JAVA BASICS

By
Sudha Agarwal

INDEX

- AutoBoxing
- Unboxing
- Wrapper Classes
- Packages
- Collections

AUTOBOXING

❖**Autoboxing**: Converting a primitive value into an object of the corresponding wrapper class is called autoboxing.

❖For example, converting int to Integer class. The Java compiler applies autoboxing when a primitive value is:

- ❖Passed as a parameter to a method that expects an object of the corresponding wrapper class.
- ❖Assigned to a variable of the corresponding wrapper class.

UNBOXING

❖**Unboxing**: Converting an object of a wrapper type to its corresponding primitive value is called unboxing.

❖For example conversion of Integer to int. The Java compiler applies unboxing when an object of a wrapper class is:

- ❖Passed as a parameter to a method that expects a value of the corresponding primitive type.
- ❖Assigned to a variable of the corresponding primitive type.

Wrapper Classes

❖A Wrapper class is a class whose object wraps or contains a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

NEED OF WRAPPER CLASSES

❖They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).

❖The classes in java.util package handles only objects and hence wrapper classes help in this case also.

❖Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.

❖An object is needed to support synchronization in multithreading.

WRAPPER CLASSES

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
long	Integer
float	Float
double	Double
boolean	Boolean

PACKAGES

❖A java package is a group of similar types of classes, interfaces and sub-packages.

❖Package in java can be categorized in two form, built-in package and user-defined package.

