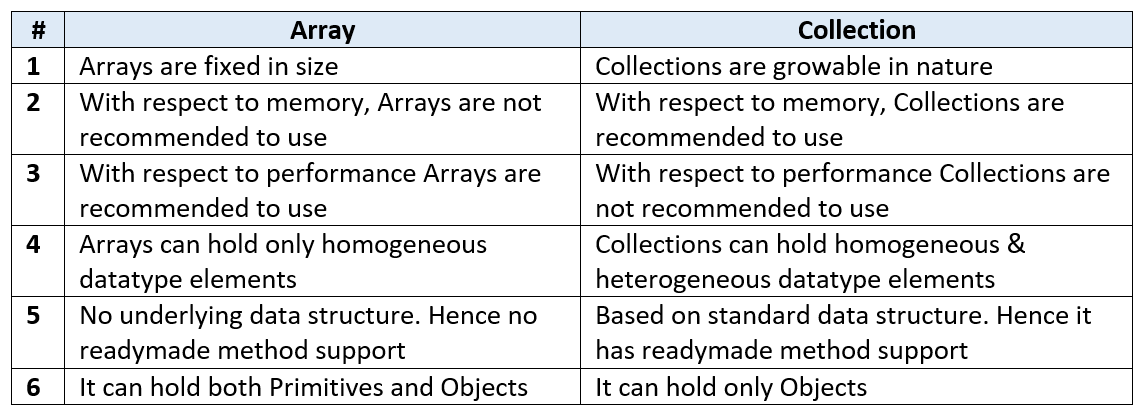
|  |  |
| --- | --- |
| Week7&8 | Collections |



Ex:- Student[] s= new Student[100];

*To overcome above problems of Array, we should go for “****Collection****” concept*

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Operations that can be performed on java collection are searching, sorting, insertion, manipulation, and deletion.

Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList](https://www.javatpoint.com/java-arraylist), Vector, [LinkedList](https://www.javatpoint.com/java-linkedlist), [PriorityQueue](https://www.javatpoint.com/java-priorityqueue), HashSet, LinkedHashSet, TreeSet).

# Legacy Class in Java

In the past decade, the **Collection** framework didn't include in Java. In the early version of Java, we have several classes and interfaces which allow us to store objects. So, classes and interfaces that formed the collections framework in the older version of [Java](https://www.javatpoint.com/java-tutorial) are known as **Legacy classes**.

All the legacy classes are synchronized. The **java.util** package defines the following **legacy** classes:

1. Dictionary
2. HashTable
3. Stack
4. Properties
5. Vector

**Dictionary:-**

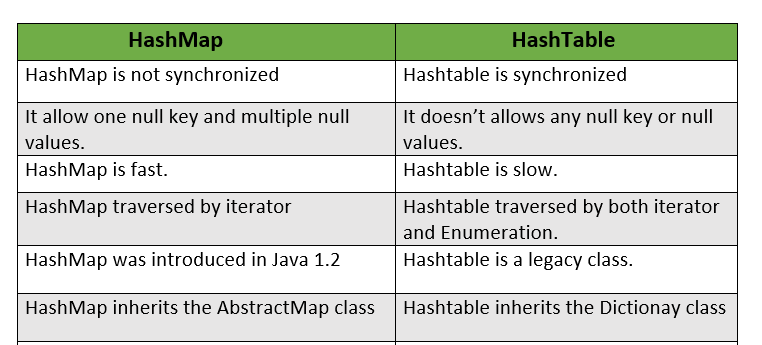
|  |  |
| --- | --- |
| Java **Dictionary** class is an abstract class.  **Java** **Dictionary** class that stores data in key-value pairs  Following are methods of the Dictionary class & all are **abstract** methods.   * put(K key, V value) * keys() * isEmpty() * get(Object key) * elements() * size() * remove() | Dictionary Class in Java |

To create a dictionary in Java is to choose a class that implements **a “key-value pair” interface**.

Dictionary direct subclass is the [**Hashtable**](https://www.javatpoint.com/java-hashtable) class

**INTERVIEW QUESTION:-**

Like HashMap HashTableis also unordered.



