



Objective:

At the end of this lecture, students will understand :

- Collection Framework in Java
- HashMap Class in Java and Java HashMap Constructors
- Methods of HashMap in Java and Its Internal Structure
- Performance of HashMap
- Comparator Interface in Java
- Rules for Using Comparator Interface
- What is Iterators in Java?
- Java Iterators Types: Enumeration, Java Iterator, List Iterator

Collection Framework in Java



Collection Framework in Java:

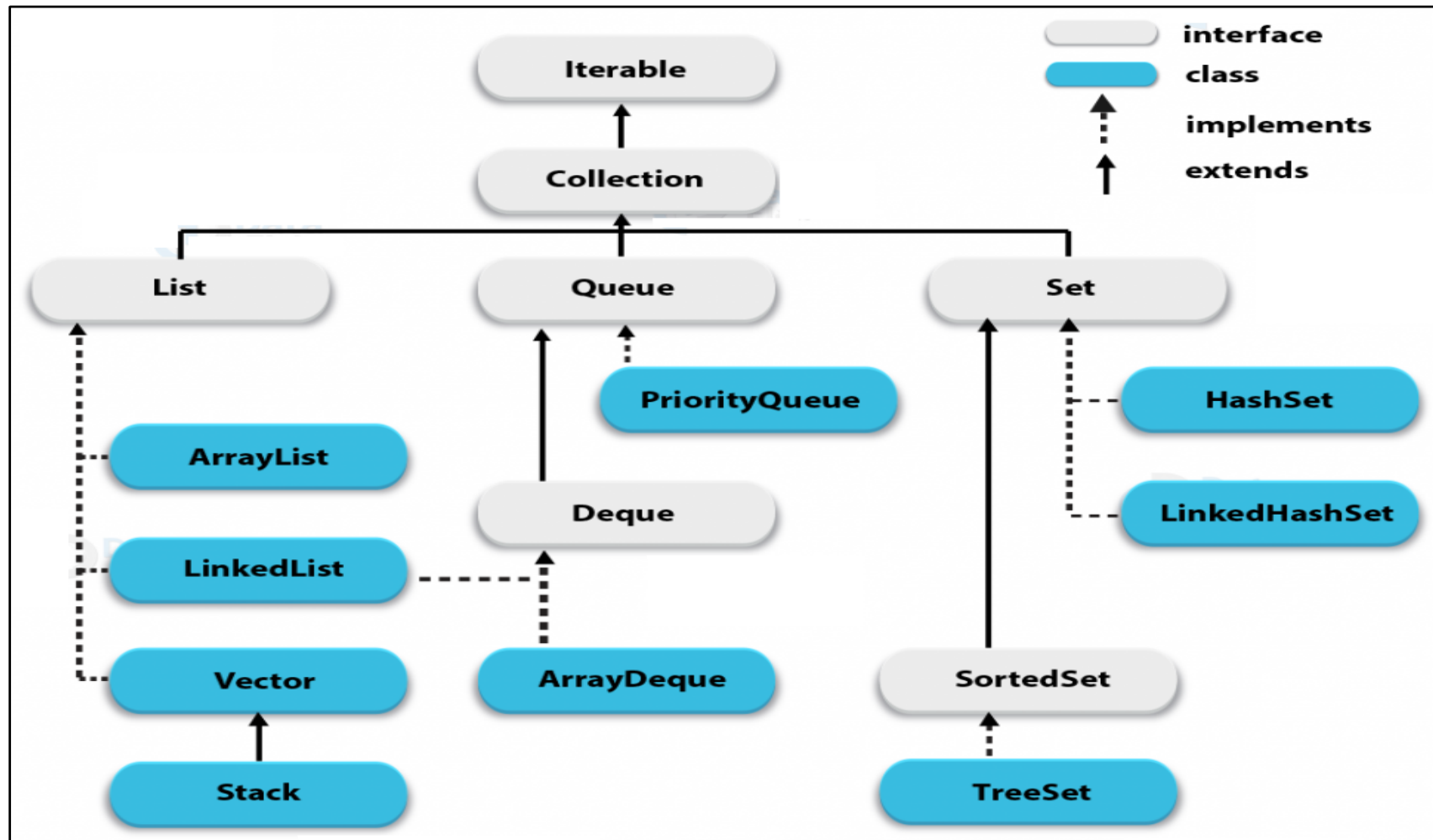
Collections in Java is a readymade architecture with an option to represent classes and objects.

