } |

You can notice that **hashCode()** and **equals()** methods are overrided in the above class so that two Students objects will be compared solely based on **rollNo**. That means, two Student objects having same rollNo will be considered as duplicates irrespective of other fields.

Create one HashSet object containing elements of Student type.

|  |  |
| --- | --- |
| 1 | HashSet<Student> set = new HashSet<Student>(); |

Add some elements to this HashSet.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | set.add(new Student("Avinash", 121, "ECE"));    set.add(new Student("Bharat", 101, "EEE"));    set.add(new Student("Malini", 151, "Civil"));    set.add(new Student("Suresh", 200, "IT"));    set.add(new Student("Vikram", 550, "CS"));    set.add(new Student("Bharat", 301, "IT"));    set.add(new Student("Amit", 301, "IT"));           //duplicate element    set.add(new Student("Bhavya", 872, "ECE"));    set.add(new Student("Naman", 301, "CS"));        //duplicate element    set.add(new Student("Samson", 565, "Civil")); |

Iterate through this HashSet.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Iterator<Student> it = set.iterator();    while (it.hasNext())  {      Student student = (Student) it.next();        System.out.println(student);  } |

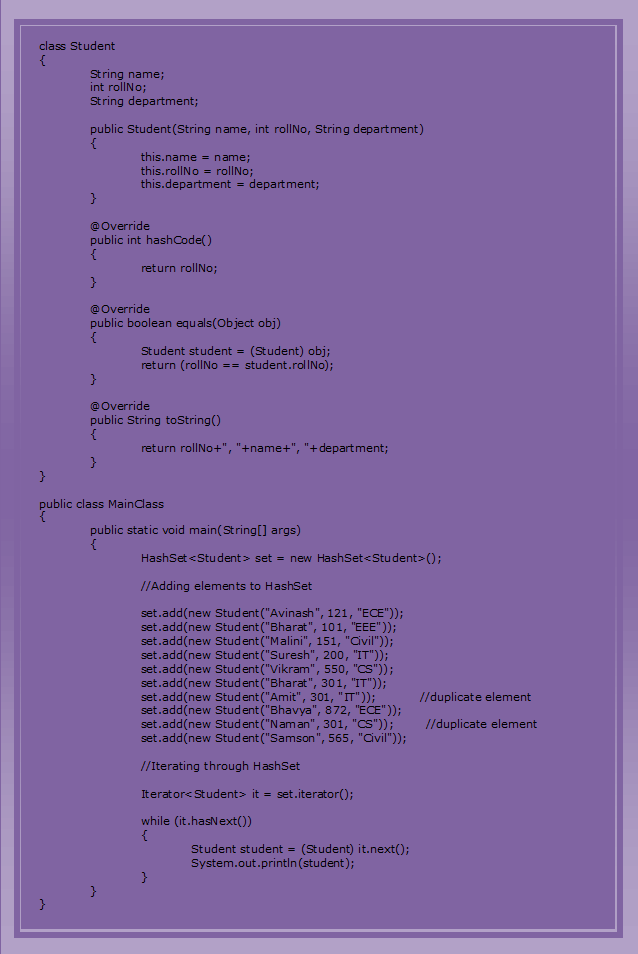
Output will be,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 550, Vikram, CS  565, Samson, Civil  101, Bharat, EEE  200, Suresh, IT  872, Bhavya, ECE  301, Bharat, IT  121, Avinash, ECE  151, Malini, Civil |

You can notice that duplicate elements are not added to HashSet.

Here is the whole program.

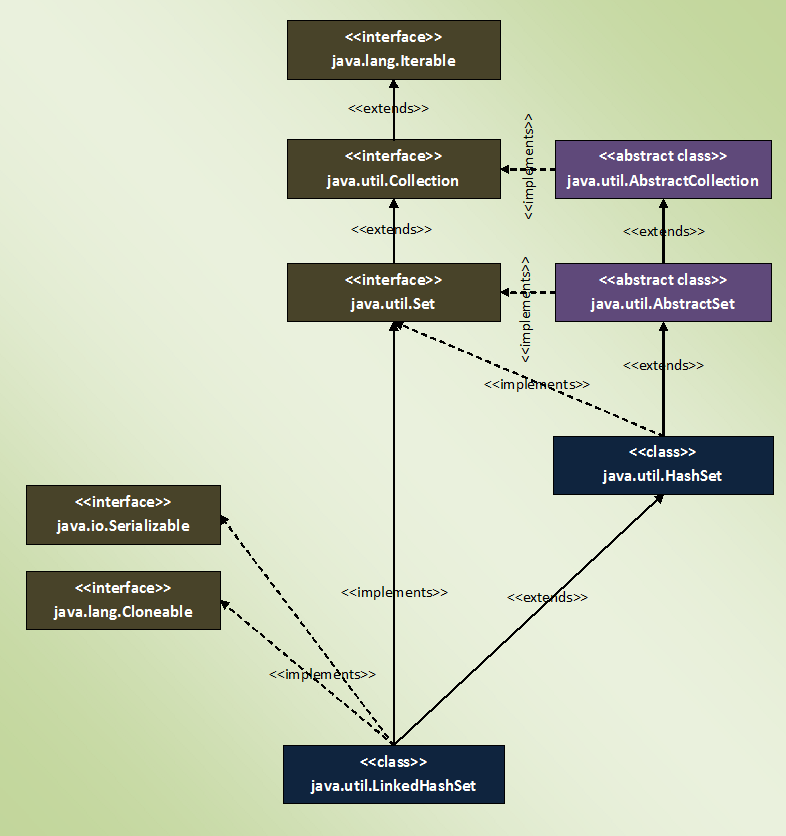
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78 | class Student  {      String name;        int rollNo;        String department;        public Student(String name, int rollNo, String department)      {          this.name = name;            this.rollNo = rollNo;            this.department = department;      }        @Override      public int hashCode()      {          return rollNo;      }        @Override      public boolean equals(Object obj)      {          Student student = (Student) obj;            return (rollNo == student.rollNo);      }        @Override      public String toString()      {          return rollNo+", "+name+", "+department;      }  }    public class MainClass  {      public static void main(String[] args)      {          HashSet<Student> set = new HashSet<Student>();            //Adding elements to HashSet            set.add(new Student("Avinash", 121, "ECE"));            set.add(new Student("Bharat", 101, "EEE"));            set.add(new Student("Malini", 151, "Civil"));            set.add(new Student("Suresh", 200, "IT"));            set.add(new Student("Vikram", 550, "CS"));            set.add(new Student("Bharat", 301, "IT"));            set.add(new Student("Amit", 301, "IT"));           //duplicate element            set.add(new Student("Bhavya", 872, "ECE"));            set.add(new Student("Naman", 301, "CS"));        //duplicate element            set.add(new Student("Samson", 565, "Civil"));            //Iterating through HashSet            Iterator<Student> it = set.iterator();            while (it.hasNext())          {              Student student = (Student) it.next();                System.out.println(student);          }      }  } |



# 10. Java Collection Framework – The LinkedHashSet Class:

The **LinkedHashSet** in java is an ordered version of **HashSet** which internally maintains one **doubly linked list** running through it’s elements. This doubly linked list is responsible for maintaining the insertion order of the elements. Unlike HashSet which maintains no order, LinkedHashSet maintains **insertion order** of elements. i.e elements are placed in the order they are inserted. LinkedHashSet is recommended over HashSet if you want a unique collection of objects in an insertion order.

The **LinkedHashSet class** extends **HashSet class** and implements **Set interface**. It also implements **Cloneable** and **Serializable** marker interfaces. Below is the hierarchy diagram of LinkedHashSet class in java.



## **Properties Of LinkedHashSet Class In Java:**

* LinkedHashSet internally uses **LinkedHashMap** to store it’s elements just like HashSet which internally uses HashMap to store it’s elements.
* LinkedHashSet maintains **insertion order**. This is the main difference between LinkedHashSet and HashSet.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | public class LinkedHashSetExample  {      public static void main(String[] args)      {          //Creating LinkedHashSet            LinkedHashSet<String> set = new LinkedHashSet<String>();            //Adding elements to LinkedHashSet            set.add("JAVA");            set.add("J2EE");            set.add("STRUTS");            set.add("JSP");            set.add("JDBC");            set.add("HIBERNATE");            //Printing elements of LinkedHashSet            System.out.println(set);            //Output : [JAVA, J2EE, STRUTS, JSP, JDBC, HIBERNATE]            //Notice the order of elements. They are placed according to their insertion order.      }  } |

* LinkedhashSet also gives **constant time performance** for insertion, removal and retrieval operations. The performance of LinkedHashSet is slightly less than the Hashset as it has to maintain doubly linked list internally to order it’s elements.
* Iterator returned by LinkedHashSet is **fail-fast**. i.e if the LinkedHashSet is modified at any time after the Iterator is created, it throws ConcurrentModificationException.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | public class LinkedHashSetExample  {      public static void main(String[] args)      {          //Creating LinkedHashSet            LinkedHashSet<String> set = new LinkedHashSet<String>();            //Adding elements to LinkedHashSet            set.add("JAVA");            set.add("J2EE");            set.add("STRUTS");            set.add("JSP");            set.add("JDBC");            set.add("HIBERNATE");            //Getting Iterator object            Iterator<String> it = set.iterator();            //Modifying the LinkedHashSet after the Iterator is created            set.add("JSF");            while (it.hasNext())          {              //This statement will throw ConcurrentModificationException                System.out.println(it.next());          }      }  } |

* LinkedHashSet doesn’t allow **duplicate** elements and allows only one **null** element.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | public class LinkedHashSetExample  {      public static void main(String[] args)      {          //Creating LinkedHashSet            LinkedHashSet<String> set = new LinkedHashSet<String>();            //Adding elements to LinkedHashSet            set.add("BLUE");            set.add("RED");            set.add("GREEN");            set.add("BLUE");     //duplicate element            set.add("BLACK");            set.add("WHITE");            //Adding two null elements            set.add(null);            set.add(null);            //printing the elements of LinkedHashSet            System.out.println(set);     //Output : [BLUE, RED, GREEN, BLACK, WHITE, null]            //You can notice that LinkedHashSet doesn't allow duplicates and allows only one null element      }  } |

* LinkedHashSet is not **synchronized**. To get the synchronized LinkedHashSet, use **Collections.synchronizedSet()** method.

# 10.1 How LinkedHashSet Works Internally In Java?

LinkedHashSet is an **extended version** of HashSet. HashSet doesn’t follow any order where as LinkedHashSet maintains **insertion order**. HashSet uses **HashMap object** internally to store it’s elements where as LinkedHashSet uses **LinkedHashMap object** internally to store and process it’s elements. In this article, we will see how LinkedHashSet works internally and how it maintains insertion order.

Let’s start with constructors of LinkedHashSet class. There are 4 constructors in LinkedHashSet class. All these constructors are simply calling to super class constructor i.e constructor of HashSet class. Below is the how the constructors are defined in LinkedHashSet class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | //Constructor - 1    public LinkedHashSet(int initialCapacity, float loadFactor)  {        super(initialCapacity, loadFactor, true);              //Calling super class constructor  }    //Constructor - 2    public LinkedHashSet(int initialCapacity)  {          super(initialCapacity, .75f, true);             //Calling super class constructor  }    //Constructor - 3    public LinkedHashSet()  {          super(16, .75f, true);                //Calling super class constructor  }    //Constructor - 4    public LinkedHashSet(Collection<? extends E> c)  {          super(Math.max(2\*c.size(), 11), .75f, true);          //Calling super class constructor          addAll(c);  } |

In the above code snippet, you might have noticed that all 4 constructors are calling the same super class constructor. This constructor is a package private constructor which is used only by the LinkedHashSet class. This constructor takes initial capacity, load factor and one boolean dummy value as it’s arguments. This **boolean dummy value** is just used to differentiate this constructor from other constructors of HashSet class which take initial capacity and load factor as their arguments. Here is the how this constructor is defined in HashSet class.

|  |  |
| --- | --- |
| 1  2  3  4 | HashSet(int initialCapacity, float loadFactor, boolean dummy)  {          map = new LinkedHashMap<>(initialCapacity, loadFactor);  } |

As you are seeing, this constructor internally creates one new **LinkedHashMap** object. This LinkedHashMap object is used by the LinkedHashSet to store it’s elements.

LinkedHashSet doesn’t have it’s own methods. All methods are inherited from it’s super class i.e HashSet. So. all operations on LinkedHashSet work in the same manner as that of HashSet. The only change is the internal object used to store the elements. In hashSet, elements you insert are stored as **keys of HashMap** object. Where as in LinkedHashSet, elements you insert are stored as **keys of LinkedHashMap** object. The values of these keys will be the same constant i.e “**PRESENT**“. We have seen this in [How HashSet works internally in Java.](https://javaconceptoftheday.com/how-hashset-works-internally-in-java/)

## **How LinkedHashSet Maintains Insertion Order?**

LinkedHashSet uses LinkedHashMap object to store it’s elements. The elements you insert in the LinkedHashSet are stored as keys of this LinkedHashMap object. Each **key, value pair** in the LinkedHashMap are instances of it’s static inner class called **Entry<K, V>**. This Entry<K, V> class extends **HashMap.Entry** class. The insertion order of elements into LinkedHashMap are maintained by adding two new fields to this class. They are **before** and **after**. These two fields hold the references to previous and next elements. These two fields make LinkedHashMap to function as a doubly linked list.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | private static class Entry<K,V> extends HashMap.Entry<K,V>  {          // These fields comprise the doubly linked list used for iteration.          Entry<K,V> before, after;            Entry(int hash, K key, V value, HashMap.Entry<K,V> next) {              super(hash, key, value, next);          }  } |

The first two fields of above inner class of LinkedHashMap – **before** and **after** are responsible for maintaining the insertion order of the LinkedHashSet. The header field of LinkedHashMap stores the head of this doubly linked list. It is declared like below,

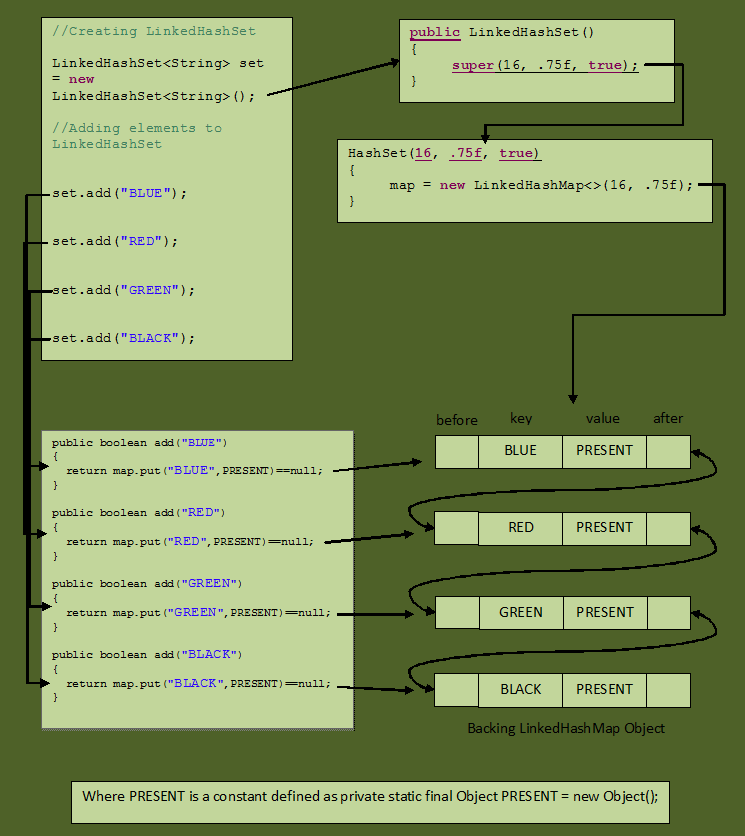
|  |  |
| --- | --- |
| 1 | private transient Entry<K,V> header;        //Stores the head of the doubly linked list |

In LinkedHashMap, the same set of Entry objects (rather references to Entry objects) are arranged in two different manner. One is the HashMap and another one is Doubly linked list. The Entry objects just sit on heap memory, unaware of that they are part of two different data structures.

Let’s see one example of LinkedHashSet to know how it works internally.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class LinkedHashSetExample  {      public static void main(String[] args)      {          //Creating LinkedHashSet            LinkedHashSet<String> set = new LinkedHashSet<String>();            //Adding elements to LinkedHashSet            set.add("BLUE");            set.add("RED");            set.add("GREEN");            set.add("BLACK");      }  } |

Look at the below image to see how above program works.