**Example:-**

Dictionary dict = new Hashtable();

dict.put("101", "ramesh");

dict.put("102", "madhav");s

for (Enumeration e = dict.elements(); e.hasMoreElements();)

{

System.out.println("Values in dictionary : " + e.nextElement());

}

**Properties Class:-**

The Properties class represents a persistent set of properties.

**Properties** is a subclass of [Hashtable](https://www.geeksforgeeks.org/hashtable-in-java/).

It is used to maintain a list of values in which the **key and value are both strings**.

It is utilized to store data that is to be **changed frequently**. And changes in properties file needs **no recompilation**.

Example:-

// create a reader object on the properties file

        FileReader reader = new FileReader("db.properties");

        // create properties object

        Properties p = new Properties();

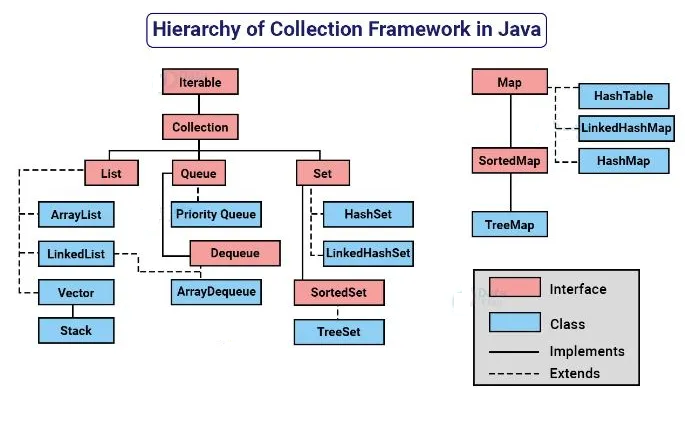
        // Add a wrapper around reader object

        p.load(reader);

        // access properties data

        System.out.println(p.getProperty("username"));

        System.out.println(p.getProperty("password"));



**NOTE:-**Legacy classes are iterated by using **enumerator** interface where as java collection frameworkclasses are iterated by using **iterator** interface

**Iteratable interface**

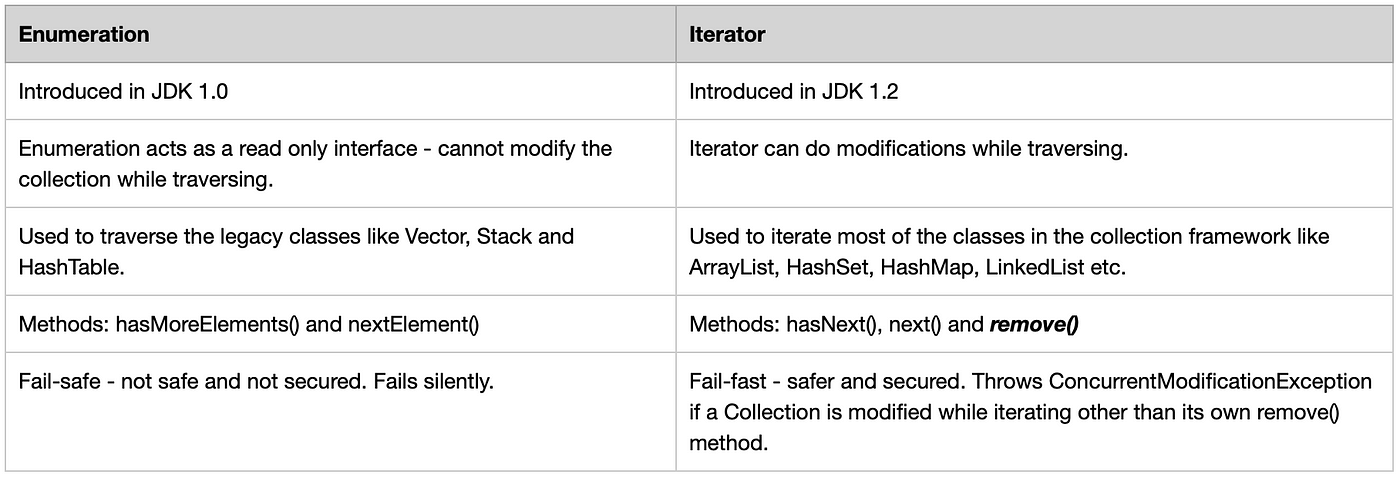
The Iterable interface is present in java.lang.Iterable package. It was introduced in JDK 1.5. It allows users to iterate through elements sequentially from a collection.