On the other hand, Collection framework in Java represents a better and unified method to store objects and classes.

It has an algorithm, interfaces and implementations.

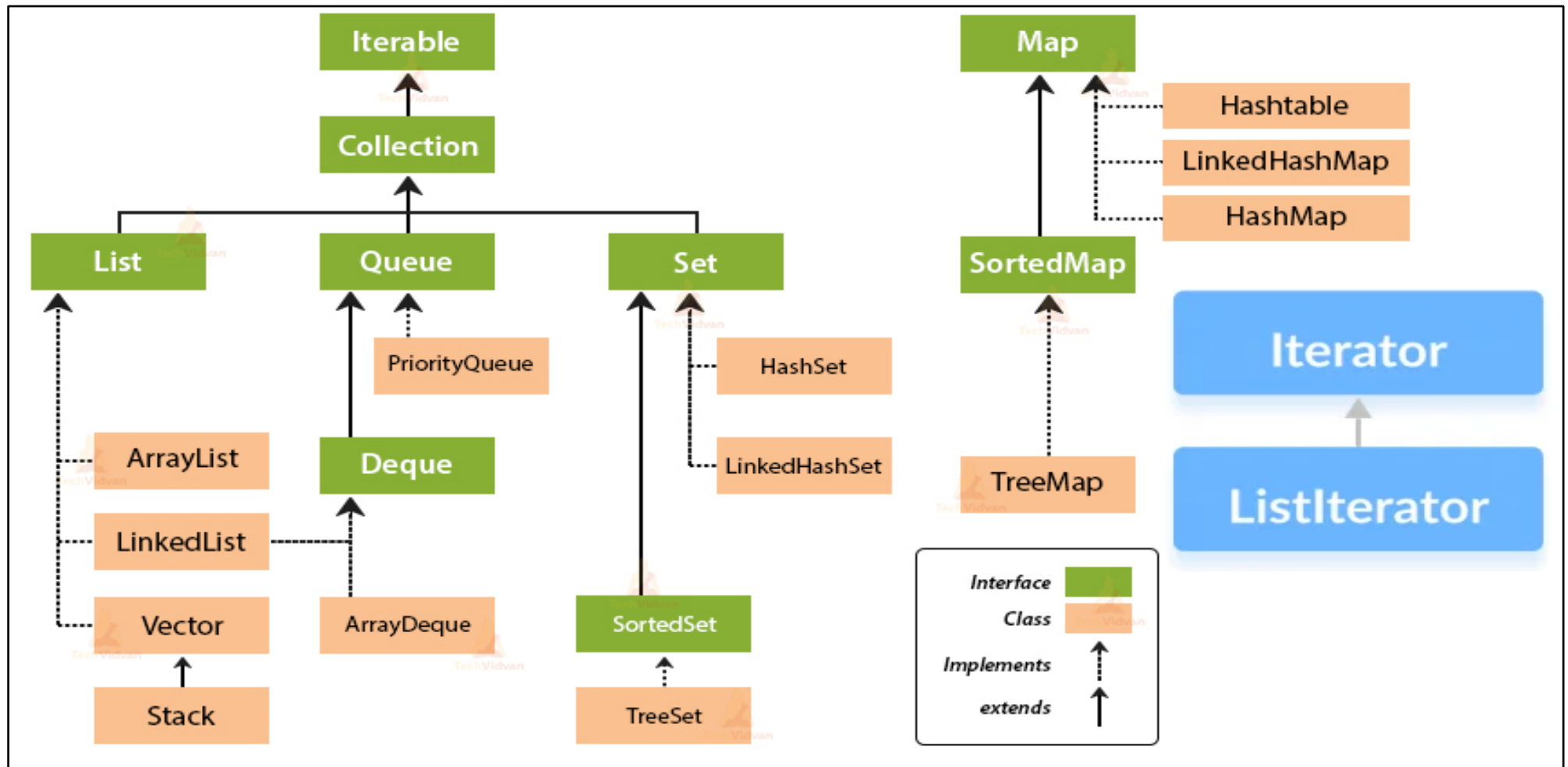


Collection Framework in Java: (Contd.)





Collection Framework in Java: (Contd.)