If you know how LinkedHashMap works internally, it will be easy for you to understand how LinkedHashSet works internally. Go through source code of LinkedHashSet class and LinkedHashMap class once, you will get precise understanding about how LinkedHashSet works internally in Java.

# 10.2 Java LinkedHashSet Example:

As you already know, **LinkedHashSet** is an ordered version of HashSet. That means, HashSet doesn’t maintain any order where as LinkedHashSet maintains insertion order of the elements. LinkedHashSet uses **doubly linked list** internally to maintain the insertion order of it’s elements. We have seen this in [How LinkedHashSet Works Internally In Java?](https://javaconceptoftheday.com/how-linkedhashset-works-internally-in-java/). As LinkedHashSet maintains doubly linked list (along with HashMap), the performance of LinkedHashSet is slightly slower than the HashSet. But, LinkedHashSet will be very useful when you need a collection of elements placed in the order they have inserted. We will see one such example of LinkedHashSet in this article.

Let’s consider that you want to create a pool of customers placed in the order they have arrived. Assume that it is also mandatory that duplicate customers must not be allowed. For such requirements, LinkedHashSet is the best suitable. In this article, we will try to implement this example using LinkedHashSet class.

Let’s create **Customer** class with two fields – **name** and **id**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | class Customer  {      String name;        int id;        public Customer(String name, int id)      {          this.name = name;            this.id = id;      }        @Override      public int hashCode()      {          return id;      }        @Override      public boolean equals(Object obj)      {          Customer customer = (Customer) obj;            return (id == customer.id);      }        @Override      public String toString()      {          return id+" : "+name;      }  } |

You might have observed that **equals()** and **hashCode()** methods in the above class are overrided so that Customer objects will be compared solely based on **id**. That means two Customer objects having same **id** will be considered as duplicates and they will not be allowed in the pool.

Create one LinkedHashSet object containing elements of Customer type.

|  |  |
| --- | --- |
| 1 | LinkedHashSet<Customer> set = new LinkedHashSet<Customer>(); |

Add some elements to this set.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | set.add(new Customer("Jack", 021));    set.add(new Customer("Peter", 105));    set.add(new Customer("Ramesh", 415));    set.add(new Customer("Julian", 814));    set.add(new Customer("Avinash", 105));      //Duplicate Element    set.add(new Customer("Sapna", 879));    set.add(new Customer("John", 546));    set.add(new Customer("Moni", 254));    set.add(new Customer("Ravi", 105));        //Duplicate Element |

Iterate through this LinkedHashSet.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Iterator<Customer> it = set.iterator();    while (it.hasNext())  {      Customer customer = (Customer) it.next();        System.out.println(customer);  } |

Output will be,

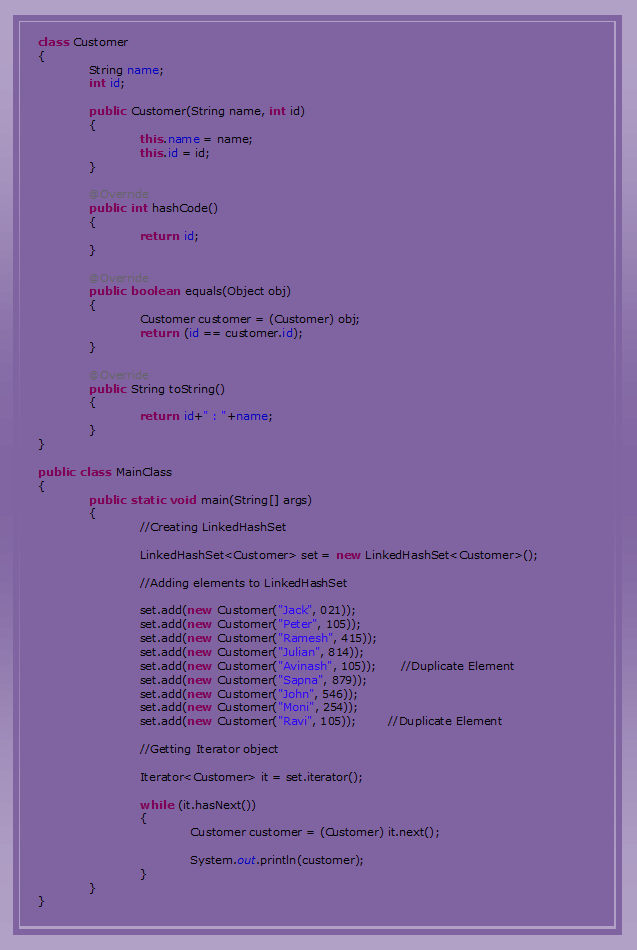
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 17 : Jack  105 : Peter  415 : Ramesh  814 : Julian  879 : Sapna  546 : John  254 : Moni |

You can notice that Customer objects are placed in the order they are inserted into the set and also duplicate elements are avoided.

Below is the code for the whole program.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74 | class Customer  {      String name;        int id;        public Customer(String name, int id)      {          this.name = name;            this.id = id;      }        @Override      public int hashCode()      {          return id;      }        @Override      public boolean equals(Object obj)      {          Customer customer = (Customer) obj;            return (id == customer.id);      }        @Override      public String toString()      {          return id+" : "+name;      }  }    public class MainClass  {      public static void main(String[] args)      {          //Creating LinkedHashSet            LinkedHashSet<Customer> set = new LinkedHashSet<Customer>();            //Adding elements to LinkedHashSet            set.add(new Customer("Jack", 021));            set.add(new Customer("Peter", 105));            set.add(new Customer("Ramesh", 415));            set.add(new Customer("Julian", 814));            set.add(new Customer("Avinash", 105));      //Duplicate Element            set.add(new Customer("Sapna", 879));            set.add(new Customer("John", 546));            set.add(new Customer("Moni", 254));            set.add(new Customer("Ravi", 105));        //Duplicate Element            //Getting Iterator object            Iterator<Customer> it = set.iterator();            while (it.hasNext())          {              Customer customer = (Customer) it.next();                System.out.println(customer);          }      }  } |

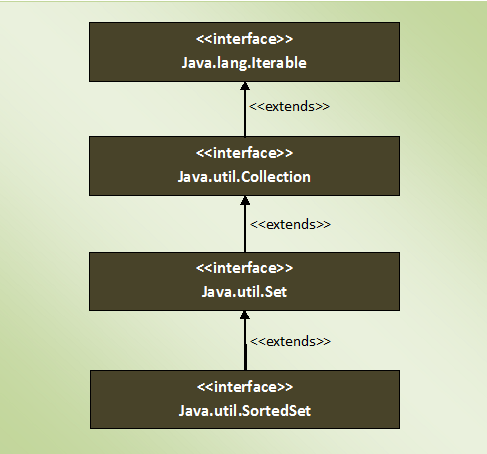
## **Java LinkedHashSet Example :**



# 11. Collection Framework – The SortedSet Interface:

**The SortedSet interface** extends Set interface. SortedSet is a set in which elements are placed according to supplied comparator. This Comparator is supplied while creating a SortedSet. If you don’t supply comparator, elements will be placed in ascending order.

Here is the hierarchy diagram of SortedSet Interface.



## **Methods Of SortedSet Interface :**

SortedSet interface defines 6 more methods along with the inherited methods from Set interface. These methods make the processing of SortedSet elements more easy. Here is the list of SortedSet interface methods.

|  |  |  |
| --- | --- | --- |
| SL NO. | **SortedSet Interface Methods** | **Description** |
| 1 | Comparator<? super E> comparator() | Returns Comparator used to order the elements. If no comparator is supplied, it returns null. |
| 2 | SortedSet<E> subSet(E fromElement, E toElement) | Returns a portion of this set whose elements range from ‘fromElement’ (Inclusive) and ‘toElement’ (Exclusive). |
| 3 | SortedSet<E> headSet(E toElement) | Returns a SortedSet whose elements are in the range from first element of the set (Inclusive) to ‘toElement’ (exclusive). |
| 4 | SortedSet<E> tailSet(E fromElement) | Returns a SortedSet whose elements are in the range from ‘fromElement’ (Inclusive) to last element of the set (exclusive). |
| 5 | E first() | Returns first element of the SortedSet. |
| 6 | E last() | Returns last element of the SortedSet. |

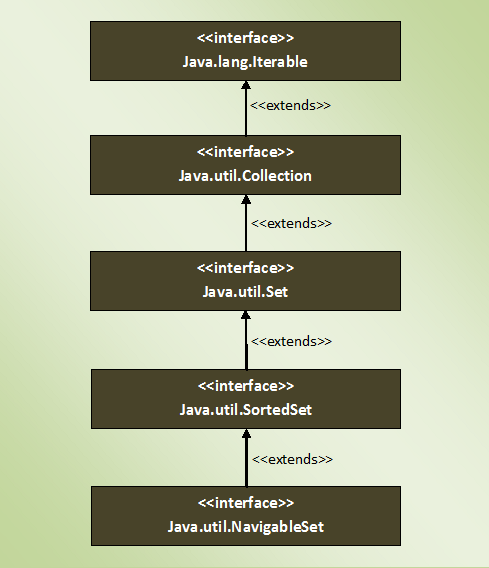
## **Properties Of SortedSet Interface :**

* SortedSet can not have **null** elements. If you try to insert null element, it gives NullPointerException at run time.
* As SortedSet is a set, **duplicate** elements are not allowed.
* SortedSet elements are sorted according to supplied **Comparator**. If you don’t mention any Comparator while creating a SortedSet, elements will be placed in ascending order.
* Inserted elements must be of **Comparable** type and they must be mutually Comparable.
* You can retrieve first element and last elements of the SortedSet. You can’t access SortedSet elements randomly. i.e **Random access** is denied.
* SortedSets returned by **headSet()**, **tailSet()** and **subSet()** methods are just views of the original set. So, changes in the returned set are reflected in the original set and vice versa.

# 12. Collection Framework – The NavigableSet Interface:

The **NavigableSet** is a SortedSet with navigation facilities. The **NavigableSet interface** provides many methods through them you can easily find closest matches of any given element. It has the methods to find out less than, less than or equal to, greater than and greater than or equal of any element in a SortedSet.

The NavigableSet interface extends SortedSet interface. Here is the hierarchy diagram of NavigableSet Interface.



## **Methods Of NavigableSet Interface :**

|  |  |  |
| --- | --- | --- |
| SL NO. | NavigableSet Interface Methods | Description |
| 1 | E lower(E e) | Returns greatest element in this set which is strictly less than the given element. |
| 2 | E floor(E e) | Returns greatest element in this set which is less than or equal to the given element. |
| 3 | E ceiling(E e) | Returns the least element in this set which is greater than or equal to the given element. |
| 4 | E higher(E e) | Returns the least element in this set which is strictly greater than the given element. |
| 5 | E pollFirst() | Retrieves and removes the first element in this set. |
| 6 | E pollLast() | Retrieves and removes last element in this set. |
| 7 | NavigableSet<E> descendingSet() | Returns reverse order view of this set. |
| 8 | Iterator<E> descendingIterator() | Returns an iterator over the elements of this set in descending order. |
| 9 | NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive) | Returns a view of this set whose elements are in the range from ‘fromElement’ to ‘toElement’. |
| 10 | NavigableSet<E> headSet(E toElement, boolean inclusive) | Returns a view of this set whose elements are in the range from first element of this set to ‘toElement’. |
| 11 | NavigableSet<E> tailSet(E fromElement, boolean inclusive) | Returns a view of this element whose elements are in the range from ‘fromElement’ to last element of this set. |

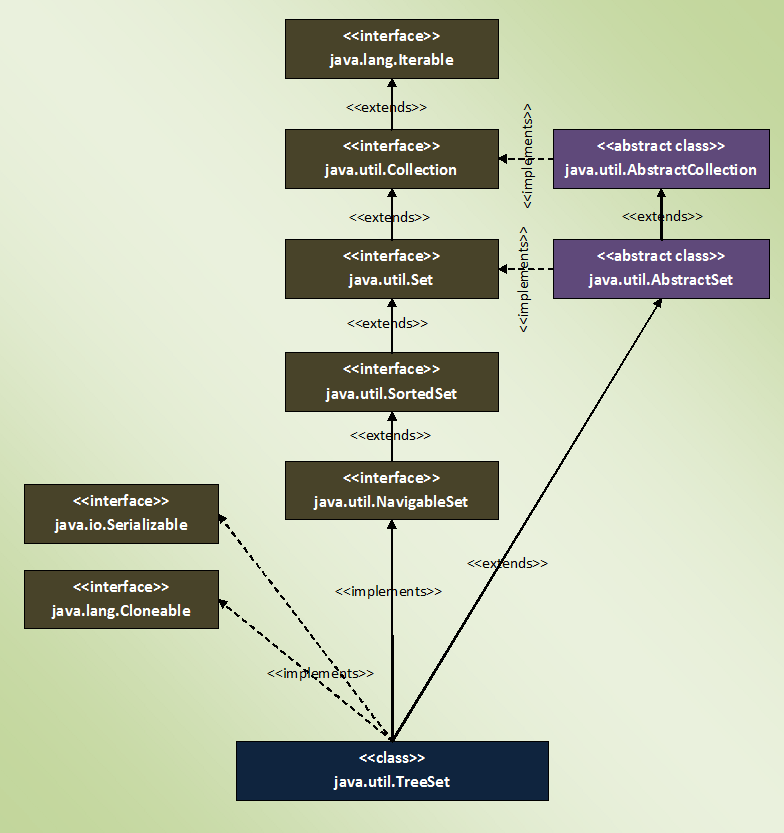
## **Properties Of NavigableSet Interface :**

* NavaigableSet can’t have null elements.
* NavigableSet doesn’t support duplicate elements.
* NavigableSet can be traversed and accessed in either ascending or descending order.
* Methods subSet(), headSet() and tailSet() differ from SortedSet interface in taking additional arguments describing whether upper bound and lower bound are inclusive or exclusive.

# 13. Collection Framework – The TreeSet class:

The **TreeSet** is another popular implementation of Set interface. We have seen other two implementations of Set interface – [HashSet](https://javaconceptoftheday.com/java-collection-framework-hashset-class/) and [LinkedHashSet](https://javaconceptoftheday.com/java-collection-framework-linkedhashset-class/). HashSet doesn’t maintain any order where as LinkedHashSet maintains insertion order. The main difference between these two implementations and Treeset is, elements in TreeSet are **sorted** according to supplied **Comparator**. You need to supply this Comparator while creating a TreeSet itself. If you don’t pass any Comparator while creating a TreeSet, elements will be placed in their **natural ascending order**.

**The TreeSet class** in java is a direct implementation of **NavigableSet** interface which in turn extends **SortedSet** interface (which in turn extends Set interface). Below is the hierarchy diagram of TreeSet class.



## **Properties Of TreeSet Class In Java :**

* The elements in TreeSet are sorted according to specified **Comparator**. If no Comparator is specified, elements will be placed according to their natural ascending order.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public class TreeSetExample  {      public static void main(String[] args)      {          //Creating a TreeSet            TreeSet<Integer> set = new TreeSet<Integer>();            //Adding elements to TreeSet            set.add(20);            set.add(10);            set.add(40);            set.add(80);            set.add(30);            //Printing elements of TreeSet            System.out.println(set);      //Output : [10, 20, 30, 40, 80]            //Notice that elements are placed in the sorted order.      }  } |

* Elements inserted in the TreeSet must be of **Comparable** type and elements must be mutually comparable. If the elements are not mutually comparable, you will get **ClassCastException** at run time.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class TreeSetExample  {      public static void main(String[] args)      {          //Creating a TreeSet            TreeSet<Object> set = new TreeSet<Object>();            //Adding elements to TreeSet            set.add("kkk");      //inserting String type element            set.add(10);        //inserting Integer type element            set.add(new Object());      //inserting Object type element            set.add(20.65);     //inserting Double type element            //The elements inserted are not mutually comparable. So, it will throw ClassCastException.      }  } |

* TreeSet does not allow even a single **null** element.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class TreeSetExample  {      public static void main(String[] args)      {          //Creating a TreeSet            TreeSet<String> set = new TreeSet<String>();            //Adding elements to TreeSet            set.add("aaa");            set.add(null);    //It will throw NullPointerException            set.add("ccc");            set.add("ddd");      }  } |

* TreeSet is not **synchronized**. To get a synchronized TreeSet, use **Collections.synchronizedSortedSet()** method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class TreeSetExample  {      public static void main(String[] args)      {          //Creating a TreeSet            TreeSet<String> treeSet = new TreeSet<String>();            //Getting a synchronized TreeSet            Set<String> set = Collections.synchronizedSortedSet(treeSet);      }  } |

* TreeSet gives performance of order **log(n)** for insertion, removal and retrieval operations.
* Iterator returned by TreeSet is of **fail-fast** nature. That means, If TreeSet is modified after the creation of Iterator object, you will get **ConcurrentModificationException**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | public class TreeSetExample  {      public static void main(String[] args)      {          //Creating a TreeSet            TreeSet<String> set = new TreeSet<String>();            //Adding elements to TreeSet            set.add("aaa");            set.add("bbb");            set.add("ccc");            set.add("ddd");            //Getting Iterator object            Iterator<String> it = set.iterator();            //Modifying the TreeSet after getting Iterator object            set.add("eee");            while (it.hasNext())          {              //This statement will throw ConcurrentModificationException                System.out.println(it.next());          }      }  } |

* TreeSet internally uses **TreeMap** to store it’s elements just like HashSet and LinkedHashSet which use HashMap and LinkedHashMap respectively to store their elements.