The Collection framework extends the Iterable interface, thus all the classes implementing the collections framework also implement the Iterable interface. Thus, objects of these classes can use the Iterable feature of Java.

|  |  |
| --- | --- |
| Any class that implements an Iterable interface, overrides the iterator() method present in the Iterable interface. This iterator() method calls an Iterator interface which then returns an iterator. This iterator is used to iterate over an object of that class.  The Iterator interface has 4 methods namely  next(),  hasNext(),  remove() and  forEachRemaining(). | Java] Iterable과 Iterator :: 그냥 그냥 블로그 |

**INTERVIEW QUESTION:-**

**Diiference between iterator and iteratable**



**Collection interface**

* root interface of Collection Framework
* defines the most common methods which are applicable for any collection object.

add​(E e)

addAll​()

toArray()

size()

containsAll​(Collection<?> c)

equals​(Object o)

hashCode()

isEmpty()

iterator()

parallelStream()

remove​(Object o)

removeAll​(Collection<?> c)

**INTERVIEW QUESTION:-**

**What is the difference between Collection & Collections ❓**

“**Collection**” is an interface. If we want to represent a group of individual objects as a single entity, then we should go for Collection.

“**Collections**” is an utility class present in java.util package to define several utility methods for Collection objects (like sorting, searching, etc).  It defines several utility methods like sorting and searching which is used to operate on collection.

Ex:-

List<Integer> list = new ArrayList<Integer>();

list.add(4);

list.add(5);

list.add(7);

list.add(8);

list.add(2)

Collections.sort(list);

OUTPUT:-

[4, 5, 7, 8, 2]

[2, 4, 5, 7, 8]

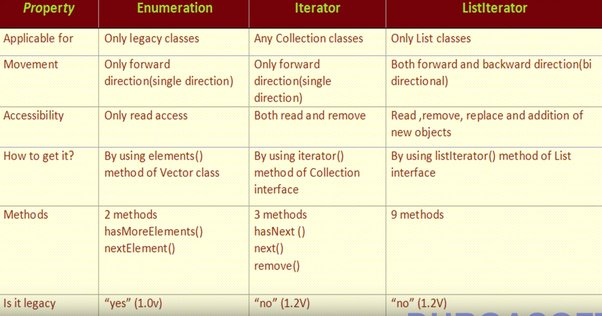
**INTERVIEW QUESTION:-**

**Differentiate List , Set, Map**

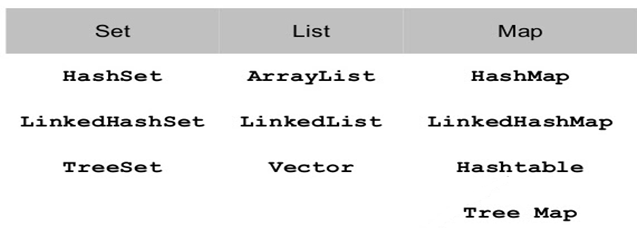
|  |  |  |
| --- | --- | --- |
| LIST | SET | MAP |
| -Allows both Heterogenious & Homogenious data  -Duplicates are allowed  -Multiple nulls allowed  -Insertion order is maintained | -Allows both Heterogenious & Homogenious data  -No Duplicates  -One null allowed  -Does not maintain the order of insertion, it is based on the hash code of the values, the order is confirmed. | -Allows both Heterogenious & Homogenious data  -No Duplicates for keys  -One null for keys & multiple nulls allowed form values  -Does not maintain the order |

**INTERVIEW QUESTION:-**

Differentiate Enumerator, Iterator and ListIterator



**Major Implementations**

****

**INTERVIEW QUESTION:-**

**How to implement collection ordering?**

One of the most ways to sort list data in Java is to use the Collections.sort() method. It sorts a list in ascending order by default.

List<Integer> numbers = new ArrayList<Integer>();

numbers.add(3);

numbers.add(1);

numbers.add(4);

numbers.add(2);

Collections.sort(numbers);

[OR]

Another way to do this, you will need to create a comparator and pass it as an argument to the Collections.sort() method.