HashMap Class in Java



What is Java HashMap?

- Java HashMap permits null key and null values.
- HashMap isn't an ordered collection. You'll be able to iterate over HashMap entries through keys set however they're not guaranteed to be within the order of their addition to the HashMap.
- HashMap in Java uses its inner category Node for storing map entries.

An **internal node** (also known as an **inner node**, **inode** for short, or **branch node**) is any node of a tree that has child nodes.



What is Java HashMap? (Contd.)

- Java Hashmap stores entries into multiple one by one linked lists which are called buckets or bins. Default range of bins is 16 and it's always a power of 2.
- HashMap uses hashCode() and equals() ways on keys to get and place operations. Therefore HashMap key object should offer good implementation of those ways. This can be the reason immutable classes are higher suitable for keys, for example, String and Integer.



What is Java HashMap? (Contd.)

- Java HashMap isn't thread-safe, for the multithreaded environment, you must use ConcurrentHashMap category or get a synchronous map using `Collections.synchronizedMap()` method.



What is Java HashMap? (Contd.)





Java HashMap Constructors:

HashMap in Java provides four constructors. So, below we are discussing 4 constructors in Java HashMap:

a. public HashMap(): Most commonly used HashMap constructor. This constructor can produce an empty HashMap with default initial capacity 16 and load factor 0.75

*The Load factor is a measure that decides when to increase the HashMap capacity to maintain the get() and put() operation complexity of $O(1)$. The default load factor of HashMap is **0.75f** (75% of the map size).*



Java HashMap Constructors: (Contd.)

- b. `public HashMap(int initialCapacity)`:** This HashMap constructor is used to specify the initial capacity and 0.75 ratio. This can help in avoiding rehashing if you know the number of mappings to hold HashMap.
- c. `public HashMap(int initialCapacity, float loadFactor)`:** This HashMap constructor can produce an empty HashMap with specified initial capacity and load factor.



Java HashMap Constructors: (Contd.)

d. public HashMap(Map<? extends K, ? extends V> m):

Creates a Map having same mappings because of the specified map and with load factor 0.75.

Examples are:

```
Map map1 = new HashMap<>();
```

```
Map map2 = new HashMap<>(2^5);
```

```
Map map3 = new HashMap<>(32, 0.80f);
```

```
Map map4 = new HashMap<>(map1);
```



Methods of HashMap in Java:

1. Important methods of Java HashMap

Let's have a look at the important methods of Java HashMap:

- a. public void clear():** This Java HashMap method will remove all the mappings and HashMap will become empty.
- b. public boolean containsKey(Object key):** This Hashmap in Java method returns 'true' if the key exists otherwise it will return 'false'.



Methods of HashMap in Java: (Contd.)

- c. public boolean containsValue(Object value):** This Java HashMap method returns true if the value exists.
- d. public Set entrySet():** This Hashmap Method in Java returns a set view of the HashMap mappings.
- e. public V get(Object key):** This Java Hashmap Method Returns the value mapped to the specified key, or null if there's no mapping for the key.



Methods of HashMap in Java: (Contd.)

f. public boolean isEmpty(): A utility method returning true if no key-value mappings are present.

g. public Set keySet(): Returns a set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected within the set, and vice-versa.



Methods of HashMap in Java: (Contd.)

h. public V put(K key, V value): This method associates the specified value with the specified key in this map and if the map previously contained a mapping for the key, the old value is replaced.

i. public void putAll(Map<? extends K, ? extends V> m):
Copies all of the mappings from the required map to the present map. These mappings can replace any mappings that this map had for any of the keys currently within the specified map.



Methods of HashMap in Java: (Contd.)

j. public V remove(Object key): This Java HashMap method helps remove the mapping for the required key from this map if present. The method returns the value that was previously mapped to the specified key if the key exists else the method returns NULL.

k. public int size(): This method in Hashmap in Java helps return the number of key-value mappings in this map.



Methods of HashMap in Java: (Contd.)

I. public Collection values(): This method of Hashmap in Java helps returns a collection view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa.



Methods of HashMap in Java: (Contd.)

2. New Methods of Java HashMap

Many new methods in Java HashMap introduced in Java 8, let us have a look at them –

a. `public V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)`: If the specified key is not already associated with a value (or is mapped to null), this method tries to reason its value using the given mapping function and therefor enters it into the HashMap unless it is Null.