❖There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

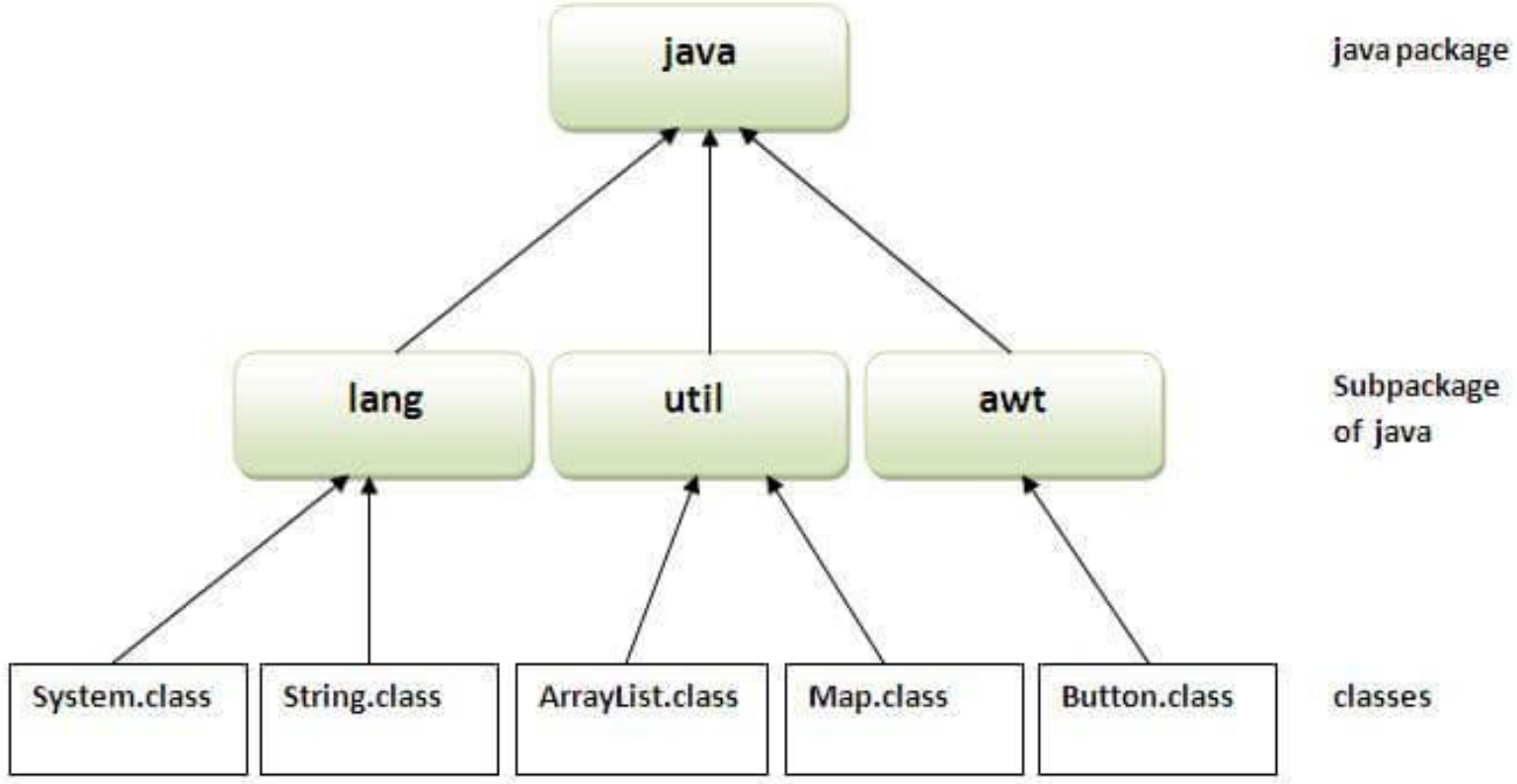
Advantage of Java Package

❖1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

❖2) Java package provides access protection.

❖3) Java package removes naming collision.

PACKAGES


java
java package
lang
util
awt
Subpackage
of java
System.class
String.class
ArrayList.class
Map.class
Button.class
classes

PACKAGES

❖Packages are declared using keyword ‘package’. They should be declared in the first statement in a java file. If you try to declare packages at any other statements, you will get compile time error.

```
package com;
class A
{
     //Some statements
}
//package com; If you declare here, it gives compile time error
```

PACKAGES

❖Only alphabets, numbers and an underscore are allowed in naming the packages. By convention, names of package should start with lowercase although it is not a condition. Package name should start with a alphabets or underscore but not with a number.

```
package javaConcept;       //Valid package name
package java_Concept;      //Valid package name
package java_12;           //Valid package name
package 12_java;           //Invalid package name, should not start with a number.
package _java12            //Valid package Name
package JAVA;             //Valid package name but not recommended.
```

PACKAGES

❖When you declare a package name in your java file, and after compiling it with -d option, a folder with the same name is created in the specified location and all generated .class files will be stored in that folder.

```
To Compile: javac -d . Simple.java
To Run: java mypack.Simple
```

PACKAGES

- There are three ways to access the package from outside the package.

- import package.*;
- import package.classname;
- fully qualified name.

USING PACKAGENAME.*

❖If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

❖The import keyword is used to make the classes and interface of another package accessible to the current package.

```
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

USING PACKAGENAME.CLASSNAME

❖If you import package.classname then only declared class of this package will be accessible.

```
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

USING FULLY QUALIFIED NAME

❖If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

❖It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

```
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

SUBPACKAGES

❖Package inside the package is called the subpackage. It should be created to categorize the package further.