# 14.1 Java TreeSet Example:

**TreeSet** is another popular implementation of Set interface along with **HashSet** and **LinkedHashSet**. All these implementations of Set interface are required in different scenarios. If you don’t want any order of elements, then you can use HashSet. If you want insertion order of elements to be maintained, then use LinkedHashSet. If you want elements to be ordered according to some Comparator, then use TreeSet. The common thing of these three implementations is that they don’t allow duplicate elements.

In this article, I have tried to explain two examples of Java TreeSet. One example doesn’t use Comparator and another example uses Comparator to order the elements. You can go through some basic properties of TreeSet class [here](https://javaconceptoftheday.com/java-collection-framework-treeset-class/).

## **Java TreeSet Example With No Comparator :**

You already know that if you don’t pass any comparator while creating a TreeSet, elements will be placed in their natural ascending order. In this example, we create a TreeSet of Integers without supplying any Comparator like this,

|  |  |
| --- | --- |
| 1 | TreeSet<Integer> set = new TreeSet<Integer>(); |

Let’s add some integer elements to it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | set.add(23);    set.add(11);    set.add(41);    set.add(7);    set.add(69);    set.add(18);    set.add(38); |

Print these elements and observe the output.

|  |  |
| --- | --- |
| 1 | System.out.println(set);      //Output : [7, 11, 18, 23, 38, 41, 69] |

You can notice that elements are placed in the ascending order.

The whole code for this example is,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class TreeSetExample  {      public static void main(String[] args)      {          //Creating a TreeSet without supplying any Comparator            TreeSet<Integer> set = new TreeSet<Integer>();            //Adding elements to TreeSet            set.add(23);            set.add(11);            set.add(41);            set.add(7);            set.add(69);            set.add(18);            set.add(38);            //printing elements of TreeSet            System.out.println(set);      //Output : [7, 11, 18, 23, 38, 41, 69]      }  } |

## **Java TreeSet Example With Comparator :**

In this example, we create one TreeSet by supplying a customized Comparator. In this example, we will try to create a TreeSet of **Student** objects ordered in the descending order of the percentage of marks they have obtained. That means, student with highest marks will be placed at the top.

Let’s create **‘Student’** class with three fields – **id**, **name** and **perc\_Of\_Marks\_Obtained**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class Student  {      int id;        String name;        int perc\_Of\_Marks\_Obtained;        public Student(int id, String name, int perc\_Of\_Marks\_Obtained)      {          this.id = id;            this.name = name;            this.perc\_Of\_Marks\_Obtained = perc\_Of\_Marks\_Obtained;      }        @Override      public String toString()      {          return id+" : "+name+" : "+perc\_Of\_Marks\_Obtained;      }  } |

Let’s define our own Comparator class **“MyComparator”** which compares the marks of two students.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class MyComparator implements Comparator<Student>  {      @Override      public int compare(Student s1, Student s2)      {          if(s1.id == s2.id)          {              return 0;          }          else          {              return s2.perc\_Of\_Marks\_Obtained - s1.perc\_Of\_Marks\_Obtained;          }      }  } |

**Important Note :** TreeSet doesn’t use hashCode() and equals() methods to compare it’s elements. It uses compare() (or compareTo()) method to determine the equality of two elements. Therefore, I have kept the code which compares two Student objects based on their id in compare method itself. This removes possible duplicate elements (elements having same id) from the TreeSet.

Create one TreeSet of ‘**Student**‘ objects with ‘**MyComparator**‘ as a Comparator.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //Instantiating MyComparator    MyComparator comparator = new MyComparator();    //Creating TreeSet with 'MyComparator' as Comparator.    TreeSet<Student> set = new TreeSet<Student>(comparator); |

Add some elements of type ‘**Student**‘ to this TreeSet.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | set.add(new Student(121, "Santosh", 85));    set.add(new Student(231, "Cherry", 71));    set.add(new Student(417, "David", 82));    set.add(new Student(562, "Praveen", 91));    set.add(new Student(231, "Raj", 61));         //Duplicate element    set.add(new Student(458, "John", 76));    set.add(new Student(874, "Peter", 83));    set.add(new Student(231, "Hari", 52));       //Duplicate element    set.add(new Student(568, "Daniel", 89)); |

Iterate through the TreeSet.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //Iterating through TreeSet    Iterator<Student> it = set.iterator();    while (it.hasNext())  {      System.out.println(it.next());  } |

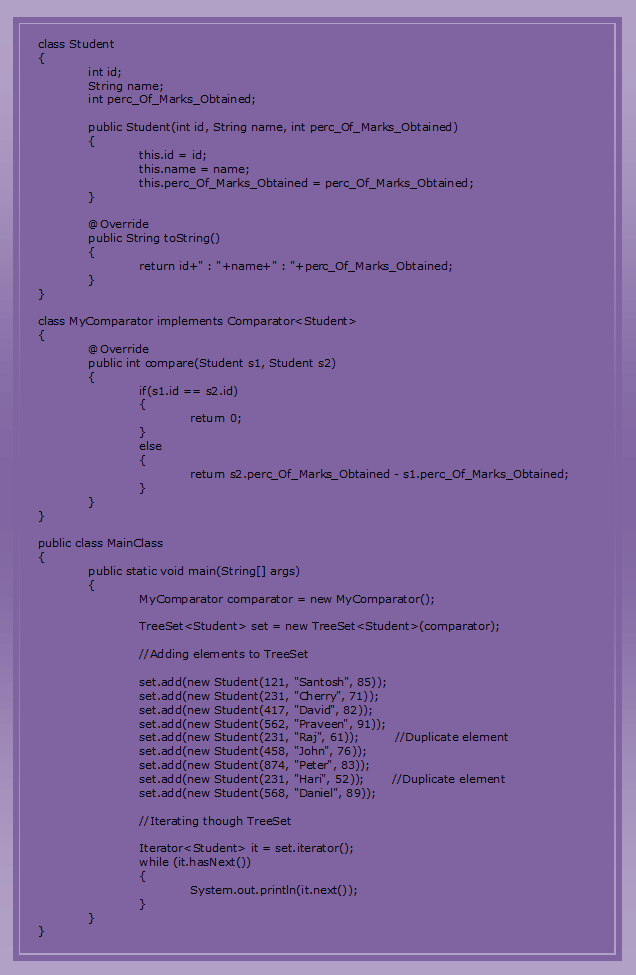
Output will be,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 562 : Praveen : 91  568 : Daniel : 89  121 : Santosh : 85  874 : Peter : 83  417 : David : 82  458 : John : 76  231 : Cherry : 71 |

You can notice that student with highest percentage of marks is placed at the top and also duplicate elements are not allowed in the TreeSet.

Below is the whole code of the above example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88 | // Student Class    class Student  {      int id;        String name;        int perc\_Of\_Marks\_Obtained;        public Student(int id, String name, int perc\_Of\_Marks\_Obtained)      {          this.id = id;            this.name = name;            this.perc\_Of\_Marks\_Obtained = perc\_Of\_Marks\_Obtained;      }        @Override      public String toString()      {          return id+" : "+name+" : "+perc\_Of\_Marks\_Obtained;      }  }    //MyComparator Class    class MyComparator implements Comparator<Student>  {      @Override      public int compare(Student s1, Student s2)      {          if(s1.id == s2.id)          {              return 0;          }          else          {              return s2.perc\_Of\_Marks\_Obtained - s1.perc\_Of\_Marks\_Obtained;          }      }  }    //MainClass    public class MainClass  {      public static void main(String[] args)      {          //Instantiating MyComparator            MyComparator comparator = new MyComparator();            //Creating TreeSet with 'MyComparator' as Comparator.            TreeSet<Student> set = new TreeSet<Student>(comparator);            //Adding elements to TreeSet            set.add(new Student(121, "Santosh", 85));            set.add(new Student(231, "Cherry", 71));            set.add(new Student(417, "David", 82));            set.add(new Student(562, "Praveen", 91));            set.add(new Student(231, "Raj", 61));         //Duplicate element            set.add(new Student(458, "John", 76));            set.add(new Student(874, "Peter", 83));            set.add(new Student(231, "Hari", 52));       //Duplicate element            set.add(new Student(568, "Daniel", 89));            //Iterating though TreeSet            Iterator<Student> it = set.iterator();            while (it.hasNext())          {              System.out.println(it.next());          }      }  } |



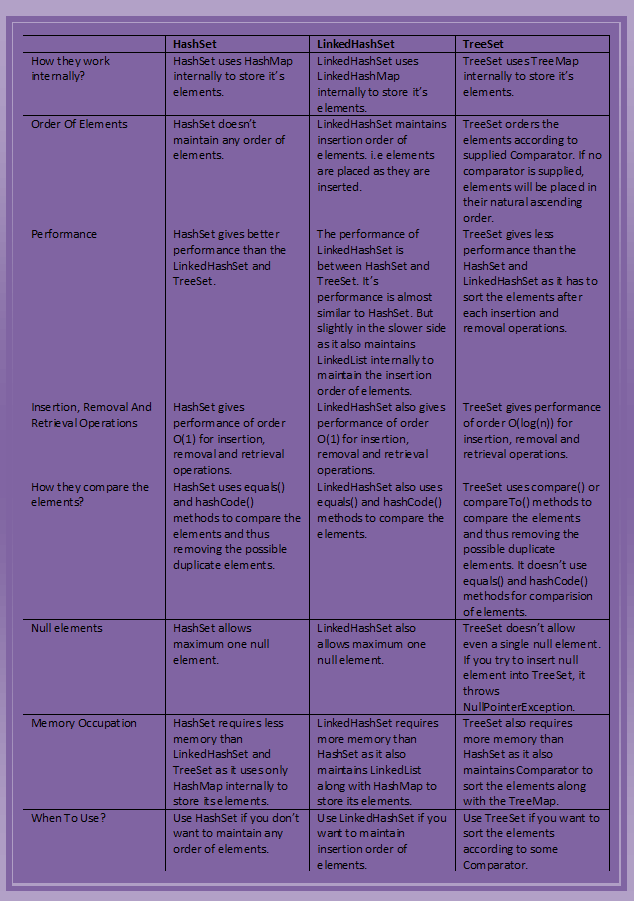
# 14.2 HashSet Vs LinkedHashSet Vs TreeSet In Java:

## **HashSet Vs TreeSet Vs LinkedHashSet In Java :**

Even though, **HashSet**, **LinkedHashSet** and **TreeSet** are all implementations of Set interface, there are some differences exist between them. In this article, I have tried to list out the differences between HashSet, LinkedHashSet and TreeSet in java. They also have some similarities between them. We will also discuss them at the end.

## **Differences Between HashSet, LinkedHashSet and TreeSet In Java :**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **HashSet** | **LinkedHashSet** | **TreeSet** |
| How they work internally? | HashSet uses HashMap internally to store it’s elements. | LinkedHashSet uses  LinkedHashMap internally to store it’s elements. | TreeSet uses TreeMap internally to store it’s elements. |
| Order Of Elements | HashSet doesn’t maintain any order of elements. | LinkedHashSet maintains insertion order of elements. i.e elements are placed as they are inserted. | TreeSet orders the elements according to supplied Comparator. If no comparator is supplied, elements will be placed in their natural ascending order. |
| Performance | HashSet gives better performance than the LinkedHashSet and TreeSet. | The performance of LinkedHashSet is between HashSet and TreeSet. It’s performance is almost similar to HashSet. But slightly in the slower side as it also maintains LinkedList internally to maintain the insertion order of elements. | TreeSet gives less performance than the HashSet and LinkedHashSet as it has to sort the elements after each insertion and removal operations. |
| Insertion, Removal And Retrieval Operations | HashSet gives performance of order O(1) for insertion, removal and retrieval operations. | LinkedHashSet also gives performance of order O(1) for insertion, removal and retrieval operations. | TreeSet gives performance of order O(log(n)) for insertion, removal and retrieval operations. |
| How they compare the elements? | HashSet uses equals() and hashCode() methods to compare the elements and thus removing the possible duplicate elements. | LinkedHashSet also uses equals() and hashCode() methods to compare the elements. | TreeSet uses compare() or compareTo() methods to compare the elements and thus removing the possible duplicate elements. It doesn’t use equals() and hashCode() methods for comparision of elements. |
| Null elements | HashSet allows maximum one null element. | LinkedHashSet also allows maximum one null element. | TreeSet doesn’t allow even a single null element. If you try to insert null element into TreeSet, it throws NullPointerException. |
| Memory Occupation | HashSet requires less memory than LinkedHashSet and TreeSet as it uses only HashMap internally to store its elements. | LinkedHashSet requires more memory than HashSet as it also maintains LinkedList along with HashMap to store its elements. | TreeSet also requires more memory than HashSet as it also maintains Comparator to sort the elements along with the TreeMap. |
| When To Use? | Use HashSet if you don’t want to maintain any order of elements. | Use LinkedHashSet if you want to maintain insertion order of elements. | Use TreeSet if you want to sort the elements according to some Comparator. |



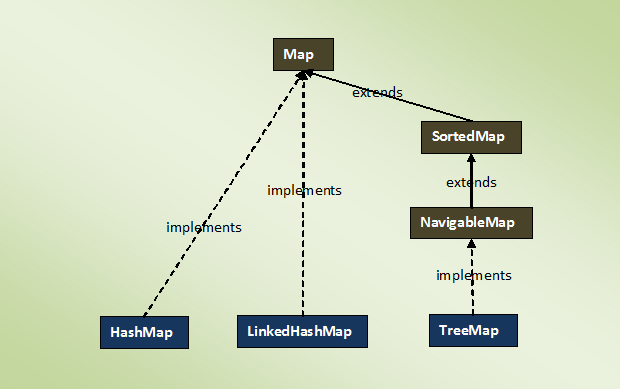
## **Similarities Between HashSet, LinkedHashSet and TreeSet In Java :**

* All three doesn’t allow duplicate elements.
* All three are not synchronized.
* All three are Cloneable and Serializable.
* Iterator returned by all three is fail-fast in nature. i.e You will get ConcurrentModificationException if they are modified after the creation of Iterator object.

# 15. Collection Framework – The Map Interface:

The **Map interface** in java is one of the four top level interfaces of Java Collection Framework along with [List](https://javaconceptoftheday.com/collection-framework-list-interface/), [Set](https://javaconceptoftheday.com/collection-framework-set-interface/) and [Queue](https://javaconceptoftheday.com/collection-framework-queue-interface/) interfaces. But, unlike others, it doesn’t inherit from [Collection](https://javaconceptoftheday.com/collection-framework-collection-interface/) interface. Instead it starts it’s own interface hierarchy for maintaining the **key-value associations**. Map is an object of key-value pairs where each key is associated with a value. This interface is the replacement for ‘**Dictionary**‘ class which is an abstract class introduced in JDK 1.0.

**HashMap**, **LinkedHashMap** and **TreeMap** are three popular implementations of Map interface. Below picture shows the hierarchy of Map interface in java.



## **Properties Of Map Interface In Java :**

1) Map interface is a part of Java Collection Framework, but it doesn’t inherit **Collection Interface**.

2) Map interface stores the data as a **key-value pairs** where each key is associated with a value.

3) A map can not have duplicate **keys** but can have duplicate **values**.

4) Each key **at most** must be associated with one value.

5) Each key-value pairs of the map are stored as **Map.Entry** objects. Map.Entry is an inner interface of Map interface.

6) The common implementations of Map interface are **HashMap**, **LinkedHashMap** and **TreeMap**.