Methods of HashMap in Java: (Contd.)

b. public V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction): In this method, if the value for the required key is present and non-null, it, therefore, tries to reason a replacement mapping given the key and its current mapped value.

c. public V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction): This HashMap method attempts to compute a mapping for the required key and its current mapped value.



Methods of HashMap in Java: (Contd.)

d. public void forEach(BiConsumer<? super K, ? super V> action): This method performs the given action for each entry in this map.

e. public V getOrDefault(Object key, V defaultValue): Same as get except that defaultValue is returned if no mapping found for the specified key.



Methods of HashMap in Java: (Contd.)

f. public V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction): If the required key is not already related to a value or is associated with null, associates it with the given non-null value.

g. public V putIfAbsent(K key, V value): In this method, if the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns this price.



Methods of HashMap in Java: (Contd.)

- h. public boolean remove(Object key, Object value):** In this method, it removes the entry for the specified key only if it's currently mapped to the specified value.
- i. public boolean replace(K key, V oldValue, V newValue):** In this method it helps to replace the entry for the specified key only if currently mapped to the specified value.
- j. public V replace(K key, V value):** In this method, it helps to replace the entry for the specified key only if it currently map to some value.



Methods of HashMap in Java: (Contd.)

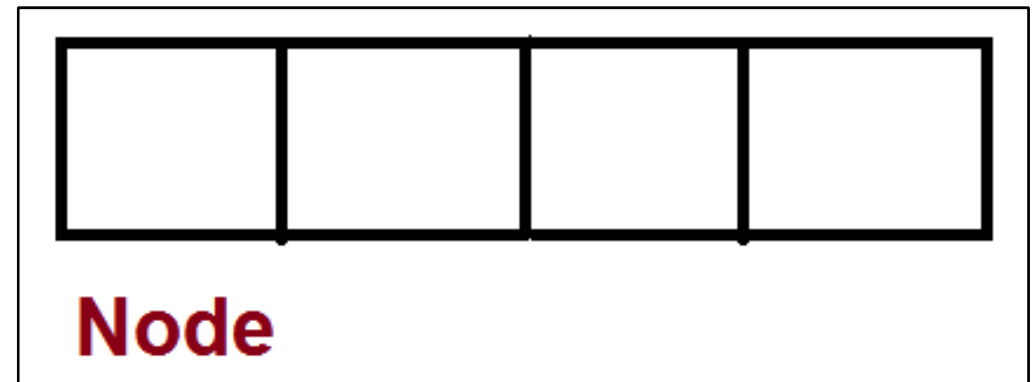
k. public void replaceAll(BiFunction<? super K, ? super V, ? extends V> function): In this method, it helps replace every entry's value with the result of invoking the given function on that entry.



Java HashMap – Internal Structure:

Internal Structure of Java Hashmap contains these four nodes of the array, and further, the node represents with the help of a class –

- i. int hash**
- ii. K key**
- iii. V value**
- iv. Node next**



A node contains a reference to its own object and hence it is a linked list.



Performance of HashMap:

Performance of Java HashMap depends on these two factors:

i. Initial Capacity

ii. Load Factor

In a Java HashMap, the capacity simply defines as the number of buckets, while the Initial capacity of HashMap in Java define when it we create it initially. Further, we multiply capacity with 2.



Performance of HashMap: (Contd.)

In a Load Factor, a measure of much rehashing is to do. It is also a measure of how much rehashing to do. It is initially kept higher so rehashing doesn't take place, but this also increases the iteration time. The most common load factor value is 0.75. It varies from 0 - 1.



Map Interface: HashMap (Sample Code)

```
package com.collections;
import java.util.*;
public class HashMapDemo {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        print(map);
        map.put("Monday", 10);
        map.put("Sunday", 30);
        map.put("Tuesday", 20);
        System.out.println("Size of map is:- " + map.size());
        print(map);
        if (map.containsKey("Monday")) {
            Integer a = map.get("Monday");
            System.out.println("value for key \"Monday\" is:- " + a);
        }
        map.clear();
        print(map);
    }
    public static void print(Map<String, Integer> map) {
        if (map.isEmpty()) {
            System.out.println("map is empty");
        }
        else {
            System.out.println(map);
        }
    }
}
```

Output:


map is empty
Size of map is:- 3
{Monday=10, Sunday=30,
Tuesday=20}
value for key "Monday"
is:- 10
map is empty

Comparator Interface in Java



Comparator Interface in Java:

Comparator Interface in Java



- `public int compare`
- `public boolean equals`
- `static <T> Comparator<T> nullsFirst`
- `static <T> Comparator<T> nullsLast`



Comparator Interface in Java: (Contd.)