```
package com.java.core;
class Simple{
  public static void main(String args[]){
   System.out.println("Hello subpackage");
  }
}
```

❖**To Compile**: javac -d . Simple.java

❖**To Run**: java com.javatpoint.core.Simple

COLLECTIONS

❖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.

COLLECTION FRAMEWORK

❖Collection Framework in java is a centralized and unified theme to store and manipulate the group of objects.

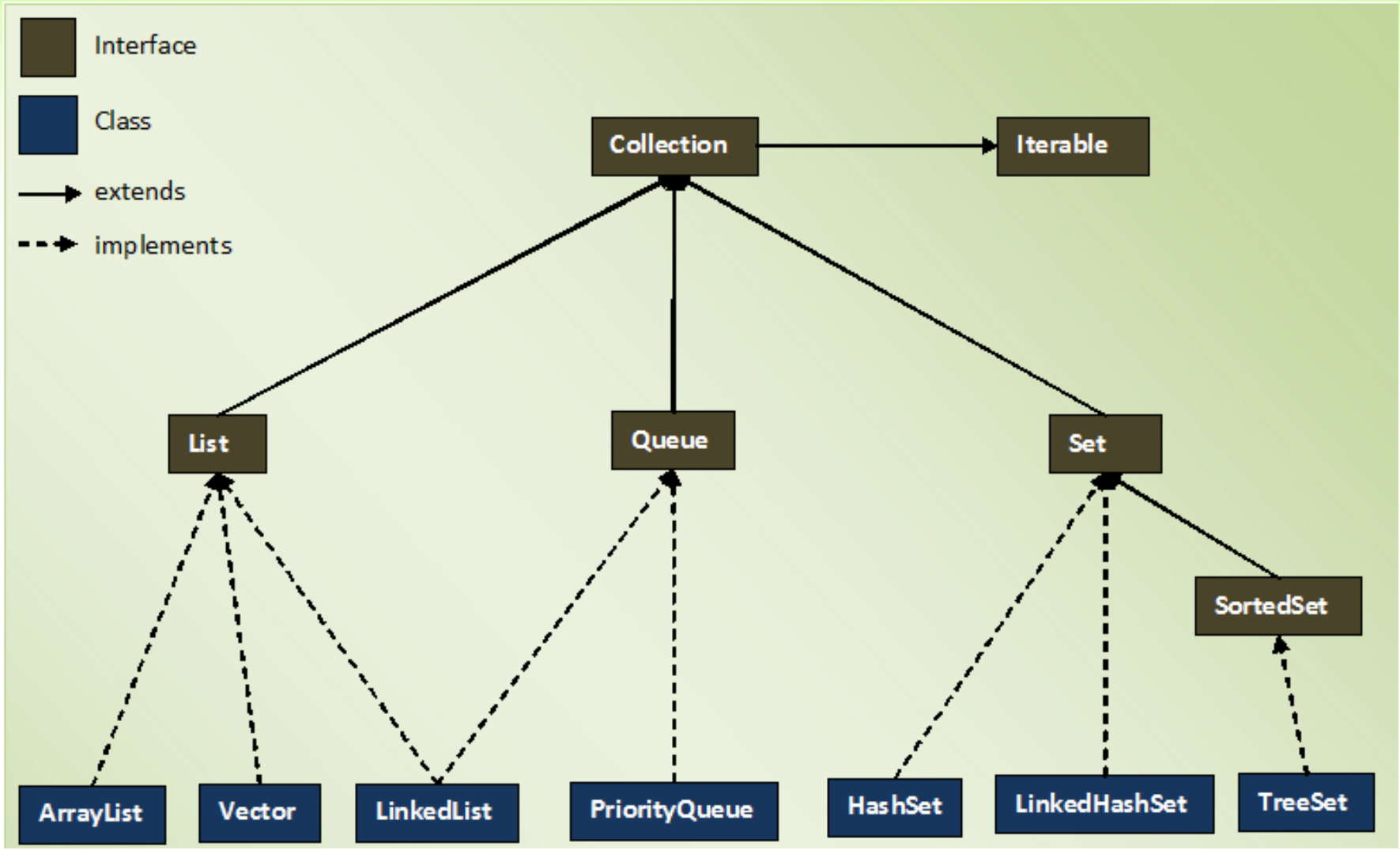
❖Java Collection Framework provides some pre-defined classes and interfaces to handle the group of objects.

❖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.

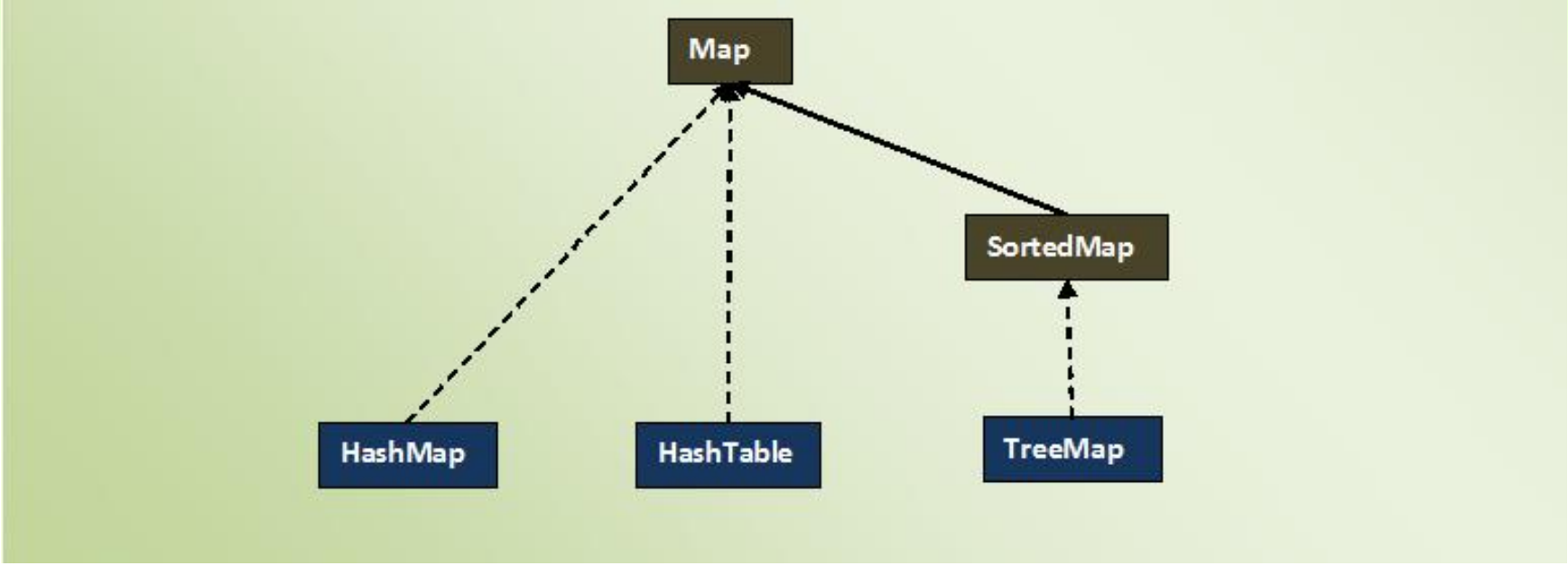
COLLECTION FRAMEWORK HIERARCHY

❖All classes and interfaces related to Collection Framework are placed in **java.util** package. **java.util.Collection** class is at the top of class hierarchy of Collection Framework.

COLLECTION FRAMEWORK HIERARCHY



COLLECTION FRAMEWORK HIERARCHY



COLLECTION FRAMEWORK HIERARCHY

❖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 special list of objects in which elements are removed only from the head. **LinkedList** and **PriorityQueue** classes implement this interface.

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

COLLECTION FRAMEWORK HIERARCHY

❖4) **Map** —> This is the one interface in Collection Framework which is not inherited from Collection interface. It handles 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**.

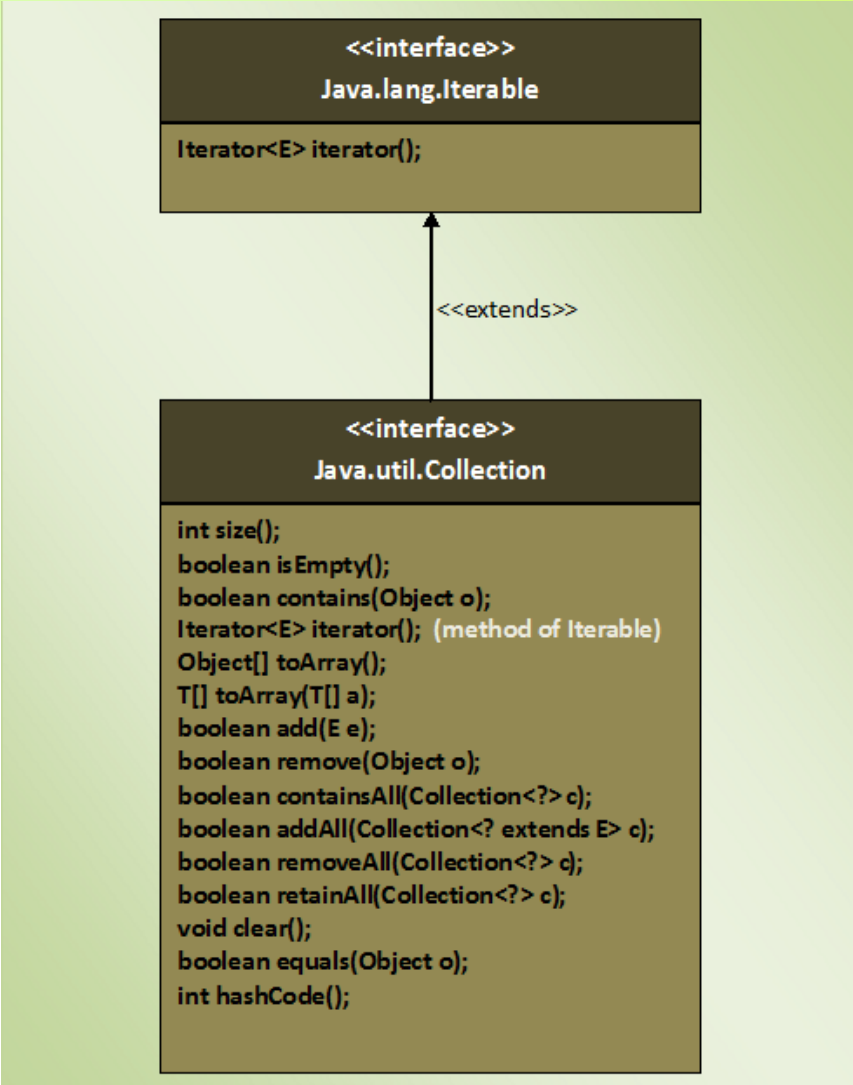
❖Three of above interfaces (List, Queue and Set) inherit from Collection interface. Although, Map is included in collection framework it does not inherit from Collection interface.

COLLECTION INTERFACE

❖Collection interface is the root level interface in the collection framework. List, Queue and Set are all sub interfaces of Collection interface.

❖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.

COLLECTION INTERFACE



COLLECTION INTERFACE METHODS

boolean add(E obj)	Used to add objects to a collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates.
boolean addAll(Collection C)	Add all elements of collection C to the invoking collection. Returns true if the element were added. Otherwise, returns false.