A comparator is an object that implements the java.util.Comparator interface. It has a single method called compare() that compares two objects and returns an integer indicating their relative order.

Class person{

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

class PersonComparator implements Comparator<Person> {

public int compare(Person a, Person b) {

return a.getAge() - b.getAge();

}

}

List<Person> people = new ArrayList<>();

people.add(new Person("Alice", 25));

people.add(new Person("Bob", 30));

people.add(new Person("Charlie", 20));

Collections.sort(people, new PersonComparator());

System.out.println("Sorted List: " + people);

**[OR]**

Comparable interface is used to order the objects of the String, wrappers & user-defined class. It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only.

**public int compareTo(Object obj):** It is used to compare the current object with the specified object. It returns

* positive integer, if the current object is greater than the specified object.
* negative integer, if the current object is less than the specified object.
* zero, if the current object is equal to the specified object.

class Student implements Comparable<Student>{

int rollno;

String name;

int age;

Student(int rollno,String name,int age){

this.rollno=rollno;

this.name=name;

this.age=age;

}

public int compareTo(Student st){

if(age==st.age)

return 0;

else if(age>st.age)

return 1;

else

return -1;

}

}

**List:-**

|  |  |
| --- | --- |
| List in Java - Ugt world | * Duplicates are allowed * Multiple nulls allowed * Maintains Insertion order |

The difference is that, the elements in List can be of different types.

You can restrict the type in List by using **List<> instead of List**.

|  |  |
| --- | --- |
| //List allows heterogenious data  List arr1 = new ArrayList();  list.add(1)  list.add("a");  System.out.println(list);  OUTPUT:-  [1, a] | // List<> allows homogenious datas  List<Integer> list = new ArrayList<Integer>();  list.add(1)  list.add(2);  System.out.println(list);  OUTPUT:-  [1, 2] |

**INTERVIEW QUESTION:-**

## Differentiate ArrayList and Vector

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| Not synchronized  ArrayList have default capacity of 10. ArrayList increments the size by 50%.  No way to increment the size  ArrayList uses Iterator interface to iterarte  **Example:** | Synchronized  Vector have the default capacity of 10. Vector will be increment the size by double.  **Vector size can be incremented byusing size(int size)**method  uses both Enumeration and Iterator interfaces to iterate  **Example:** |
| List<Integer> list = new ArrayList<Integer>();  list.add(1);  list.add(2);  List<Integer> l2 = new ArrayList<Integer>();  l2.add(3);  l2.add(4);  l2.add(5);  list.addAll(1, l2);  System.out.println(list.get(2));  list.set(3, 7); // index, object  list.remove(1);  //convert list to array  Object[] arr = list.toArray();  // searching  System.out.println(list.indexOf(2));  System.out.println(list. **lastIndexOf**(2));  System.out.println(list.subList(1,3));  // list traverses in either direction using  Iterator iter = list.listIterator(1);  while (iter.hasNext()) {  System.out.println (iter.next());  }  /\* simillarly for backward iteration hasPrevious()& previous()can be used\*/  // ver. 1 for-loop  for (int i=0; i < list.size() ; i++) {  System.out.println(list[i]);  }  // ver. 2 for-loop  for (Object elem : list)  System.out.println(elem); | Vector vec1 = new Vector();  // Creating Vector with initial capacity 5  Vector vec2 = new Vector(5);  /\* Creating the Vector with initial capacity is 3 and increment value is 2\*/  Vector vec3 = new Vector(3, 2);  vec1.add(200)  vec1.insertElementAt(100, 1)  vec1.contains(900)  vec1.get(4)  vec1.elementAt(4) |

**INTERVIEW QUESTION:-**

## GUESS THE O/P

## If list = [4,5,7,8,2], what is the result of the following code:

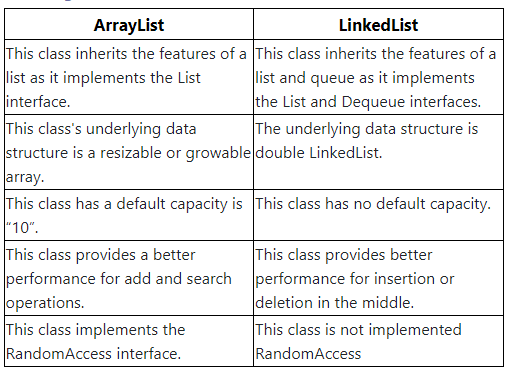
## System.out.println(list.set(4,10));