Working of Collections.Sort()

When we work with user-defined classes in Java, there may have been some kind of a need to arrange data in a proper way. However since the data is user-defined, there is absolutely no way for inbuilt functions to arrange these elements. Moreover, if there is a specific condition for arranging these data, then we need to have a different method inside the class. To avoid these kinds of problems, Java has an interface known as the Comparator Interface.



Comparator Interface in Java: (Contd.)

When you need to order the data items of a user-defined data, i.e., classes, then you can use the comparator interface. It can prove to be fruitful when you need a data item based on a specific condition that would differ from user to user.

There are following primary methods in the Java Comparator Interface:



Comparator Interface in Java: (Contd.)

- **public int compare(Object ob1, Object ob2)** – This function has the responsibility of comparing any two objects and returns an integer value based on it. This method returns values -1, 0, 1 to say that it is either less than, equal to, or greater than the other object.
- **public boolean equals(Object obj)** – This method is particularly useful for checking whether the current object is exactly equal to the second object.



Comparator Interface in Java: (Contd.)

- **static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator)** – This function essentially compares objects. However, it takes into consideration that nulls are lesser than non-null elements.
- **static <T> Comparator<T> nullsLast(Comparator<? super T> comparator)** – This is similar to the nullsFirst comparator except for a fundamental difference. This method takes into consideration that nulls are greater than non-null elements in the collection.



How does the Sort Method in the Collection Sort Work?

- Whenever we need to sort the values in a collection, this “sort” method transfers control to the compare method in the class.
- The compare method then returns some values based on the comparison.
- It returns 0 if both the objects are equal.



How does the Sort Method in the Collection Sort Work? (Contd.)

- This returns 1 if the first object is greater than the second.
- It returns -1 if the second object is greater than the first.
- Using these values the function decides whether to swap the values for the sorting process.

Let us go for a program demonstration for its implementation.



Basic Differences between Comparable and Comparator in Java:

- The comparable interface can only be useful for sorting the elements in a class in a single way. However, the comparator interface is useful for sorting multiple types of data in a class.
- While using comparable Interfaces, the class itself has to implement the interface. However, we may choose to implement or not implement the Comparator interface in the base class. You can take advantage of anonymous classes in Java to leverage this issue.



Basic Differences between Comparable and Comparator in Java: (Contd.)

- While using the Comparable interface, we do not need to make any changes to the code. This is because the sort functions of the collections class automatically use the compareTo method in the class. However, while we implement the Comparator interface, we need to use the comparator name along with the sort function.
- The java.lang houses the Comparable interface whereas the java.util package contains the Comparator interface. You have to be cautious while using interfaces because you would run into errors if you do not import the correct package.



Rules for Using Comparator Interface:

There are certain rules that you should keep in mind before actually attempting to implement an interface. Some of them are:

- You can use the Comparator interface to sort the elements of the collection.
- If you keep your object type as undefined while implementing the Comparator Interface, then you need to make sure that you typecast all the variables inside it to the Object inside the method.



Rules for Using Comparator Interface: (Contd.)

- You need to specify the generic of the user datatype while passing it to a comparator. If you do not then java converts it to an Object type by default. You would not be able to compare individual elements after that.

Iterator Interface in Java



What is Iterators in Java?

Basically, Java Iterators are used in Java for retrieving the elements one by one, they are used in Collection Framework and allow us:

- 1. To traverse the collection**
- 2. Access the Data Elements**
- 3. Delete the Element**



Types of Java Iterators: Enumeration

a. Enumeration

It is an interface used to get components of inheritance collections (Vector, Hashtable). Enumeration is the principal Java iterator introduced from JDK 1.0, later incorporated into JDK 1.2 with greater usefulness. Enumerations are additionally used for determining the information streams to a `SequenceInputStream`. We can make Enumeration object by calling `elements()` method for vector class on any vector object.