boolean remove(Object obj)	To remove an object from collection. Returns true if the element was removed. Otherwise, returns false.
boolean removeAll(Collection C)	Removes all element of collection C from the invoking collection. Returns true if the collection's elements were removed. Otherwise, returns false.

COLLECTION INTERFACE METHODS

boolean contains(Object obj)	To determine whether an object is present in collection or not. Returns true if obj is an element of the invoking collection. Otherwise, returns false.
boolean isEmpty()	Returns true if collection is empty, else returns false.
int size()	Returns number of elements present in collection.
void clear()	Removes total number of elements from the collection.

COLLECTION INTERFACE METHODS

boolean retainAll(Collection c)	Deletes all the elements of invoking collection except the specified collection.
Iterator iterator( )	Returns an iterator for the invoking collection.
boolean equals(Object obj)	Returns true if the invoking collection and obj are equal. Otherwise, returns false.
Object[] toArray(Object array[])	Returns an array containing only those collection elements whose type matches of the specified array.
boolean equals(Object o)	Compares the specified object with this collection for equality.
int hashCode()	Returns the hash code value of this collection.

COLLECTION INTERFACE

❖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 sub classes of 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.

Why Collections were made Generic ?

❖Generics added type safety to Collection framework.

❖Earlier collections stored Object class references which meant any collection could store any type of object.

❖ Hence there were chances of storing incompatible types in a collection, which could result in run time mismatch.

❖ Hence Generics was introduced through which you can explicitly state the type of object being stored.

COLLECTIONS AND AUTOBOXING

❖Autoboxing converts primitive types into Wrapper class Objects.

❖As collections doesn't store primitive data types(stores only refrences), hence Autoboxing facilitates the storing of primitive data types in collection by boxing it into its wrapper type.