**Ans:- 2**

public E set(int index, E element)

**Returns Value:** This method returns the **element** previously at the specified position.

**INTERVIEW QUESTION:-**

****

**STACK:-**

|  |  |
| --- | --- |
| Stack Class in Java - GeeksforGeeks | Class, Computer science, Stack | The **Java collections** framework has a class named **Stack** that provides the functionality of the **stack** data structure.  It represents a LIFO (last in, first out) collection of objects allowing for pushing/popping elements  If the number of added elements exceeds the total *Stack*size, it will be doubled utomatically.  However, its size will never shrink after removing elements.  This will **create a Stack with the default capacity of 10.** |

**Example:-**

Stack<String> animals= new Stack<>();

System.out.println("Is the stack empty? " + animals.empty());

animals.push("Dog");

animals.push("Horse");

animals.push("Cat");

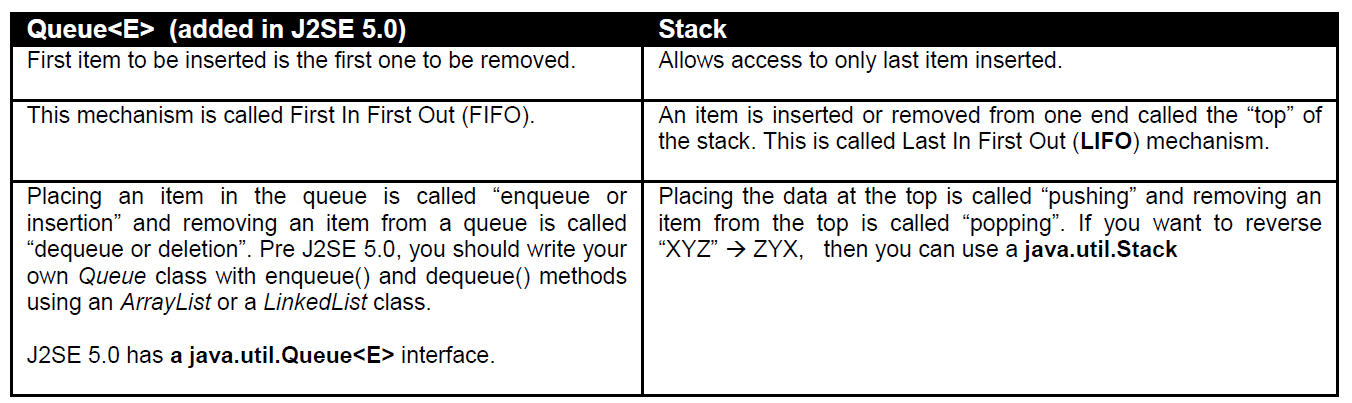
System.out.println("Stack: " + animals);

System.out.println("Removed Element: " + animals.pop());

System.out.println("Element at top: " + animals.peek());

System.out.println("Position of Horse: " + animals.search("Horse"));

**INTERVIEW QUESTION:-**

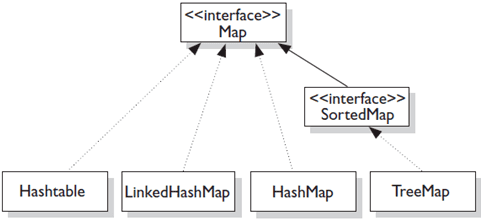
****

**Set:-**

|  |  |
| --- | --- |
| Working with Set in Java like a master | No Duplicates  One null allowed  Doesnot maintain the order of insertion Based on the hash code of the values, the order is confirmed |

|  |  |  |
| --- | --- | --- |
| **HashSet** | **LinkedHashSet** | **TreeSet** |
| Uses Hash Map  Can store heterogenious data  Allows null value  Faster  Does not maintain the insertion order | Uses Linked List  Can store heterogenious data  Allows null value  maintain the insertion order | Uses Tree  Can store Only homogenious data  Doesnot allow null value  Maintains ascendingorder. |
| Set set = new HashSet();  set.add("A");  set.add("M");  set.add("b");  set.add(4);  set.add(1);  set.add(9);  OUTPUT:-  set : [A, 1, b, 4, 9, M] | Set set = new LinkedHashSet();  set.add("A");  set.add("M");  set.add("b");  set.add(4);  set.add(1);  set.add(9);  OUTPUT:-  set : [A, M, b, 4, 1, 9] | Set set = new TreeSet();  set.add(1);  set.add(2);  set.add(4);  set.add(4);  set.add(1);  set.add(9);  OUTPUT:-  set : [1, 2, 4, 9] |