7) Order of elements in map is implementation dependent. **HashMap** doesn’t maintain any order of elements. **LinkedHashMap** maintains **insertion order** of elements. Where as **TreeMap** places the elements according to supplied **Comparator**.

8) The Map interface provides three methods, which allows map’s contents to be viewed as a **set of keys**(keySet() method), **collection of values**(values() method), or **set of key-value mappings**(entrySet() method).

## **Methods Of Map Interface In Java :**

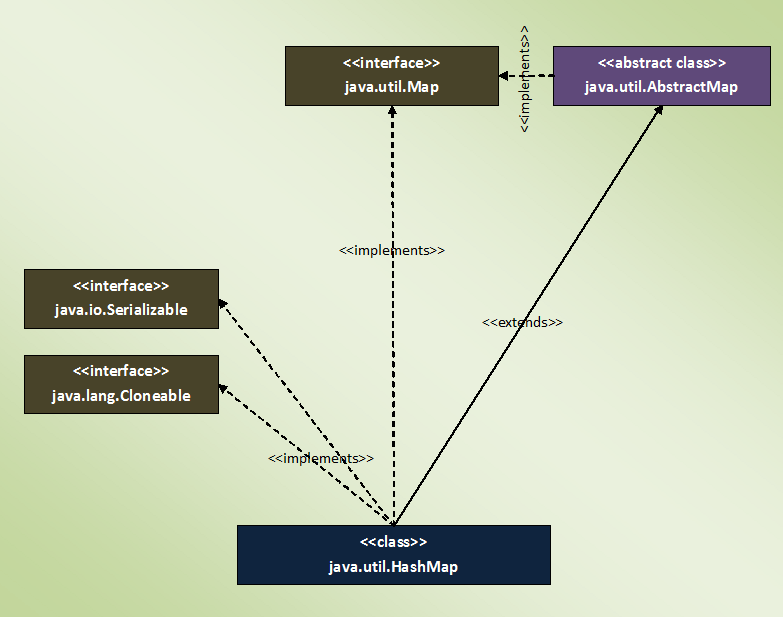
|  |  |  |
| --- | --- | --- |
| SL NO. | Methods | Descriptions |
| 1 | int size() | Returns number of key-value pairs in this map. |
| 2 | boolean isEmpty() | Checks whether this map is empty or not. |
| 3 | boolean containsKey(Object key) | Returns true if this map contains a mapping for the specified key. |
| 4 | boolean containsValue(Object value) | Returns true if this map contains one or more keys associated with the specified value. |
| 5 | V get(Object key) | Returns value associated with the specified key. |
| 6 | V put(K key, V value) | Adds the specified key-value pair to this map. If the specified key already exist in the map, old value will be replaced by the specified value. |
| 7 | V remove(Object key) | Removes the specified key along with it’s value from this map. |
| 8 | void putAll(Map<? extends K, ? extends V> m) | Copies all key-value pairs from the specified map to this map. |
| 9 | void clear() | Removes all mappings from this map. |
| 10 | Set<K> keySet() | Returns a set containing all keys of this map. The returned set is backed by actual map. So, changes made to the map are reflected in the set and vice-versa. |
| 11 | Collection<V> values() | Returns a collection of values of this map. The returned collection is backed by actual map. So, any changes made to the map is reflected in collection and vice-versa. |
| 12 | Set<Map.Entry<K, V>> entrySet() | Returns set view of the mappings contained in this map. |
| 13 | boolean equals(Object o) | Compares the specified object with this map. |
| 14 | int hashCode() | Returns hashcode value of this map. |

# 15.1 HashMap In Java With Example:

The java.util.HashMap is a popular implementation of Map interface which holds the data as key-value pairs. HashMap extends AbstractMap class and implements Cloneable and Serializable interfaces. In this article, we will discuss about hierarchy of HashMap, properties of HashMap and some important methods of HashMap in java.

### **Hierarchy Of HashMap In Java :**

As already said, HashMap extends AbstractMap class and implements Cloneable and Serializable interfaces. AbstractMap is an abstract class which provides skeletal implementation of Map interface. Below is the hierarchy structure of java.util.HashMap class.



### **Properties Of HashMap In Java :**

1) HashMap holds the data in the form of key-value pairs where each key is associated with one value.

2) HashMap doesn’t allow duplicate keys. But it can have duplicate values.

3) HashMap can have multiple null values and only one null key.

4) HashMap is not synchronized. To get the synchronized HashMap, use Collections.synchronizedMap() method.

5) HashMap maintains no order.

6) HashMap gives constant time performance for the operations like get() and put() methods.

7) Default initial capacity of HashMap is 16.

### **Important Methods Of HashMap In Java :**

1) public V put(K key, V value)

This method inserts specified key-value mapping in the map. If map already has a mapping for the specified key, then it rewrites that value with new value.

2) public void putAll(Map m)

This method copies all of the mappings of the map m to this map.

3) public V get(Object key)

This method returns the value associated with a specified key.

4) public int size()

This method returns the number of key-value pairs in this map.

5) public boolean isEmpty()

This method checks whether this map is empty or not.

6) public boolean containsKey(Object key)

This method checks whether this map contains the mapping for the specified key.

7) public boolean containsValue(Object value)

This method checks whether this map has one or more keys mapping to the specified value.

8) public V remove(Object key)

This method removes the mapping for the specified key.

9) public void clear()

This method removes all the mappings from this map.

10) public Set<K> keySet()

This method returns the Set view of the keys in the map.

11) public Collection<V> values()

This method returns Collection view of the values in the map.

12) public Set<Map.Entry<K, V>> entrySet()

This method returns the Set view of all the mappings in this map.

13) public V putIfAbsent(K key, V value)

This method maps the given value with specified key if this key is currently not associated with a value or mapped to a null.

13) public boolean remove(Object key, Object value)

This method removes the entry for the specified key if this key is currently mapped to a specified value.

14) public boolean replace(K key, V oldValue, V newValue)

This method replaces the oldValue of the specified key with newValue if the key is currently mapped to oldValue.

15) public V replace(K key, V value)

This method replaces the current value of the specified key with new value.

(Reference : [HashMap Java 8 Documentation](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html))

### **Java HashMap Example :**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66 | import java.util.HashMap;  import java.util.Iterator;  import java.util.Set;    public class JavaHashMapExample  {      public static void main(String[] args)      {          //Defining the HashMap            HashMap<String, Double> map = new HashMap<String, Double>();            //Adding some elements to HashMap            map.put("Ashwin", 87.55);            map.put("Bharat", 95.65);            map.put("Chetan", 68.13);            map.put("Dhanjay", 74.23);            map.put("Kartik", 65.42);            //HashMap can have one null key and multiple null values            map.put(null, null);            map.put("Sandesh", null);            //Getting the size of the map            System.out.println("Size Of The Map : "+map.size());            System.out.println("-----------------");            //Displaying the elements            System.out.println("The elements are :");            Set set = map.keySet();            Iterator keySetIterator = set.iterator();            while (keySetIterator.hasNext())          {              Object key = keySetIterator.next();                System.out.println(key+"  : "+map.get(key));          }            System.out.println("-----------------");            //Checking the map for a particular key/value            System.out.println("Does this map has Chetan as key? "+map.containsKey("Chetan"));            System.out.println("Does this map has 74.23 as value? "+map.containsValue(74.23));            System.out.println("-----------------");            //Removing an element from the map            System.out.println("Value removed from the map : "+map.remove("Kartik"));      }  } |

**Output :**

Size Of The Map : 7  
—————–  
The elements are :  
null : null  
Ashwin : 87.55  
Dhanjay : 74.23  
Chetan : 68.13  
Bharat : 95.65  
Kartik : 65.42  
Sandesh : null  
—————–  
Does this map has Chetan as key? true  
Does this map has 74.23 as value? true  
—————–  
Value removed from the map : 65.42

# 15.2 How HashMap Works Internally In Java?

If the data is the most important part of an application, then data structure chosen to handle that data is even more important. Because, data structure arranges the data so that insertion of new elements or searching of old elements will be faster. Java provides wide range of data structures to handle the data. You can choose array, list, queue, set or map. Each of these are used in different scenarios. It is up to you to select which one is better for your application.

If your application demands faster insertion and faster retrieval then HashMap is the ultimate choice. While selecting the data structure, you must keep two things in your mind. First one is that the data structure must give better performance while inserting the new elements and second one is that it should give even more better performance while searching for an element. Because insertion and retrieval are two operations which you perform very frequently in your applications. These things will matter even more when you are handling the big data. HashMap is the most sought after data structure when you are handling the big data with more preference to insertion and retrieval operations.

HashMap is the most used data structure in java because it gives almost constant time performance of O(1) for put and get operations irrespective of how big is the data. As you already know, HashMap stores the data in the form of key-value pairs. In this post, we will see how HashMap works internally in java and how it stores the elements to give O(1) performance for put and get operations.

### **HashMap Internal Structure :**

HashMap stores the data in the form of key-value pairs. Each key-value pair is stored in an object of Entry<K, V> class. Entry<K, V> class is the static inner class of HashMap which is defined like below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | static class Entry<K,V> implements Map.Entry<K,V>  {          final K key;          V value;          Entry<K,V> next;          int hash;            //Some methods are defined here  } |

As you see, this inner class has four fields. key, value, next and hash.

**key** : It stores the key of an element and its final.

**value** : It holds the value of an element.

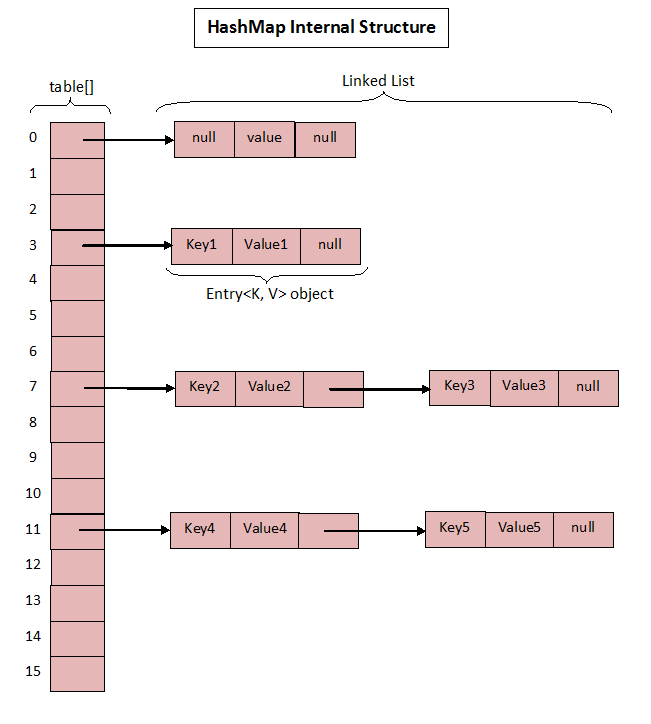
**next** : It holds the pointer to next key-value pair. **This attribute makes the key-value pairs stored as a linked list.**

**hash** : It holds the hashcode of the key.

These Entry objects are stored in an array called table[]. This array is initially of size 16. It is defined like below.

|  |  |
| --- | --- |
| 1  2  3  4 | /\*\*       \* The table, resized as necessary. Length MUST Always be a power of two.       \*/      transient Entry<K,V>[] table; |

To summarize the whole HashMap structure, each key-value pair is stored in an object of Entry<K, V> class. This class has an attribute called next which holds the pointer to next key-value pair. This makes the key-value pairs stored as a linked list. All these Entry<K, V> objects are stored in an array called table[]. The below image best describes the HashMap structure.



The above image roughly shows how the HashMap stores its elements. Internally it uses an array of Entry<K, V> class called table[] to store the key-value pairs. But how HashMap allocates slot in table[] array to each of its key-value pair is very interesting. It doesn’t inserts the objects as you put them into HashMap i.e first element at index 0, second element at index 1 and so on. Instead it uses the hashcode of the key to decide the index for a particular key-value pair. It is called **Hashing**.

### **What Is Hashing?**

The whole HashMap data structure is based on the principle of **Hashing**. Hashing is nothing but the function or algorithm or method which when applied on any object/variable returns an unique integer value representing that object/variable. This unique integer value is called **hash code**. Hash function or simply hash said to be the best if it returns the same hash code each time it is called on the same object. Two objects can have same hash code.

Whenever you insert new key-value pair using put() method, HashMap blindly doesn’t allocate slot in the table[] array. Instead it calls hash function on the key. HashMap has its own hash function to calculate the hash code of the key. This function is implemented so that it overcomes poorly implemented hashCode() methods. Below is implementation code of hash().

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | /\*\*       \* Retrieve object hash code and applies a supplemental hash function to the       \* result hash, which defends against poor quality hash functions.  This is       \* critical because HashMap uses power-of-two length hash tables, that       \* otherwise encounter collisions for hashCodes that do not differ       \* in lower bits. Note: Null keys always map to hash 0, thus index 0.       \*/      final int hash(Object k) {          int h = 0;          if (useAltHashing) {              if (k instanceof String) {                  return sun.misc.Hashing.stringHash32((String) k);              }              h = hashSeed;          }            h ^= k.hashCode();            // This function ensures that hashCodes that differ only by          // constant multiples at each bit position have a bounded          // number of collisions (approximately 8 at default load factor).          h ^= (h >>> 20) ^ (h >>> 12);          return h ^ (h >>> 7) ^ (h >>> 4);      } |

After calculating the hash code of the key, it calls indexFor() method by passing the hash code of the key and length of the table[] array. This method returns the index in the table[] array for that particular key-value pair.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | /\*\*       \* Returns index for hash code h.       \*/      static int indexFor(int h, int length) {          return h & (length-1);      } |

Now, let’s see how put() method works in detail.

### **How put() method works?**

Below is the code implementation of put() method in the HashMap class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | /\*\*       \* Associates the specified value with the specified key in this map.       \* If the map previously contained a mapping for the key, the old       \* value is replaced.       \*       \* @param key key with which the specified value is to be associated       \* @param value value to be associated with the specified key       \* @return the previous value associated with <tt>key</tt>, or       \*         <tt>null</tt> if there was no mapping for <tt>key</tt>.       \*         (A <tt>null</tt> return can also indicate that the map       \*         previously associated <tt>null</tt> with <tt>key</tt>.)       \*/      public V put(K key, V value) {          if (key == null)              return putForNullKey(value);          int hash = hash(key);          int i = indexFor(hash, table.length);          for (Entry<K,V> e = table[i]; e != null; e = e.next) {              Object k;              if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {                  V oldValue = e.value;                  e.value = value;                  e.recordAccess(this);                  return oldValue;              }          }            modCount++;          addEntry(hash, key, value, i);          return null;      } |

Let’s see how this code works step by step.

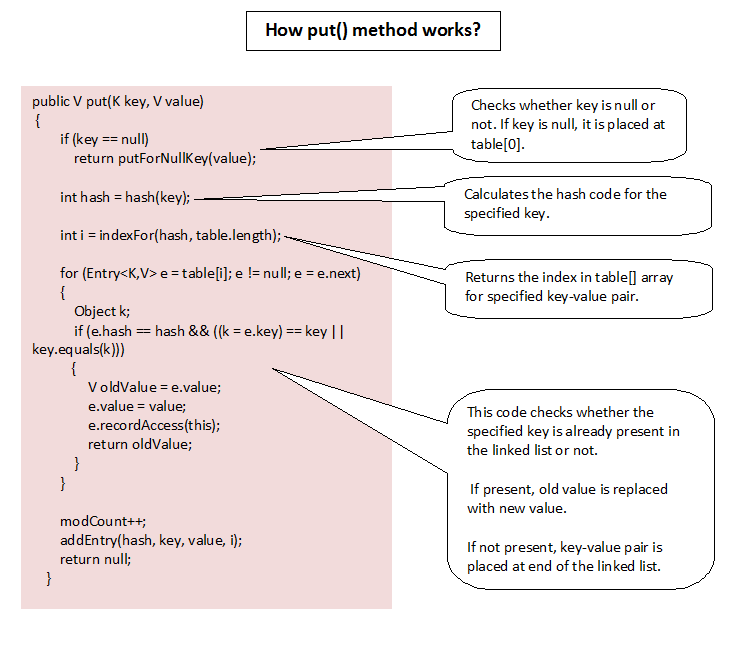
Step 1 : First checks whether the key is null or not. If the key is null, it calls putForNullKey() method. table[0] is always reserved for null key. Because, hash code of null is 0.