EXCEPTIONS IN COLLECTIONS

Exception Name	Description
UnSupportedOperationException	occurs if a Collection cannot be modified
ClassCastException	occurs when one object is incompatible with another
NullPointerException	occurs when you try to store null object in Collection
IllegalArgumentException	thrown if an invalid argument is used
IllegalStateException	thrown if you try to add an element to an already full Collection

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.

LIST INTERFACE

- 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

E get(int index)	Returns element at the specified position.
E set(int index, E element)	Replaces an element at the specified position with the passed element.
void add(int index, E element)	Inserts passed element at a specified index.
E remove(int index)	Removes an element at specified index.
int indexOf(Object o)	It returns an index of first occurrence of passed object.
int lastIndexOf(Object o)	It returns an index of last occurrence of passed object.

LIST INTERFACE

ListIterator<E> listIterator()	It returns a list iterator over the elements of this list.
ListIterator<E> listIterator(int index)	Returns a list iterator over the elements of this list starting from the specified index.
List<E> subList(int fromIndex, int toIndex)	Returns sub list of this list starting from ‘fromIndex’ to ‘toIndex’.

Advantages of using arraylist over arrays

❖Drawbacks of arrays are:

❖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.

ARRAYLIST CLASS

❖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.

❖ 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**.

MODIFICATION OPERATIONS

❖**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.

BULK MODIFICATION OPERATIONS

❖**boolean addAll(Collection c)** : This method appends all elements of the passed collection at the end of this list.

❖**boolean addAll(int index, Collection 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.

Iterator Vs. ListIterator

❖**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.

❖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.

Iterator Vs. ListIterator