**MAP:-**



A map contains key and value pair

Each key and value pair is known as an entry

|  |  |  |
| --- | --- | --- |
| **HashMap** | **LinkedHashMap** | **TreeMap** |
| Key are unique, but values can be duplicated  Null keys & values allowed  unordered | Key are unique, but values can be duplicated  Null keys & values allowed  Maintains Insertion order | Key are unique, but values can be duplicated  Null keys & values not allowed  Maintains ascending order |

**NOTE**:- A Map can't be traversed, so you need to convert it into Set using **keySet() or entrySet()** method.

**Example:-**

Map<Integer,String> numbers = new HashMap<>();

numbers.put(2,"Two");

numbers.put(1,"One");

numbers.put(4,"Four");

numbers.put(3,"Three");

numbers.put(6,"Six");

numbers.put(5,"Five");

System.out.println("Map: " + numbers);

System.out.println("Keys: " + numbers.keySet());

System.out.println("Values: " + numbers.values());

System.out.println("Entries: " + numbers.entrySet());

System.out.println("Removed Value: " + numbers.remove(2)) ;

//Traversing Map

Set set=numbers.entrySet();//Converting to Set so that we can traverse

Iterator itr=set.iterator();

while(itr.hasNext()){

//Converting to Map.Entry to print key and value separately

Map.Entry entry=(Map.Entry)itr.next();

System.out.println(entry.getKey()+" "+entry.getValue());

}

**INTERVIEW QUESTION:-**

How to check equality of 2 objects?

[OR]

What is the difference between “==” and equals(…) method? What is the difference between shallow comparison and deep comparison of objects?

It is done by using equals(Object) method of java.lang.Object class

Two types of checking the equality

Shallow comparision--- checks whether 2 objects refer to same object

Deep comparition---- classes provid own implementation of equal(Object)method

Example:-

Student s1=newStudent(1,”Ravi”);

Student s2=newStudent(1,”Ravi”);

System.out.println(s1==s2);

// although 2 objects s1 & s2 has same values they are not equal since they have different hashcode.

Some we have to override equals(Object) for checking equality of 2 objects.

Public boolean equals(Object obj){

if(obj==null)

returnfalse;

if(!(obj instanceOf Student))

return false;

if(this.getId()==((Student)obj.getId())

return true;

}

}

implementation of equals() and hashCode() should follow these rules.

* If o1.equals(o2), then o1.hashCode() == o2.hashCode() should always be true.
* If o1.hashCode() == o2.hashCode is true, it doesn’t mean that o1.equals(o2) will be true.

**INTERVIEW QUESTION:-**

Why do you get a **ConcurrentModificationException** when using an iterator?

The java.util Collection classes are fail-fast, which means that if one thread changes a collection while another

thread is traversing it through with an iterator the iterator.hasNext() or iterator.next() call will throw

***ConcurrentModificationException.*** Even the synchronized collection wrapper classes *SynchronizedMap* and *SynchronizedList* are only conditionally thread-safe, which means all individual operations are thread-safe but compound

operations where flow of control depends on the results of previous operations may be subject to threading issues.

**INTERVIEW QUESTION:-**

What is marker interface?

An interface that does not have any methods, fields, or constants, i.e, an empty interface in java is known as Marker or Tag Interface. It is used to deliver type information at runtime to the JVM so that it can take some action based on the information received.

Example:-

Serializable interface is a marker interface. The marker interface provides a hint to the Java runtime that the implementing class allows itself to be serialized.

**INTERVIEW QUESTION:-**

Object cloning?

Class Student implements clonable{

-------

}

Student S1= new Student(1,”Hari”);

Student S2= (Student)S1.clone();