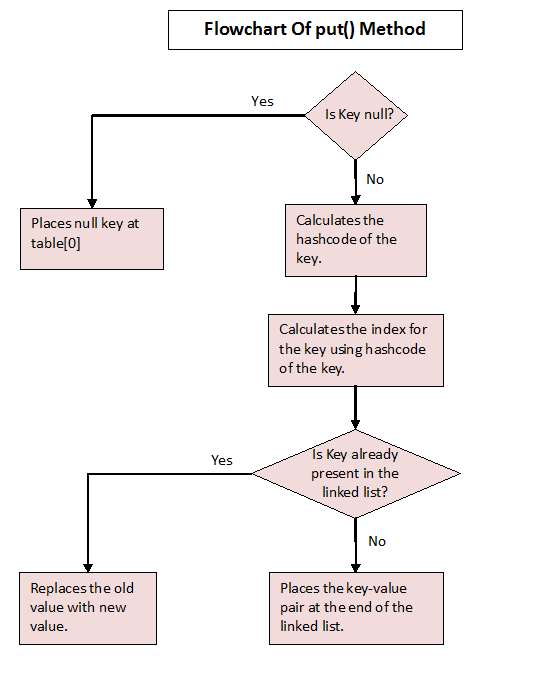
Step 2 : If the key is not null, then it calculates the hash code of the key by calling hash() method.

Step 3 : Calls indexFor() method by passing the hash code calculated in step 2 and length of the table[] array. This method returns index in table[] array for the specified key-value pair.

Step 4 : After getting the index, it checks all keys present in the linked list at that index ( or bucket). If the key is already present in the linked list, it replaces the old value with new value.

Step 5 : If the key is not present in the linked list, it appends the specified key-value pair at the end of the linked list.





### **How get() method Works?**

Let’s see how get() method has implemented.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | /\*\*  \* Returns the value to which the specified key is mapped, or {@code null}  \* if this map contains no mapping for the key.  \*  \*    \* More formally, if this map contains a mapping from a key {@code k} to a  \* value {@code v} such that {@code (key==null ? k==null :  \* key.equals(k))}, then this method returns {@code v}; otherwise it returns  \* {@code null}. (There can be at most one such mapping.)  \*  \*    \* A return value of {@code null} does not <i>necessarily</i> indicate that  \* the map contains no mapping for the key; it's also possible that the map  \* explicitly maps the key to {@code null}. The {@link #containsKey  \* containsKey} operation may be used to distinguish these two cases.  \*  \* @see #put(Object, Object)  \*/  public V get(Object key) {      if (key == null)      return getForNullKey();      int hash = hash(key.hashCode());      for (Entry<K , V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) {          Object k;          if (e.hash == hash && ((k = e.key) == key || key.equals(k)))              return e.value;      }      return null;  } |

Step 1 : First checks whether specified key is null or not. If the key is null, it calls getForNullKey() method.

Step 2 : If the key is not null, hash code of the specified key is calculated.

Step 3 : indexFor() method is used to find out the index of the specified key in the table[] array.

Step 4 : After getting index, it will iterate though linked list at that position and checks for the key using equals() method. If the key is found, it returns the value associated with it. otherwise returns null.

# 15.3 What Are Initial Capacity And Load Factor Of HashMap In Java?

HashMap is one of the high performing data structure in java collection framework. Whatever may be the size of the data, HashMap almost gives constant time performance for most frequent operations – insertion and retrieval. That’s why HashMap is the first choice for the big sized data having requirement of faster retrieval and faster insertion operations. There are two factors which affect the performance of HashMap. One is the **load factor** and another one is **initial capacity**. You have to choose these two factors very carefully while constructing an HashMap object. In this post, we will have a look at initial capacity and load factor in HashMap and see how they affect the performance of HashMap.

### **Initial Capacity Of HashMap :**

The capacity of an HashMap is the number of buckets in the hash table. The initial capacity is the capacity of an HashMap at the time of its creation. The default initial capacity of the HashMap is 24 i.e 16. The capacity of the HashMap is doubled each time it reaches the threshold. i.e the capacity is increased to 25=32, 26=64, 27=128….. when the threshold is reached.

**Also Read :** [How HashMap Works Internally In Java?](https://javaconceptoftheday.com/how-hashmap-works-internally-in-java/)

### **Load Factor Of HashMap :**

Load factor is the measure which decides when to increase the capacity of the HashMap. The default load factor is 0.75f.

### **How The Threshold Is Calculated?**

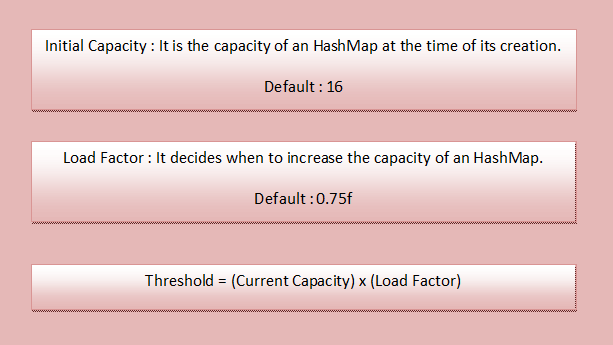
The threshold of an HashMap is the product of current capacity and load factor.

**Threshold = (Current Capacity) \* (Load Factor)**

For example, if the HashMap is created with initial capacity of 16 and load factor of 0.75f, then threshold will be,

Threshold = 16 \* 0.75 = 12

That means, the capacity of the HashMap is increased from 16 to 32 after the 12th element (key-value pair) is added into the HashMap.



### **How Initial Capacity And Load Factor Affect Performance Of HashMap?**

Whenever HashMap reaches its threshold, **rehashing** takes place. Rehashing is a process where new HashMap object with new capacity is created and all old elements (key-value pairs) are placed into new object after recalculating their hashcode. This process of rehashing is both space and time consuming. So, you must choose the initial capacity, by keeping the number of expected elements (key-value pairs) in mind, so that rehashing process doesn’t occur too frequently.

You also have to be very careful while choosing the load factor. According to HashMap doc, the default load factor of 0.75f always gives best performance in terms of both space and time. For example,

If you choose load factor as 1.0f, then rehashing takes place after filling 100% of the current capacity. This may save the space but it will increase the retrieval time of existing elements. Suppose if you choose load factor as 0.5f, then rehashing takes place after filling 50% of the current capacity. This will increase the number of rehashing operations. This will further degrade the HashMap in terms of both space and time.

So, you have to be very careful while choosing the initial capacity and load factor of an HashMap object. Choose the initial capacity and load factor such that they minimize the number of rehashing operations.

# 15.4 Differences Between HashMap Vs HashSet In Java:

HashMap and HashSet, though they spell similar, are totally two different data structures in the Java Collection Framework. HashMap is inherited from the Map interface where as HashSet is inherited from the Set interface. The structure in which they hold the data is also different. HashMap holds the data as key-value pairs where as HashSet holds the data as only objects. There are also some similarities exist between them. In this post, we discuss some of the differences and similarities between HashMap Vs HashSet in java.

### **Differences Between HashMap And HashSet In Java :**

**1) Hierarchy**

HashSet implements the Set interface which in turn extends the Collection interface, the top level interface in the Java Collection Framework. But, HashMap implements the Map interface which starts it’s own hierarchy totally different from the Collection interface.

**2) Data Storage**

HashSet stores the data as objects where as HashMap stores the data as key-value pairs. Where each value is recognized and retrieved by it’s key.

**3) Internal Structure**

HashSet internally uses HashMap to store it’s elements *[*[*See more*](https://javaconceptoftheday.com/how-hashset-works-internally-in-java/)*].*HashSet is sometimes considered as a wrapper around the HashMap. On the other hand, HashMap internally uses an array of Entry<K, V> objects to store the data.

**4) Duplicate Values**

HashSet doesn’t allow duplicate elements. If you try to insert duplicate element, HashSet will be unchanged. Where as HashMap allows duplicate values but doesn’t allow duplicate keys.

**5) null values**

HashSet can hold only one null value where as HashMap can hold multiple null values but allows only one null key.

**6) Insertion Operation**

Insertion operation on HashSet requires only one object where as insertion operation on HashMap requires two objects, key and value.

**7) Performance**

Performance of both is almost the same. But, some developers say that HashMap is slightly faster than the HashSet.

**8) Usage**

Use the HashSet when you need uniqueness of the data. Otherwise, HashMap is always preferred as HashSet internally uses HashMap to store the data.

### **Similarities Between HashMap And HashSet In Java :**

1) Both data structures don’t maintain any order for the elements.

2) Both use hashCode() and equals() method to maintain the uniqueness of the data.

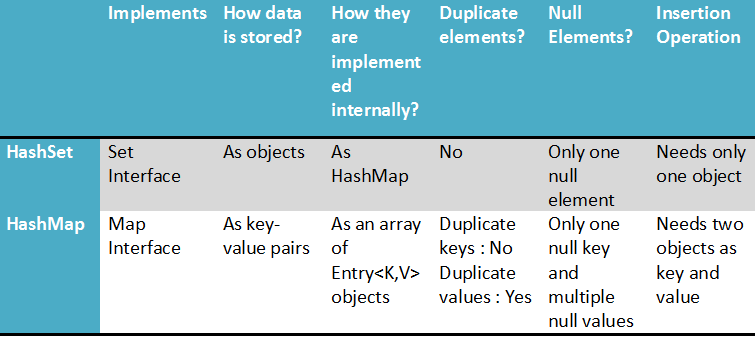
3) The iterators returned by both are fail-fast in nature.

4) Both give constant time performance for insertion and removal operations.

5) Both are not synchronized.

### **HashMap vs HashSet In Java :**

|  |  |
| --- | --- |
| **HashSet** | **HashMap** |
| HashSet implements Set interface. | HashMap implements Map interface. |
| HashSet stores the data as objects. | HashMap stores the data as key-value pairs. |
| HashSet internally uses HashMap. | HashMap internally uses an array of Entry<K, V> objects. |
| HashSet doesn’t allow duplicate elements. | HashMap doesn’t allow duplicate keys, but allows duplicate values. |
| HashSet allows only one null element. | HashMap allows one null key and multiple null values. |
| Insertion operation requires only one object. | Insertion operation requires two objects, key and value. |
| HashSet is slightly slower than HashMap. | HashMap is slightly faster than HashSet. |



# 15.5 Differences Between HashMap And HashTable In Java:

HashMap and HashTable in java are two important data structures in the Collection Framework which have some common things between them. Both implement Map interface. Both store the data in the form of key-value pairs. Both use Hashing technique to store the elements. But, there also exist significant differences between them. One important difference being the thread safety. HashMap is not thread safe where as HashTable is thread safe. In this post, we will discuss the differences and similarities between HashMap Vs HashTable in java.

### **Differences Between HashMap And HashTable In Java :**

**1) Thread Safe**

HashTable is internally synchronized. Therefore, it is very much safe to use HashTable in multi threaded applications. Where as HashMap is not internally synchronized. Therefore, it is not safe to use HashMap in multi threaded applications without external synchronization. You can externally synchronize HashMap using Collections.synchronizedMap() method.

**2) Inherited From**

Though both HashMap and HashTable implement Map interface, but they extend two different classes. HashMap extends AbstractMap class where as HashTable extends Dictionary class which is the legacy class in java.

**3) Null Keys And Null Values**

HashMap allows maximum one null key and any number of null values. Where as HashTable doesn’t allow even a single null key and null value.

**4) Traversal**

HashMap returns only Iterators which are used to traverse over the elements of HashMap. HashTable returns Iterator as well as Enumeration which can be used to traverse over the elements of HashTable.

**5) Fail-Fast Vs Fail-Safe**

Iterator returned by HashMap are fail-fast in nature i.e they throw ConcurrentModificationException if the HashMap is modified after the creation of Iterator other than iterator’s own remove() method. On the other hand, Enumeration returned by the HashTable are fail-safe in nature i.e they don’t throw any exceptions if the HashTable is modified after the creation of Enumeration.

**6) Performance**

As HashTable is internally synchronized, this makes HashTable slightly slower than the HashMap.

**7) Legacy Class**

HashTable is a legacy class. It is almost considered as due for deprecation. Since JDK 1.5, ConcurrentHashMap is considered as better option than the HashTable.

**8) Member Of Java Collection Framework**

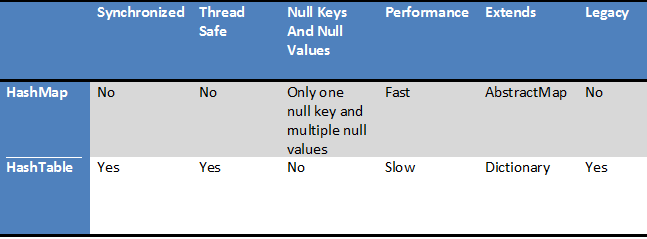
HashMap is a member of Java Collection Framework right from the beginning of its introduction in JDK 1.2. But, HashTable was there before JDK 1.2. From JDK 1.2, it has been made to implement Map interface, making it a member of collection framework.

**9) When To Use What?**

HashMap is always recommended if you don’t want thread safety. If you want thread safety, use either ConcurrentHashMap or make HashMap thread safe by using external synchronization through Collections.synchronizedMap() method. HashTable is not always recommended to use as it is considered as a legacy class.

### **HashMap Vs HashTable In Java :**

|  |  |
| --- | --- |
| **HashMap** | **HashTable** |
| HashMap is not synchronized and therefore it is not thread safe. | HashTable is internally synchronized and therefore it is thread safe. |
| HashMap allows maximum one null key and any number of null values. | HashTable doesn’t allow null keys and null values. |
| Iterators returned by the HashMap are fail-fast in nature. | Enumeration returned by the HashTable are fail-safe in nature. |
| HashMap extends AbstractMap class. | HashTable extends Dictionary class. |
| HashMap returns only iterators to traverse. | HashTable returns both Iterator as well as Enumeration for traversal. |
| HashMap is fast. | HashTable is slow. |
| HashMap is not a legacy class. | HashTable is a legacy class. |
| HashMap is preferred in single threaded applications. If you want to use HashMap in multi threaded application, wrap it using Collections.synchronizedMap() method. | Although HashTable is there to use in multi threaded applications, now a days it is not at all preferred. Because, ConcurrentHashMap is better option than HashTable. |



### **Similarities Between HashMap And HashTable In Java :**

1) Both store the data in the form of key-value pairs.

2) Both use Hashing technique to store the key-value pairs.

3) Both implement Map interface.

4) Both doesn’t maintain any order for elements.

5) Both give constant time performance for insertion and retrieval operations.