❖ 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().

LIST ITERATOR 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.

LIST ITERATOR METHODS

❖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.

Iterating an ArrayList in Java

- Iteration Using Normal for loop.
- Iteration Using Iterator Object.
- Iteration Using ListIterator Object.
- Iteration Using Enhanced for loop.

LINKEDLIST CLASS

❖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 can have any type of elements including null and duplicates.

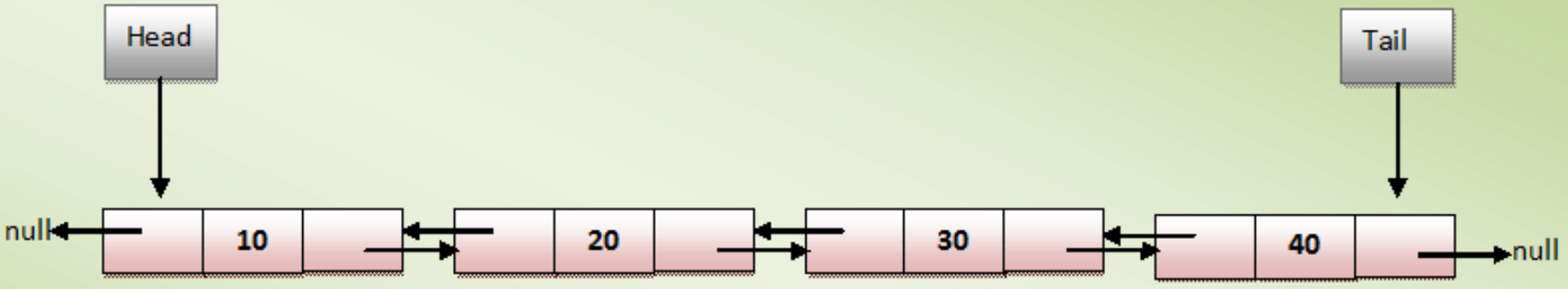
❖Elements can be inserted and can be removed from both the ends and can be retrieved from any arbitrary position.

LINKEDLIST CLASS

- LinkedList has two constructors.
- LinkedList() - It creates an empty LinkedList
- LinkedList(Collection C) - It creates a LinkedList that is initialized with elements of the Collection c

LinkedList Class - Properties

❖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.



Insertions and Removals in LinkedList are faster than ArrayList. Because, You don't need to resize LinkedList after each insertions and removals.

Retrieval operations in LinkedList are slow compared to ArrayList. Because, it needs traversing from the beginning or from end to reach the element.

LinkedList Class - Methods

❖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
❖addFirst(E e)
❖offerFirst(E e)

Insertion In The Middle
add(int index, E e)
addAll(int index, Collection c)

Insertion At Tail
add(E e)
addAll(Collection c)
offer(E e)
offerLast(E e)

LinkedList Class - Methods

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

❖Removing from head
❖poll()
❖pollFirst()
❖remove()
❖removeFirst()

Removing from middle
Remove(int index)

Removing from tail
pollLast()
removeLast()

LinkedList Class - Methods

❖You can retrieve the elements form the head, from the middle and from the tail of the LinkedList.

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

LINKED LIST

❖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.

LINKED LIST

❖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.

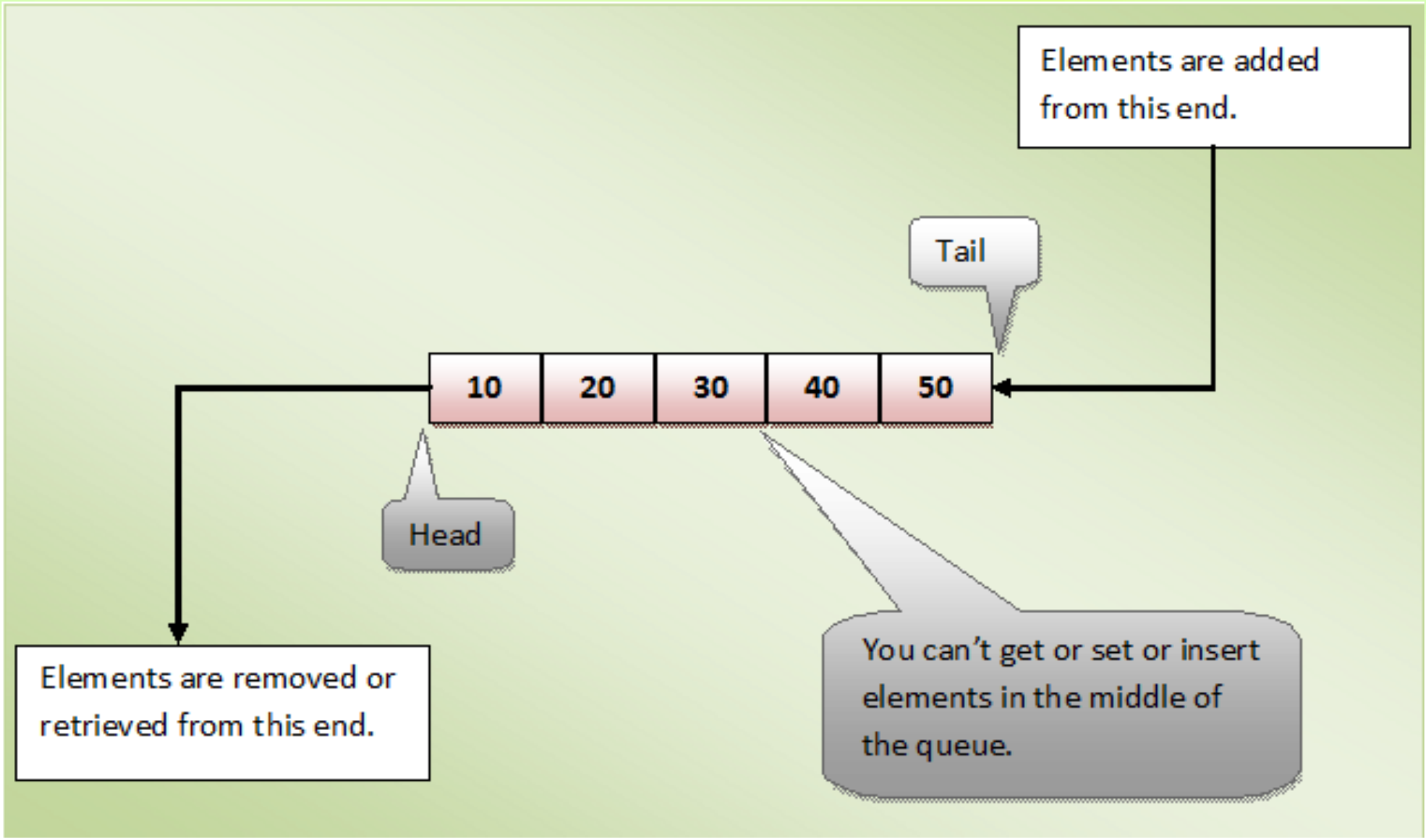
QUEUE INTERFACE

❖Queue interface 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.