Types of Java Iterators: Enumeration (Sample Code)

```
package com.collections;
import java.util.*;
public class IteratorDemo1 {
    public static void main(String[] args) {
        Vector v = new Vector();
        for (int i = 0; i < 10; i++)
            v.addElement(i);
        System.out.println(v);
        Enumeration e = v.elements();
        while (e.hasMoreElements()) {
            int i = (Integer)e.nextElement();
            System.out.print(i + " ");
        }
    }
}
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
```



Limitations of Enumeration:

- It is for a legacy class, and hence not a universal Java iterator.
- The operations of removing cannot be performed.
- Only forward iteration is possible.



Types of Java Iterators: Java Iterator

b. Java Iterator

It is a general (universal) Java iterator as we can apply it to any collection object. By utilizing Java Iterator, we can perform both read and remove operations. It is an enhanced variant of Enumeration with the extra usefulness of expelling capacity of an element.



Types of Java Iterators: Java Iterator (Contd.)

Iterator must be utilized at whatever point we need to identify elements in all collection structure actualized interfaces like Set, List, Queue, Deque and furthermore in every single executed class of Map interface. Generally, Java iterator is the main cursor accessible for whole collection Framework.

We can make an Iterator operator by calling `iterator()` strategy show in Collection interface.



Types of Java Iterators: Java Iterator (Contd.)

Iterator must be utilized at whatever point we need to identify elements in all collection structure actualized interfaces like Set, List, Queue, Deque and furthermore in every single executed class of Map interface. Generally, Java iterator is the main cursor accessible for whole collection Framework.

We can make an Iterator operator by calling `iterator()` strategy show in Collection interface.



Types of Java Iterators: Java Iterator (Sample Code)

```
package com.collections;
import java.util.*;
public class IteratorDemo2 {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        for (int i = 0; i < 10; i++)
            al.add(i);
        System.out.println(al);
        Iterator itr = al.iterator();
        while (itr.hasNext()) {
            int i = (Integer)itr.next();
            System.out.print(i + " ");
            if (i % 2 != 0)
                itr.remove();
        }
        System.out.println();
        System.out.println(al);
    }
}
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
[0, 2, 4, 6, 8]
```



Limitations of Java Iterator:

There are some limitations of Java Iterator. Some of them are -

- Iteration of only forward direction is possible.
- Replacement and addition not possible.



Types of Java Iterators: List Iterator:

This is only applicable where there is List implemented classes such as ArrayList, linked list etc. It provides the user to give bi-directional iteration, it is used when we want to enumerate the list, it has more functionality.

Generally, we can create the list by using `listIterator()` method, present in List interface.

`ListIterator ltr = l.listIterator();`

It extends the Java iterator interface.



Types of Java Iterators: List Iterator (Sample Code)

```
package com.collections;
import java.util.*;
public class IteratorDemo3 {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        for (int i = 0; i < 10; i++)
            al.add(i);
        System.out.println(al);
        ListIterator ltr = al.listIterator();
        while (ltr.hasNext()) {
            int i = (Integer)ltr.next();
            System.out.print(i + " ");
            if (i%2==0) {
                i++;
                ltr.set(i);
                ltr.add(i);
            }
        }
        System.out.println();
        System.out.println(al);
    }
}
```

Output:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

0 1 2 3 4 5 6 7 8 9

[1, 1, 1, 3, 3, 3, 5, 5, 5, 7, 7, 7, 9, 9, 9]

Quiz!



1. Which interface does `java.util.Hashtable` implement?

- A) `Java.util.Map`
- B) `Java.util.List`
- C) `Java.util.HashTable`
- D) `Java.util.Collection`



1. Which interface does `java.util.Hashtable` implement?

- ☒ A) `Java.util.Map`
- ☐ B) `Java.util.List`
- ☐ C) `Java.util.HashTable`
- ☐ D) `Java.util.Collection`

Explanation:

Hash table based implementation of the Map interface.



2. Which of these return type of hasNext() method of an iterator?

- A) Integer
- B) Double
- C) Boolean
- D) Collections Object



2. Which of these return type of hasNext() method of an iterator?

- A) Integer
- B) Double
- ☒ C) Boolean
- D) Collections Object

Explanation:
hasNext() returns boolean values true or false.