# 15.6 15 Java HashMap Programs And Examples:

### **15 Java HashMap Programs And Examples For Beginners :**

**1) Explain the different ways of creating HashMap in java?**

Below example shows 4 different methods for creating HashMap.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.util.HashMap;    public class ExampleOne  {      public static void main(String[] args)      {          //1. Creating HashMap with default initial capacity and load factor            HashMap<String, Integer> map1 = new HashMap<String, Integer>();            //2. Creating HashMap with 30 as initial capacity            HashMap<String, Integer> map2 = new HashMap<String, Integer>(30);            //3. Creating HashMap with 30 as initial capacity and 0.5 as load factor            HashMap<String, Integer> map3 = new HashMap<String, Integer>(30, 0.5f);            //4. Creating HashMap by copying another HashMap            HashMap<String, Integer> map4 = new HashMap<String, Integer>(map1);      }  } |

**2) How do you add key-value pairs to HashMap?**

By using put() and putAll() methods. put() method adds key-value pair one by one where as putAll() method copies all key-value pairs from one HashMap to another HashMap.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | import java.util.HashMap;  import java.util.Map.Entry;  import java.util.Set;    public class JavaHashMapPrograms  {      public static void main(String[] args)      {          //Creating HashMap with default initial capacity and load factor            HashMap<String, Integer> map = new HashMap<String, Integer>();            //Inserting key-value pairs to map using put() method            map.put("ONE", 1);            map.put("TWO", 2);            map.put("THREE", 3);            map.put("FOUR", 4);            map.put("FIVE", 5);            //Printing key-value pairs            Set<Entry<String, Integer>> entrySet = map.entrySet();            for (Entry<String, Integer> entry : entrySet)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }            System.out.println("-------------------------");            //Creating another HashMap            HashMap<String, Integer> anotherMap = new HashMap<String, Integer>();            //Inserting key-value pairs to anotherMap using put() method            anotherMap.put("SIX", 6);            anotherMap.put("SEVEN", 7);            //Inserting key-value pairs of map to anotherMap using putAll() method            anotherMap.putAll(map);            //Printing key-value pairs of anotherMap            entrySet = anotherMap.entrySet();            for (Entry<String, Integer> entry : entrySet)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }      }  } |

**Output :**

FIVE : 5  
ONE : 1  
FOUR : 4  
TWO : 2  
THREE : 3  
————————-  
FIVE : 5  
SIX : 6  
ONE : 1  
FOUR : 4  
TWO : 2  
SEVEN : 7  
THREE : 3

**3) How do you add given key-value pair to HashMap if and only if it is not present in the HashMap?**

Using putIfAbsent() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | import java.util.HashMap;  import java.util.Map.Entry;  import java.util.Set;    public class HashMapExampleThree  {      public static void main(String[] args)      {          //Creating HashMap with default initial capacity and load factor            HashMap<String, Integer> map = new HashMap<String, Integer>();            //Adding key-value pairs            map.put("ONE", 1);            map.put("TWO", 2);            map.put("THREE", 3);            map.put("FOUR", 4);            //Adds key-value pair 'ONE-111' only if it is not present in map            map.putIfAbsent("ONE", 111);            //Adds key-value pair 'FIVE-5' only if it is not present in map            map.putIfAbsent("FIVE", 5);            //Printing key-value pairs of map            Set<Entry<String, Integer>> entrySet = map.entrySet();            for (Entry<String, Integer> entry : entrySet)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }      }  } |

**Output :**

FIVE : 5  
ONE : 1  
FOUR : 4  
TWO : 2  
THREE : 3

**4) How do you retrieve a value associated with a given key from the HashMap?**

Using get() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | import java.util.HashMap;    public class HashMapExampleFour  {      public static void main(String[] args)      {          //Creating HashMap with default initial capacity and load factor            HashMap<String, Integer> map = new HashMap<String, Integer>();            //Adding key-value pairs to HashMap            map.put("ONE", 1);            map.put("TWO", 2);            map.put("THREE", 3);            map.put("FOUR", 4);            //Retrieving a value associated with key 'TWO'            int value = map.get("TWO");            System.out.println(value);       //Output : 2      }  } |

**5) How do you check whether a particular key/value exist in a HashMap?**

Using containsKey() and containsValue() methods.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | import java.util.HashMap;    public class ExampleFive  {      public static void main(String[] args)      {          //Creating the HashMap            HashMap<Integer, Double> map = new HashMap<Integer, Double>();            //Adding key-value pairs to HashMap            map.put(1, 1.1);            map.put(2, 2.2);            map.put(3, 3.3);            map.put(4, 4.4);            //Checking whether key '3' exist in map            System.out.println(map.containsKey(3));      //Output : true            //Checking whether value '3.3' exist in map            System.out.println(map.containsValue(3.3));   //Output : true      }  } |

**6) How do you find out the number of key-value mappings present in a HashMap?**

Using size() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | import java.util.HashMap;    public class JavaHashMapPrograms  {      public static void main(String[] args)      {          //Creating the HashMap            HashMap<Integer, Double> map = new HashMap<Integer, Double>();            //Adding key-value pairs to HashMap            map.put(111, 111.111);            map.put(222, 222.222);            map.put(333, 333.333);            map.put(444, 444.444);            map.put(555, 555.555);            //Retrieving the number of key-value pairs present in map            System.out.println(map.size());      //Output : 5      }  } |

**7) How do you remove all key-value pairs from a HashMap? OR How do you clear the HashMap for reuse?**

using clear() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | import java.util.HashMap;    public class JavaHashMapExampleSeven  {      public static void main(String[] args)      {          //Creating the HashMap            HashMap<Integer, Double> map = new HashMap<Integer, Double>();            //Adding key-value pairs to HashMap            map.put(111, 111.111);            map.put(222, 222.222);            map.put(333, 333.333);            map.put(444, 444.444);            map.put(555, 555.555);            //Retrieving the number of key-value pairs            System.out.println(map.size());      //Output : 5            //Clearing the map            map.clear();            //Checking the number of key-value pairs after clearing the map            System.out.println(map.size());      //Output : 0      }  } |

**8) How do you retrieve all keys present in a HashMap?**

keySet() method returns all keys present in a HashMap in the form of Set.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | import java.util.HashMap;  import java.util.Set;    public class JavaHashMapExample  {      public static void main(String[] args)      {          //Creating the HashMap            HashMap<Integer, String> map = new HashMap<Integer, String>();            //Adding key-value pairs to HashMap            map.put(1, "AAA");            map.put(2, "BBB");            map.put(3, "CCC");            map.put(4, "DDD");            map.put(5, "EEE");            //Retrieving the Key Set            Set<Integer> keySet = map.keySet();            for (Integer key : keySet)          {              System.out.println(key);          }      }  } |

**Output :**

1  
2  
3  
4  
5

**9) How do you retrieve all the values present in a HashMap?**

Using values() method. This method returns Collection view of all the values present in a HashMap.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | import java.util.Collection;  import java.util.HashMap;    public class HashMapExampleNine  {      public static void main(String[] args)      {          //Creating the HashMap            HashMap<Integer, String> map = new HashMap<Integer, String>();            //Adding key-value pairs to HashMap            map.put(1, "AAA");            map.put(2, "BBB");            map.put(3, "CCC");            map.put(4, "DDD");            map.put(5, "EEE");            //Retrieving the Collection view of values present in map            Collection<String> values = map.values();            for (String value : values)          {              System.out.println(value);          }      }  } |

**Output :**

AAA  
BBB  
CCC  
DDD  
EEE

**10) How do you retrieve all key-value pairs present in a HashMap?**

entrySet() method returns all key-value pairs present in a HashMap in the form of Set.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | import java.util.HashMap;  import java.util.Map.Entry;  import java.util.Set;    public class JavaHashMapPrograms  {      public static void main(String[] args)      {          //Creating the HashMap            HashMap<String, String> map = new HashMap<String, String>();            //Adding key-value pairs to HashMap            map.put("ONE", "AAA");            map.put("TWO", "BBB");            map.put("THREE", "CCC");            map.put("FOUR", "DDD");            map.put("FIVE", "EEE");            //Retrieving the Set consists of all key-value pairs in map            Set<Entry<String, String>> keyValueSet = map.entrySet();            for (Entry<String, String> entry : keyValueSet)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }      }  } |

**Output :**

FIVE : EEE  
ONE : AAA  
FOUR : DDD  
TWO : BBB  
THREE : CCC

**11) How do you remove a key-value pair from the HashMap?**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | import java.util.HashMap;  import java.util.Map.Entry;  import java.util.Set;    public class MainClass  {      public static void main(String[] args)      {          //Creating the HashMap            HashMap<String, String> map = new HashMap<String, String>();            //Adding key-value pairs to HashMap            map.put("ONE", "AAA");            map.put("TWO", "BBB");            map.put("THREE", "CCC");            map.put("FOUR", "DDD");            map.put("FIVE", "EEE");            //Printing key-value pairs            System.out.println("HashMap Before Remove :");            Set<Entry<String, String>> keyValueSet = map.entrySet();            for (Entry<String, String> entry : keyValueSet)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }            System.out.println("---------------------");            //Removing the mapping for the key 'ONE'            map.remove("ONE");            System.out.println("HashMap After Remove :");            for (Entry<String, String> entry : keyValueSet)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }      }  } |

**Output :**

HashMap Before Remove :  
FIVE : EEE  
ONE : AAA  
FOUR : DDD  
TWO : BBB  
THREE : CCC  
———————  
HashMap After Remove :  
FIVE : EEE  
FOUR : DDD  
TWO : BBB  
THREE : CCC

**12) How do you remove a key-value pair from a HashMap if and only if the specified key is currently mapped to given value?**

Another version of remove() method which takes two arguments – one is key and another one is value, removes the mapping for the specified key only if it is currently mapped to given value.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | import java.util.HashMap;  import java.util.Map.Entry;  import java.util.Set;    public class JavaHashMapExample  {      public static void main(String[] args)      {          //Creating the HashMap            HashMap<String, String> map = new HashMap<String, String>();            //Adding key-value pairs to HashMap            map.put("ONE", "AAA");            map.put("TWO", "BBB");            map.put("THREE", "CCC");            map.put("FOUR", "DDD");            map.put("FIVE", "EEE");            //Printing Key-value pairs            System.out.println("HashMap Before Remove :");            Set<Entry<String, String>> keyValueSet = map.entrySet();            for (Entry<String, String> entry : keyValueSet)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }            System.out.println("------------------");            //Removes the mapping for the key 'ONE' only if it is currently mapped to 'CCC'            map.remove("ONE", "CCC");            //Removes the mapping for the key 'FIVE' only if it is currently mapped to 'EEE'            map.remove("FIVE", "EEE");            System.out.println("HashMap After Remove :");            for (Entry<String, String> entry : keyValueSet)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }      }  } |

**Output :**

HashMap Before Remove :  
FIVE : EEE  
ONE : AAA  
FOUR : DDD  
TWO : BBB  
THREE : CCC  
——————  
HashMap After Remove :  
ONE : AAA  
FOUR : DDD  
TWO : BBB  
THREE : CCC

**13) How do you replace a value associated with a given key in the HashMap?**

replace() method replaces the value associated with the specified key if the key is currently mapped to some value.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | import java.util.HashMap;  import java.util.Map.Entry;  import java.util.Set;    public class JavaHashMapPrograms  {      public static void main(String[] args)      {          //Creating the HashMap            HashMap<String, String> map = new HashMap<String, String>();            //Adding key-value pairs to HashMap            map.put("ONE", "AAA");            map.put("TWO", "BBB");            map.put("THREE", "CCC");            map.put("FOUR", "DDD");            map.put("FIVE", "EEE");            //Printing Key-value pairs            System.out.println("HashMap Before Replace :");            Set<Entry<String, String>> keyValueSet = map.entrySet();            for (Entry<String, String> entry : keyValueSet)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }            System.out.println("------------------");            //Replacing the value associated with 'THREE' to '333'            map.replace("THREE", "333");            System.out.println("HashMap After Replace :");            for (Entry<String, String> entry : keyValueSet)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }      }  } |

**Output :**

HashMap Before Replace :  
FIVE : EEE  
ONE : AAA  
FOUR : DDD  
TWO : BBB  
THREE : CCC  
——————  
HashMap After Replace :  
FIVE : EEE  
ONE : AAA  
FOUR : DDD  
TWO : BBB  
THREE : 333

**14) How do you replace a value associated with the given key if and only if it is currently mapped to given value?**

Another version of replace() method which takes three arguments, replaces the value associated with the given key only if it is currently mapped to given value.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | import java.util.HashMap;  import java.util.Map.Entry;  import java.util.Set;    public class JavaHashMapExample  {      public static void main(String[] args)      {          //Creating the HashMap            HashMap<String, String> map = new HashMap<String, String>();            //Adding key-value pairs to HashMap            map.put("ONE", "AAA");            map.put("TWO", "BBB");            map.put("THREE", "CCC");            map.put("FOUR", "DDD");            map.put("FIVE", "EEE");            //Printing Key-value pairs            System.out.println("HashMap Before Replace :");            Set<Entry<String, String>> keyValueSet = map.entrySet();            for (Entry<String, String> entry : keyValueSet)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }            System.out.println("------------------");            //Replacing the value associated with 'FOUR' to '444' only if it is currently mapped to 'DDD'            map.replace("FOUR", "DDD", "444");            System.out.println("HashMap After Replace :");            for (Entry<String, String> entry : keyValueSet)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }      }  } |

**Output :**

HashMap Before Replace :  
FIVE : EEE  
ONE : AAA  
FOUR : DDD  
TWO : BBB  
THREE : CCC  
——————  
HashMap After Replace :  
FIVE : EEE  
ONE : AAA  
FOUR : 444  
TWO : BBB  
THREE : CCC

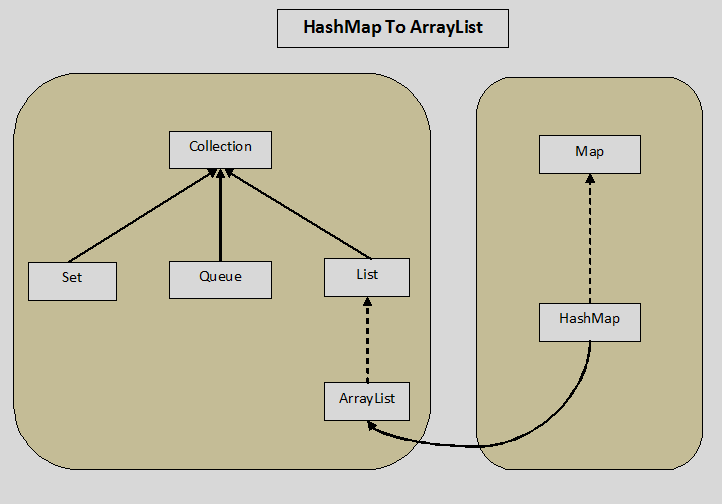
**15) How do you get synchronized HashMap in java?**

Using Collections.synchronizedMap() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | import java.util.Collections;  import java.util.HashMap;  import java.util.Map;    public class JavaHashMapPrograms  {      public static void main(String[] args)      {          //Creating the HashMap            HashMap<String, Integer> map = new HashMap<String, Integer>();            //Getting synchronized Map            Map<String, Integer> syncMap = Collections.synchronizedMap(map);      }  } |

# 15.7 Convert HashMap To ArrayList In Java – Updated With Java 8 Code:

HashMap and ArrayList are two most used data structures in java. Both classes inherit from different hierarchies. HashMap is inherited from Map interface which represents the data in the form of key-value pairs. ArrayList is inherited from List interface which arranges the data in the sequential manner. Conversion of HashMap to ArrayList has also become a regular question in the java interviews as there is no direct methods in HashMap which converts the HashMap to ArrayList. In this post, we will see how to convert HashMap to ArrayList in java with examples. At the end, we will also see java 8 code to convert map to list in java.



### **How To Convert HashMap To ArrayList In Java?**

As HashMap contains key-value pairs, there are three ways you can convert given HashMap to ArrayList. You can convert HashMap keys into ArrayList or you can convert HashMap values into ArrayList or you can convert key-value pairs into ArrayList. Let’s see these three methods in detail.

**a) Conversion Of HashMap Keys Into ArrayList :**

For this, we use keySet() method of HashMap which returns the Set containing all keys of the HashMap. And then we pass this Set while constructing the ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | //Creating a HashMap object    HashMap<String, String> map = new HashMap<String, String>();    //Getting Set of keys from HashMap    Set<String> keySet = map.keySet();    //Creating an ArrayList of keys by passing the keySet    ArrayList<String> listOfKeys = new ArrayList<String>(keySet); |

**b) Conversion Of HashMap Values Into ArrayList :**

For this, we use values() method of HashMap which returns the Collection containing all values of the HashMap. Then we use this Collection to create the ArrayList of values.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | //Creating a HashMap object    HashMap<String, String> map = new HashMap<String, String>();    //Getting Collection of values from HashMap    Collection<String> values = map.values();    //Creating an ArrayList of values    ArrayList<String> listOfValues = new ArrayList<String>(values); |

**c) Conversion Of HashMap’s Key-Value Pairs Into ArrayList :**

For this, we use entrySet() method of HashMap which returns the Set of Entry<K, V> objects where each Entry object represents one key-value pair. We pass this Set to create the ArrayList of key-value pairs.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | //Creating a HashMap object    HashMap<String, String> map = new HashMap<String, String>();    //Getting Set of entries from HashMap    Set<Entry<String, String>> entrySet = map.entrySet();    //Creating an ArrayList of Entry objects    ArrayList<Entry<String, String>> listOfEntry = new ArrayList<Entry<String, String>>(entrySet); |

### **Java Program To Convert HashMap To ArrayList :**

Below example converts the studentPerformanceMap to listOfKeys, listOfValues and listOfEntry.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77 | import java.util.ArrayList;  import java.util.Collection;  import java.util.HashMap;  import java.util.Map.Entry;  import java.util.Set;  public class Java8MapToListExamples  {      public static void main(String[] args)      {          //Creating a HashMap object            HashMap<String, String> studentPerformanceMap = new HashMap<String, String>();            //Adding elements to HashMap            studentPerformanceMap.put("John Kevin", "Average");            studentPerformanceMap.put("Rakesh Sharma", "Good");            studentPerformanceMap.put("Prachi D", "Very Good");            studentPerformanceMap.put("Ivan Jose", "Very Bad");            studentPerformanceMap.put("Smith Jacob", "Very Good");            studentPerformanceMap.put("Anjali N", "Bad");            //Getting Set of keys            Set<String> keySet = studentPerformanceMap.keySet();            //Creating an ArrayList of keys            ArrayList<String> listOfKeys = new ArrayList<String>(keySet);            System.out.println("ArrayList Of Keys :");            for (String key : listOfKeys)          {              System.out.println(key);          }            System.out.println("--------------------------");            //Getting Collection of values            Collection<String> values = studentPerformanceMap.values();            //Creating an ArrayList of values            ArrayList<String> listOfValues = new ArrayList<String>(values);            System.out.println("ArrayList Of Values :");            for (String value : listOfValues)          {              System.out.println(value);          }            System.out.println("--------------------------");            //Getting the Set of entries            Set<Entry<String, String>> entrySet = studentPerformanceMap.entrySet();            //Creating an ArrayList Of Entry objects            ArrayList<Entry<String, String>> listOfEntry = new ArrayList<Entry<String,String>>(entrySet);            System.out.println("ArrayList of Key-Values :");            for (Entry<String, String> entry : listOfEntry)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }      }  } |

**Output :**

ArrayList Of Keys :  
Rakesh Sharma  
Anjali N  
Smith Jacob  
John Kevin  
Ivan Jose  
Prachi D  
————————–  
ArrayList Of Values :  
Good  
Bad  
Very Good  
Average  
Very Bad  
Very Good  
————————–  
ArrayList of Key-Values :  
Rakesh Sharma : Good  
Anjali N : Bad  
Smith Jacob : Very Good  
John Kevin : Average  
Ivan Jose : Very Bad  
Prachi D : Very Good

#### **Java 8 – Convert Map To List**

**a) Java 8 – Convert Map Keys To List**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | //Creating a Map object    Map<String, Integer> map = new HashMap<String, Integer>();    //Java 8 code to convert map keys to list    List<String> listOfKeys = map.keySet().stream().collect(Collectors.toList());    //Java 8 code to print List elements    listOfKeys.forEach(System.out::println); |

**b) Java 8 – Convert Map Values To List**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | //Creating a Map object    Map<String, Integer> map = new HashMap<String, Integer>();    //Java 8 code to convert map values to list    List<Integer> listOfValues = map.values().stream().collect(Collectors.toList());    //Java 8 code to print List elements    listOfValues.forEach(System.out::println); |

**c) Java 8 – Sort And Convert Map Keys To List**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | //Creating a Map object    Map<String, Integer> map = new HashMap<String, Integer>();    //Java 8 code to sort and convert map keys to list    List<String> listOfKeys = map.keySet().stream().sorted().collect(Collectors.toList());    //Java 8 code to print List elements    listOfKeys.forEach(System.out::println); |

**d) Java 8 – Sort And Convert Map Values To List**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | //Creating a Map object    Map<String, Integer> map = new HashMap<String, Integer>();    //Java 8 code to sort and convert map values to list    List<Integer> listOfValues = map.values().stream().sorted().collect(Collectors.toList());    //Java 8 code to print List elements    listOfValues.forEach(System.out::println); |

Note : You can also provide Comparator to sorted() method to sort the keys or values as you wish.

# 15.8 HashMap Vs ConcurrentHashMap In Java:

HashMap and ConcurrentHashMap are two important data structures in Java. Both hold data in the form of key-value pairs. HashMap is the part of Java Collection Framework since JDK 1.2 where as ConcurrentHashMap is introduced in JDK 1.5. ConcurrentHashMap is thread safe, most suitable for concurrent multi threaded environment. HashMap is not thread safe, hence most suitable for single threaded applications. In this post, we will see the differences between HashMap Vs ConcurrentHashMap in Java.

#### **Differences Between HashMap And ConcurrentHashMap In Java :**

**1) Thread Safe**

The main difference between HashMap and ConcurrentHashMap is that ConcurrentHashMap is internally synchronized and hence it is thread safe. Where as HashMap is not synchronized internally and it is not thread safe. You can make HashMap synchronized externally using Collections.synchronizedMap() method.

**2) Internal Structure**

Not all the operations in ConcurrentHashMap are synchronized. Only modifying operations like add and delete are synchronized. Read operations are not synchronized. This will make ConcurrentHashMap a first choice map for concurrent multi threaded applications than the externally synchronized HashMap.

Because, when you make HashMap synchronized externally using Collections.synchronizedMap() method, all the operations will be synchronized. This will slow down the application.

**3) Introduction Into Java Collection Framework**

HashMap is the part of Java Collection Framework since JDK 1.2. ConcurrentHashMap is introduced later as a part of concurrency package into Java Collection Framework. ConcurrentHashMap is largely treated as an alternative to HashTable which is the legacy class.

**4) Null Keys And Null Values**

HashMap allows maximum one null key and any number of null values. ConcurrentHashMap doesn’t allow even a single null key and a null value.

**5) Fail-Fast Vs Fail-Safe**

Iterators returned by HashMap are fail-fast in nature. Because they throw ConcurrentModificationException if the map is modified after the creation of iterator. Where as iterators returned by ConcurrentHashMap are fail-safe in nature. They don’t throw any exceptions if the map is modified after the creation of iterator.

**6) Performance**

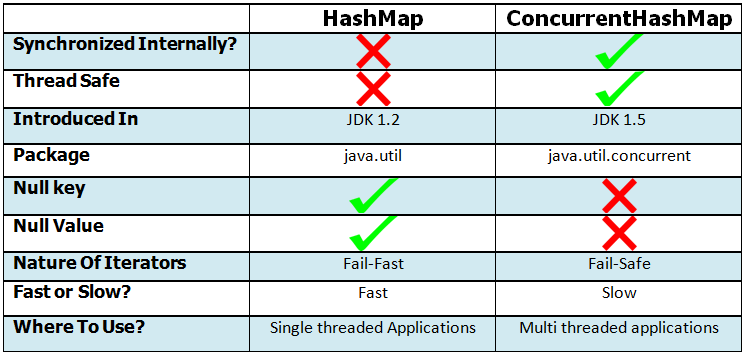
Only modifying operations on ConcurrentHashMap are synchronized. Hence, add or remove operations on ConcurrentHashMap are slower than on HashMap. The read operations on both, ConcurrentHashMap and HashMap, give same performance as read operations on both maps are not synchronized.

**7) When To Use What?**

As ConcurrentHashMap is internally synchronized and hence it is most suitable for concurrent multi threaded applications. HashMap is not synchronized internally and it is most suitable for single threaded applications.

#### **HashMap Vs ConcurrentHashMap In Java :**

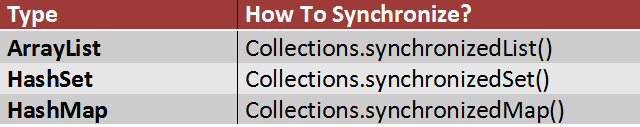
|  |  |
| --- | --- |
| **HashMap** | **ConcurrentHashMap** |
| HashMap is not synchronized internally and hence it is not thread safe. | ConcurrentHashMap is internally synchronized and hence it is thread safe. |
| HashMap is the part of Java collection framework since JDK 1.2. | ConcurrentHashMap is introduced in JDK 1.5 as an alternative to HashTable. |
| HashMap allows maximum one null key and any number of null values. | ConcurrentHashMap doesn’t allow even a single null key and null value. |
| Iterators returned by HashMap are fail-fast in nature. | Iterators returned by ConcurrentHashMap are fail-safe in nature. |
| HashMap is faster. | ConcurrentHashMap is slower. |
| Most suitable for single threaded applications. | Most suitable for multi threaded applications. |



# 16. How To Synchronize ArrayList, HashSet And HashMap In Java?

ArrayList, HashSet and HashMap are three most frequently used data structures in java. As they are most used, they are not synchronized for the sake of performance. But, java provides the methods to make them synchronized as and when the need arises. These methods are introduced in [*java.util.Collections*](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html) class. Collections class is an utility class which has some useful methods helpful for operations on collection types. In this post, we will see how to synchronize ArrayList, HashSet and HashMap in java.

### **How to synchronize ArrayList, HashSet and HashMap in java?**



### **How To Synchronize ArrayList In Java?**

To synchronize ArrayList, we use Collections.synchronizedList() method. This method returns synchronized list backed by the specified list. There is an advise from javadocs that while iterating over the synchronized list, you must use it in a synchronized block. Failed to do so may result in non-deterministic behavior.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | import java.util.ArrayList;  import java.util.Collections;  import java.util.Iterator;  import java.util.List;    public class SynchronizedListExample  {      public static void main(String[] args)      {          //Creating non synchronized ArrayList object            ArrayList<String> list = new ArrayList<String>();            //Adding elements to list            list.add("JAVA");            list.add("STRUTS");            list.add("JSP");            list.add("SERVLETS");            list.add("JSF");            //Getting synchronized list            List<String> synchronizedList = Collections.synchronizedList(list);            //you must use synchronized block while iterating over synchronizedList            synchronized (synchronizedList)          {              Iterator<String> it = synchronizedList.iterator();                while (it.hasNext())              {                  System.out.println(it.next());              }          }      }  } |

**OUTPUT :**

JAVA  
STRUTS  
JSP  
SERVLETS  
JSF

### **How To Synchronize HashSet In Java?**

We use Collections.synchronizedSet() method to synchronize HashSet. This method returns synchronized set backed by the specified set. There is also an advice from javadocs that you must use this synchronized set in a synchronized block while iterating over it. If you don’t do this, it may result in non-deterministic behavior.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | import java.util.Collections;  import java.util.HashSet;  import java.util.Iterator;  import java.util.Set;    public class SynchronizedHashSetExample  {      public static void main(String[] args)      {          //Creating non synchronized HashSet object            HashSet<String> set = new HashSet<String>();            //Adding elements to set            set.add("JAVA");            set.add("STRUTS");            set.add("JSP");            set.add("SERVLETS");            set.add("JSF");            //Getting synchronized set            Set<String> synchronizedSet = Collections.synchronizedSet(set);            //you must use synchronized block while iterating over synchronizedSet            synchronized (synchronizedSet)          {              Iterator<String> it = synchronizedSet.iterator();                while (it.hasNext())              {                  System.out.println(it.next());              }          }      }  } |

**Output :**

SERVLETS  
STRUTS  
JSP  
JAVA  
JSF

### **How To Synchronize HashMap In Java?**

We use Collections.synchronizedMap() to synchronize HashMap. This method returns synchronized map backed by the specified map. You must iterate it in a synchronized block to avoid unexpected behavior.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64 | import java.util.Collection;  import java.util.Collections;  import java.util.HashMap;  import java.util.Iterator;  import java.util.Map;  import java.util.Set;    public class SynchronizedHashMapExample  {      public static void main(String[] args)      {          //Creating HashMap object which is not synchronized            HashMap<String, Integer> map = new HashMap<String, Integer>();            //Adding elements to map            map.put("ONE", 1);            map.put("TWO", 2);            map.put("THREE", 3);            map.put("FOUR", 4);            map.put("FIVE", 5);            //Getting synchronized map            Map<String, Integer> synchronizedMap = Collections.synchronizedMap(map);            Set<String> keySet = synchronizedMap.keySet();            System.out.println("Keys.............");            //While iterating over synchronizedMap, you must use synchronized block.            synchronized (synchronizedMap)          {              Iterator<String> it = keySet.iterator();                while (it.hasNext())              {                  System.out.println(it.next());              }          }            Collection<Integer> values = synchronizedMap.values();            System.out.println("Values.............");            //While iterating over synchronizedMap, you must use synchronized block.            synchronized (synchronizedMap)          {              Iterator<Integer> it = values.iterator();                while (it.hasNext())              {                  System.out.println(it.next());              }          }      }  } |

**Output :**

Keys………….  
ONE  
TWO  
THREE  
FOUR  
FIVE  
Values………….  
1  
2  
3  
4  
5

# 16.1 Differences Between Enumeration Vs Iterator In Java:

Enumeration and Iterator are two interfaces in java.util package which are used to traverse over the elements of a Collection object. Though they perform the same function i.e traversing the Collection object, there are some differences exist between them. Using Enumeration, you can only traverse the Collection object. But using Iterator, you can also remove an element while traversing the Collection. This is the one major difference between Enumeration and Iterator in java. You can say Iterator is some what advanced version of Enumeration. In this post, we will see the differences between Enumeration Vs Iterator In Java.

### **Differences Between Enumeration And Iterator In Java :**

**1) Introduction**

Iterator interface is introduced from JDK 1.2 where as Enumeration interface is there from JDK 1.0.

**2) remove() method**

This is the main difference between Enumeration and Iterator interface. Enumeration only traverses the Collection object. You can’t do any modifications to Collection while traversing the Collection using Enumeration. Where as Iterator interface allows us to remove an element while traversing the Collection object. Iterator has remove() method which is not there in the Enumeration interface. Below is the list of Enumeration and Iterator methods.

|  |  |
| --- | --- |
| **Iterator** | **Enumeration** |
| hasNext() | hasMoreElements() |
| next() | nextElement() |
| remove() | (Not Available) |

**3) Legacy Interface**

Enumeration is a legacy interface used to traverse only the legacy classes like Vector, HashTable and Stack. Where as Iterator is not a legacy code which is used to traverse most of the classes in the collection framework. For example, ArrayList, LinkedList, HashSet, LinkedHashSet, TreeSet, HashMap, LinkedHashMap, TreeMap etc.

**4) Fail-Fast Vs Fail-Safe**

Iterator is a fail-fast in nature. i.e it throws ConcurrentModificationException if a collection is modified while iterating other than it’s own remove() method. Where as Enumeration is fail-safe in nature. It doesn’t throw any exceptions if a collection is modified while iterating. [[See more](https://javaconceptoftheday.com/fail-fast-and-fail-safe-iterators-in-java-with-examples/)]

**5) Safe And Secure**

As Iterator is fail-fast in nature and doesn’t allow modification of a collection by other threads while iterating, it is considered as safe and secure than Enumeration.

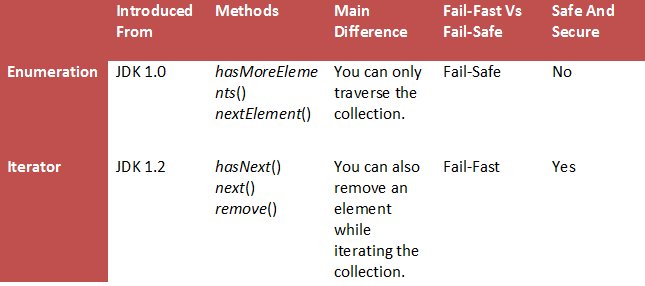
**6) Which One To Use**

According to Java API Docs, Iterator is always preferred over the Enumeration. Here is the note from the [Enumeration Docs](https://docs.oracle.com/javase/8/docs/api/java/util/Enumeration.html).

NOTE: The functionality of this interface is duplicated by the Iterator interface. In addition, Iterator adds an optional remove operation, and has shorter method names. New implementations should consider using Iterator in preference to Enumeration.

### **Enumeration Vs Iterator In Java :**

|  |  |
| --- | --- |
| **Enumeration** | **Iterator** |
| Using Enumeration, you can only traverse the collection. You can’t do any modifications to collection while traversing it. | Using Iterator, you can remove an element of the collection while traversing it. |
| Enumeration is introduced in JDK 1.0 | Iterator is introduced from JDK 1.2 |
| Enumeration is used to traverse the legacy classes like Vector, Stack and HashTable. | Iterator is used to iterate most of the classes in the collection framework like ArrayList, HashSet, HashMap, LinkedList etc. |
| Methods : hasMoreElements() and nextElement() | Methods : hasNext(), next() and remove() |
| Enumeration is fail-safe in nature. | Iterator is fail-fast in nature. |
| Enumeration is not safe and secured due to it’s fail-safe nature. | Iterator is safer and secured than Enumeration. |



# 16.2 Fail Fast Vs Fail Safe Iterators In Java With Examples:

A system is called fail-fast if it is shut down immediately when an error is occurred. These systems don’t carry on with the errors. They immediately stop operating when a fault is occurred in the system. The errors in the fail-fast systems are immediately exposed. But, fail-safe systems are not like that. They don’t stop operating even when a fault is occurred in the system. They continue the operation by hiding the errors. They don’t expose the errors immediately. They carry on with the errors. Which one is the best system is always the most discussed topic in the system design field. In this post, we limit our discussion to Fail Fast Vs Fail Safe Iterators in Java.

### **Fail Fast And Fail Safe Iterators In Java :**

Iterators in Java give us the facility to traverse over the Collection objects. Iterators returned by the collections are either fail-fast or fail-safe in nature. Fail-Fast iterators immediately throw ConcurrentModificationException if a collection is modified while iterating over it. Where as Fail-Safe iterators don’t throw any exceptions if a collection is modified while iterating over it. Because, they operate on the clone of the collection, not on the actual collection. Let’s see Fail-Fast and Fail-Safe Iterators in detail.