3. Which of these methods is used to obtain an iterator to the start of collection?

- A) start()
- B) begin()
- C) iteratorSet()
- D) iterator()



3. Which of these methods is used to obtain an iterator to the start of collection?

- A) start()
- B) begin()
- C) iteratorSet()
- ☒ D) iterator()

Explanation:

To obtain an iterator to the start of the collection we use iterator() method.



4. Which of these is a method of ListIterator used to obtain index of previous element?

- A) previous()
- B) previousIndex()
- C) back()
- D) goBack()



4. Which of these is a method of ListIterator used to obtain index of previous element?

- A) previous()
- ☒ B) previousIndex()
- C) back()
- D) goBack()

Explanation:
previousIndex() returns index of previous element. if there is no previous element then -1 is returned.



5. What will be the output of the following Java program?

```
import java.util.*;
class Collection_iterators {
    public static void main(String args[]) {
        LinkedList list = new LinkedList();
        list.add(new Integer(2));
        list.add(new Integer(8));
        list.add(new Integer(5));
        list.add(new Integer(1));
        Iterator i = list.iterator();
        Collections.reverse(list);
        while(i.hasNext())
            System.out.print(i.next() + " ");
    }
}
```

A) 2 8 5 1

B) 1 5 8 2

C) 2

D) 2 1 8 5



5. What will be the output of the following Java program?

```
import java.util.*;
class Collection_iterators {
    public static void main(String args[]) {
        LinkedList list = new LinkedList();
        list.add(new Integer(2));
        list.add(new Integer(8));
        list.add(new Integer(5));
        list.add(new Integer(1));
        Iterator i = list.iterator();
        Collections.reverse(list);
        while(i.hasNext())
            System.out.print(i.next() + " ");
    }
}
```

A) 2 8 5 1

✓ B) 1 5 8 2

C) 2

D) 2 1 8 5

Explanation:

Collections.reverse(list) reverses the given list, the list was 2->8->5->1 after reversing it became 1->5->8->2.

Output:



6. What will be the output of the following Java program?

```
import java.util.*;
class Collection_iterators
{
    public static void main(String args[])
    {
        LinkedList list = new LinkedList();
        list.add(new Integer(2));
        list.add(new Integer(8));
        list.add(new Integer(5));
        list.add(new Integer(1));
        Iterator i = list.iterator();
        Collections.reverse(list);
        Collections.sort(list);
        while(i.hasNext())
            System.out.print(i.next() + " ");
    }
}
```

A) 2 8 5 1

B) 1 5 8 2

C) 1 2 5 8

D) 2 1 8 5



6. What will be the output of the following Java program?

```
import java.util.*;
class Collection_iterators
{
    public static void main(String args[])
    {
        LinkedList list = new LinkedList();
        list.add(new Integer(2));
        list.add(new Integer(8));
        list.add(new Integer(5));
        list.add(new Integer(1));
        Iterator i = list.iterator();
        Collections.reverse(list);
        Collections.sort(list);
        while(i.hasNext())
            System.out.print(i.next() + " ");
    }
}
```

A) 2 8 5 1

B) 1 5 8 2

 C) 1 2 5 8

D) 2 1 8 5

Explanation:

Collections.sort(list) sorts the given list, the list was 2->8->5->1 after sorting it became 1->2->5->8.



7. Which of these methods can convert an object into a List?

- A) SetList()
- B) ConvertList()
- C) singletonList()
- D) CopyList()



7. Which of these methods can convert an object into a List?

- A) SetList()
- B) ConvertList()
- ☒ C) singletonList()
- D) CopyList()

Explanation:

singletonList() returns the object as an immutable List. This is an easy way to convert a single object into a list. This was added by Java 2.0.



8. What happens if we put a key object in a HashMap which exists?

- A) The new object replaces the older object
- B) The new object is discarded
- C) The old object is removed from the map
- D) It throws an exception as the key already exists in the map



8. What happens if we put a key object in a HashMap which exists?

- ✓ A) The new object replaces the older object
- B) The new object is discarded
- C) The old object is removed from the map
- D) It throws an exception as the key already exists in the map

Explanation:

HashMap always contains unique keys. If same key is inserted again, the new object replaces the previous object.