### **Fail-Fast Iterators In Java :**

Fail-Fast iterators, returned by most of the collection types, doesn’t tolerate any structural modifications to a collection while iterating over it. (Structural modifications means add, remove or updating an element in the collection). They throw ConcurrentModificationException if a collection is structurally modified while iteration is going on the collection. But, they don’t throw any exceptions if the collection is modified by the iterator’s own methods like remove().

**How Fail-Fast Iterators Work?**

All Collection types maintain an internal array of objects ( Object[] ) to store the elements. Fail-Fast iterators directly fetch the elements from this array. They always consider that this internal array is not modified while iterating over its elements. To know whether the collection is modified or not, they use an internal flag called modCount which is updated each time a collection is modified. Every time when an Iterator calls the next() method, it checks the modCount. If it finds the modCount has been updated after this Iterator has been created, it throws ConcurrentModificationException.

The iterators returned by the ArrayList, HashSet, HashMap etc are all Fail-Fast in nature.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | import java.util.ArrayList;  import java.util.Iterator;    public class FailFastIteratorExample  {      public static void main(String[] args)      {          //Creating an ArrayList of integers            ArrayList<Integer> list = new ArrayList<Integer>();            //Adding elements to list            list.add(1452);            list.add(6854);            list.add(8741);            list.add(6542);            list.add(3845);            //Getting an Iterator from list            Iterator<Integer> it = list.iterator();            while (it.hasNext())          {              Integer integer = (Integer) it.next();                list.add(8457);      //This will throw ConcurrentModificationException          }      }  } |

**Output :**

|  |  |
| --- | --- |
| 1  2  3  4 | Exception in thread "main" java.util.ConcurrentModificationException      at java.util.ArrayList$Itr.checkForComodification(Unknown Source)      at java.util.ArrayList$Itr.next(Unknown Source)      at pack1.MainClass.main(MainClass.java:32) |

### **Fail-Safe Iterators In Java :**

Fail-Safe iterators don’t throw any exceptions if the collection is modified while iterating over it. Because, they iterate on the clone of the collection not on the original collection. So, any structural modifications done on the original collection goes unnoticed by these iterators.

But, these iterators have some drawbacks. One of them is that it is not always guaranteed that you will get up-to-date data while iterating. Because, any modifications to the collection after the creation of iterator is not updated in the iterator. One more disadvantage of these iterators is that there will be additional overhead of creating the copy of the collection in the terms of both time and memory.

Iterator returned by ConcurrentHashMap is a fail-safe iterator.

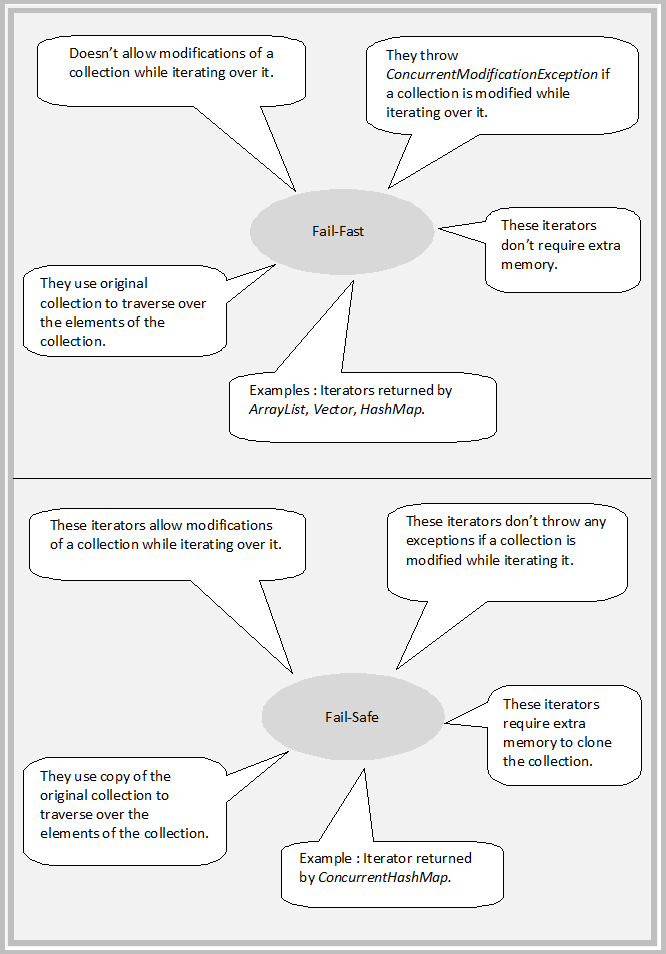
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | import java.util.Iterator;  import java.util.concurrent.ConcurrentHashMap;    public class FailSafeIteratorExample  {      public static void main(String[] args)      {          //Creating a ConcurrentHashMap            ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<String, Integer>();            //Adding elements to map            map.put("ONE", 1);            map.put("TWO", 2);            map.put("THREE", 3);            map.put("FOUR", 4);            //Getting an Iterator from map            Iterator<String> it = map.keySet().iterator();            while (it.hasNext())          {              String key = (String) it.next();                System.out.println(key+" : "+map.get(key));                map.put("FIVE", 5);     //This will not be reflected in the Iterator          }      }  } |

**Output :**

TWO : 2  
FOUR : 4  
ONE : 1  
THREE : 3

### **Fail Fast Vs Fail Safe Iterators In Java :**

|  |  |
| --- | --- |
| **Fail-Fast Iterators** | **Fail-Safe Iterators** |
| Fail-Fast iterators doesn’t allow modifications of a collection while iterating over it. | Fail-Safe iterators allow modifications of a collection while iterating over it. |
| These iterators throw ConcurrentModificationException if a collection is modified while iterating over it. | These iterators don’t throw any exceptions if a collection is modified while iterating over it. |
| They use original collection to traverse over the elements of the collection. | They use copy of the original collection to traverse over the elements of the collection. |
| These iterators don’t require extra memory. | These iterators require extra memory to clone the collection. |
| Ex : Iterators returned by ArrayList, Vector, HashMap. | Ex : Iterator returned by ConcurrentHashMap. |



# 16.3 Difference Between Collection And Collections In Java:

## **What is the difference between Collection and Collections in java?**

This is one of the most confusing java interview question asked many a times to java freshers. Most of time, this question has been asked to java freshers to check their basic knowledge about the **Java Collection Framework**. This question seems confusing because both “**Collection**” and “**Collections**” look similar. Both are part of java collection framework, but both serve different purpose. **Collection** is a top level interface of java collection framework where as **Collections** is an utility class. In this article, we will discuss the differences between Collection and Collections in java.

## **Collection Interface :**

**Collection** is a root level interface of the Java Collection Framework. Most of the classes in Java Collection Framework inherit from this interface. **List**, **Set** and **Queue** are main sub interfaces of this interface. JDK doesn’t provide any direct implementations of this interface. But, JDK provides direct implementations of it’s sub interfaces. **ArrayList**, **Vector**, **HashSet**, **LinkedHashSet**, **PriorityQueue** are some indirect implementations of Collection interface. **Map interface**, which is also a part of java collection framework, doesn’t inherit from Collection interface. Collection interface is a member of java.util package.

Click [here](https://javaconceptoftheday.com/collection-framework-collection-interface/) for more info on Collection interface in java.

## **Collections Class:**

**Collections** is an utility class in java.util package. It consists of only static methods which are used to operate on objects of type Collection. For example, it has the method to find the maximum element in a collection, it has the method to sort the collection, it has the method to search for a particular element in a collection. Below is the list of some important methods of Collections class.

|  |  |
| --- | --- |
| Collections.max() | This method returns maximum element in the specified collection. |
| Collections.min() | This method returns minimum element in the given collection. |
| Collections.sort() | This method sorts the specified collection. |
| Collections.shuffle() | This method randomly shuffles the elements in the specified collection. |
| Collections.synchronizedCollection() | This method returns synchronized collection backed by the specified collection. |
| Collections.binarySearch() | This method searches the specified collection for the specified object using binary search algorithm. |
| Collections.disjoint() | This method returns true if two specified collections have no elements in common. |
| Collections.copy() | This method copies all elements from one collection to another collection. |
| Collections.reverse() | This method reverses the order of elements in the specified collection. |

Click [here](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html) for more info on Collections class.

# 16.4 How To Make Collection Read Only In Java?

### **What Is Read Only Collection In Java?**

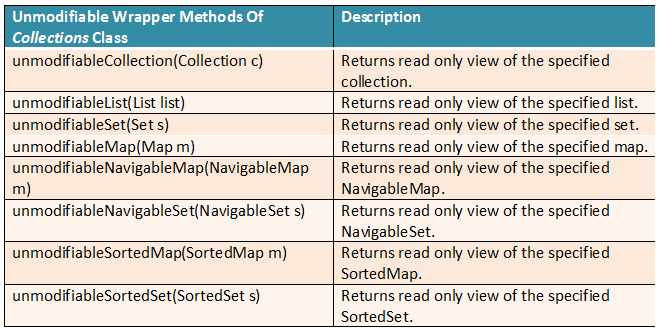
Read only collection or unmodifiable collection is a collection which can not be modified once created. You will not be able to add an element or remove an element or edit an element once you make the collection read only. If you try to perform these operations on read only collection, you will get java.lang.UnsupportedOperationException. In this post, we will see how to make collection read only in java.

### **How To Make Collection Read Only In Java?**

java.util.Collections class provides some unmodifiable wrapper methods to create read only collections in java. These methods take the Collection type as an argument and returns read only view of the specified collection. Any modification operations (like add, delete or edit an element) on the returned collection, direct or via its iterators, will result in UnsupportedOperationException. But, you can perform any modification operations on original collection and those modifications are reflected in the returned collection.

That means, what these unmodifiable wrapper methods do is, any query or read operations you perform on the returned collection, will actually read through the original collection and any modification operations you perform on the returned  collection, direct or via its iterators, will result in UnsupportedOperationException.

Below table shows complete list of all unmodifiable wrapper methods of Collections class which are used to create read only collections.



Let’s see how to make some of the important Collection types like ArrayList, HashSet and HashMap read only in java using methods of Collections class.

### **How To Make ArrayList Read Only In Java?**

Collections.unmodifiableList() method is used to create read only ArrayList in java. Below program demonstrates that modifications to read only list are not allowed and modifications to original list are reflected in read only list also.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72 | import java.util.ArrayList;  import java.util.Collections;  import java.util.List;    public class ReadOnlyList  {      public static void main(String[] args)      {          //Creating an ArrayList            List&lt;String&gt; originalList = new ArrayList&lt;String&gt;();            //Adding elements to originalList            originalList.add("John");            originalList.add("Carlos");            originalList.add("David");            originalList.add("Ian");            originalList.add("Daniel");            //Printing originalList            System.out.println("=========== Original List ===========");            System.out.println(originalList);            //Creating read only view of the originalList            List readOnlyList = Collections.unmodifiableList(originalList);            //Printing readOnlyList            System.out.println("=========== Read Only List ===========");            System.out.println(readOnlyList);            //Modification operations on readOnlyList throws UnsupportedOperationException            try          {              readOnlyList.add("AnyName");                readOnlyList.remove("John");                readOnlyList.set(1, "NameChanged");          }          catch (UnsupportedOperationException e)          {              System.out.println("====== Modification operations on read only list not allowed ======");          }            //Modification operations on originalList are reflected in readOnlyList also            originalList.add("AnyName");            originalList.remove("John");            originalList.set(1, "NameChanged");            //Printing readOnlyList            System.out.println("====== Modifications to original list are reflected in read only list ======");            System.out.println("=========== Read Only List ===========");            System.out.println(readOnlyList);      }  } |

**Output :**  
=========== Original List ===========  
[John, Carlos, David, Ian, Daniel]  
=========== Read Only List ===========  
[John, Carlos, David, Ian, Daniel]  
====== Modification operations on read only list not allowed ======  
====== Modifications to original list are reflected in read only list ======  
=========== Read Only List ===========  
[Carlos, NameChanged, Ian, Daniel, AnyName]

### **How To Make HashSet Read Only In Java?**

Collections.unmodifiableSet() method is used to create read only HashSet in java. Below program demonstrates that you will not be able to perform modification operations on read only set and modifications to original set are reflected in read only set also.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68 | import java.util.Collections;  import java.util.HashSet;  import java.util.Set;    public class ReadOnlySet  {      public static void main(String[] args)      {          //Creating an HashSet            Set&lt;String&gt; originalSet = new HashSet&lt;String&gt;();            //Adding elements to originalSet            originalSet.add("John");            originalSet.add("Carlos");            originalSet.add("David");            originalSet.add("Ian");            originalSet.add("Daniel");            //Printing originalSet            System.out.println("=========== Original Set ===========");            System.out.println(originalSet);            //Creating read only view of the originalSet            Set&lt;String&gt; readOnlySet = Collections.unmodifiableSet(originalSet);            //Printing readOnlySet            System.out.println("=========== Read Only Set ===========");            System.out.println(readOnlySet);            //Modification operations on readOnlySet throws UnsupportedOperationException            try          {              readOnlySet.add("AnyName");                readOnlySet.remove("John");          }          catch (UnsupportedOperationException e)          {              System.out.println("====== Modifications to read only set not allowed ======");          }            //Modification operations on originalSet are reflected in readOnlySet also            originalSet.add("AnyName");            originalSet.remove("John");            //Printing readOnlySet            System.out.println("====== Modifications to original set are reflected in read only set ======");            System.out.println("=========== Read Only set ===========");            System.out.println(readOnlySet);      }  } |

**Output :**  
=========== Original Set ===========  
[Ian, John, David, Daniel, Carlos]  
=========== Read Only Set ===========  
[Ian, John, David, Daniel, Carlos]  
====== Modifications to read only set not allowed ======  
====== Modifications to original set are reflected in read only set ======  
=========== Read Only set ===========  
[Ian, David, Daniel, Carlos, AnyName]

#### **How To Make HashMap Read Only In Java?**

Collections.unmodifiableMap() method is used to create read only HashMap in java. The following program demonstrates that we will not be able to perform modification operations on read only map and any modifications to original map are reflected in read only map.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85 | import java.util.Collections;  import java.util.HashMap;  import java.util.Map;  import java.util.Map.Entry;  import java.util.Set;    public class ReadOnlyMap  {      public static void main(String[] args)      {          //Creating an HashMap            Map&lt;Integer, String&gt; originalMap = new HashMap&lt;Integer, String&gt;();            //Adding elements to originalMap            originalMap.put(1, "John");            originalMap.put(2, "Carlos");            originalMap.put(3, "David");            originalMap.put(4, "Ian");            originalMap.put(5, "Daniel");            //Printing originalMap            System.out.println("=========== Original Map ===========");            Set&lt;Entry&lt;Integer, String&gt;&gt; entrySet = originalMap.entrySet();            for (Entry&lt;Integer, String&gt; entry : entrySet)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }            //Creating read only view of the originalMap            Map&lt;Integer, String&gt; readOnlyMap = Collections.unmodifiableMap(originalMap);            //Printing readOnlyMap            System.out.println("=========== Read Only Map ===========");            entrySet = readOnlyMap.entrySet();            for (Entry&lt;Integer, String&gt; entry : entrySet)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }            //Modification operations on readOnlyMap throws UnsupportedOperationException            try          {              readOnlyMap.put(6, "AnyName");                readOnlyMap.remove(2);          }          catch (UnsupportedOperationException e)          {              System.out.println("====== Modifications to read only map are not allowed ======");          }            //Modification operations on originalMap are reflected in readOnlyMap also            originalMap.put(6, "AnyName");            originalMap.remove(2);            //Printing readOnlyMap            System.out.println("====== Modifications to original map are reflected in read only map also ======");            System.out.println("=========== Read Only Map ===========");            entrySet = readOnlyMap.entrySet();            for (Entry&lt;Integer, String&gt; entry : entrySet)          {              System.out.println(entry.getKey()+" : "+entry.getValue());          }      }  } |

**Output :**

=========== Original Map ===========  
1 : John  
2 : Carlos  
3 : David  
4 : Ian  
5 : Daniel  
=========== Read Only Map ===========  
1 : John  
2 : Carlos  
3 : David  
4 : Ian  
5 : Daniel  
====== Modifications to read only map are not allowed ======  
====== Modifications to original map are reflected in read only map also ======  
=========== Read Only Map ===========  
1 : John  
3 : David  
4 : Ian  
5 : Daniel  
6 : AnyName