9. While finding the correct location for saving key value pair, how many times the key is hashed?

- A) 1
- B) 2
- C) 3
- D) unlimited till bucket is found



9. While finding the correct location for saving key value pair, how many times the key is hashed?

A) 1

☒ B) 2

C) 3

D) unlimited till bucket is found

Explanation:

The key is hashed twice; first by hashCode() of Object class and then by internal hashing method of HashMap class.



10. Is hashmap an ordered collection.

A) True

B) False



10. Is hashmap an ordered collection.

A) True

 B) False

Explanation:

**HashMap outputs in the order of
hashcode of the keys. So it is unordered
but will always have same result for same
set of keys.**



11. What will be the output of the following Java code snippet?

```
public class Demo {  
    public static void main(String[] args) {  
        Map<Integer, Object> sampleMap = new TreeMap<Integer, Object>();  
        sampleMap.put(1, null);  
        sampleMap.put(5, null);  
        sampleMap.put(3, null);  
        sampleMap.put(2, null);  
        sampleMap.put(4, null);  
        System.out.println(sampleMap);  
    }  
}
```

- A) {1=null, 2=null, 3=null, 4=null, 5=null}
- B) {5=null}
- C) Exception is thrown
- D) {1=null, 5=null, 3=null, 2=null, 4=null}



11. What will be the output of the following Java code snippet?

```
public class Demo {  
    public static void main(String[] args) {  
        Map<Integer, Object> sampleMap = new TreeMap<Integer, Object>();  
        sampleMap.put(1, null);  
        sampleMap.put(5, null);  
        sampleMap.put(3, null);  
        sampleMap.put(2, null);  
        sampleMap.put(4, null);  
        System.out.println(sampleMap);  
    }  
}
```

Explanation: HashMap needs unique keys. TreeMap sorts the keys while storing objects.

- ☒ A) {1=null, 2=null, 3=null, 4=null, 5=null}
- B) {5=null}
- C) Exception is thrown
- D) {1=null, 5=null, 3=null, 2=null, 4=null}



12. How is `Arrays.asList()` different than the standard way of initialising List?

- A) Both are same
- B) `Arrays.asList()` throws compilation error
- C) `Arrays.asList()` returns a fixed length list and doesn't allow to add or remove elements
- D) We cannot access the list returned using `Arrays.asList()`



12. How is `Arrays.asList()` different than the standard way of initialising List?

A) Both are same

B) `Arrays.asList()` throws compilation error

✓ C) `Arrays.asList()` returns a fixed length list and doesn't allow to add or remove elements

D) We cannot access the list returned using `Arrays.asList()`

Explanation:

List returned by `Arrays.asList()` is a fixed length list which doesn't allow us to add or remove element from it. `add()` and `remove()` method will throw `UnsupportedOperationException` if used.



13. What is the worst case complexity of accessing an element in ArrayList?

- A) $O(n)$
- B) $O(1)$
- C) $O(n \log n)$
- D) $O(2)$



13. What is the worst case complexity of accessing an element in ArrayList?

A) $O(n)$

 B) $O(1)$

C) $O(n \log n)$

D) $O(2)$

Explanation:

ArrayList has $O(1)$ complexity for accessing an element in ArrayList. $O(n)$ is the complexity for accessing an element from LinkedList.



14. When an array is passed to a method, will the content of the array undergo changes with the actions carried within the function?

B) True

A) False



14. When an array is passed to a method, will the content of the array undergo changes with the actions carried within the function?

- ☒ B) True
- ☐ A) False

Explanation:

If we make a copy of array before any changes to the array the content will not change. Else the content of the array will undergo changes.



15. What is the difference between `length()` and `size()` of `ArrayList`?

- A) `length()` and `size()` return the same value
- B) `length()` is not defined in `ArrayList`
- C) `size()` is not defined in `ArrayList`
- D) `length()` returns the capacity of `ArrayList` and `size()` returns the actual number of elements stored in the list



15. What is the difference between `length()` and `size()` of `ArrayList`?

- A) `length()` and `size()` return the same value
- B) `length()` is not defined in `ArrayList`
- C) `size()` is not defined in `ArrayList`
- ✓ D) `length()` returns the capacity of `ArrayList` and `size()` returns the actual number of elements stored in the list

Explanation:

`length()` returns the capacity of `ArrayList` and `size()` returns the actual number of elements stored in the list which is always less than or equal to capacity.