QUEUE INTERFACE



Queue Interface - Properties

❖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.

QUEUE INTERFACE - PROPERTIES

❖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.

PRORITYQUEUE CLASS

❖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.

SET INTERFACE

- 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.

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.

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**.

HashSet Class - Properties

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

❖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.

LINKEDHASHSET CLASS

❖The LinkedHashSet in java is an ordered version of HashSet which internally maintains one **doubly linked li**st 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.

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.

METHODS OF SORTEDSET INTERFACE

Comparator<? super E> comparator()	Returns Comparator used to order the elements. If no comparator is supplied, it returns null.
SortedSet<E> subSet(E fromElement, E toElement)	Returns a portion of this set whose elements range from ‘fromElement’ (Inclusive) and ‘toElement’ (Exclusive).
SortedSet<E> headSet(E toElement)	Returns a SortedSet whose elements are in the range from first element of the set (Inclusive) to ‘toElement’ (exclusive).
SortedSet<E> tailSet(E fromElement)	Returns a SortedSet whose elements are in the range from ‘fromElement’ (Inclusive) to last element of the set (exclusive).
E first()	Returns first element of the SortedSet.
E last()	Returns last element of the SortedSet.

SortedSet Interface - Properties

❖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.

❖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.

TREESET CLASS

❖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.

❖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.

❖TreeSet does not allow even a single null element.

❖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.

TREESET CLASS

- Constructors:
- **TreeSet()** - It is used to construct an empty tree set that will be sorted in an ascending order according to the natural order of the tree set.
- **TreeSet(Collection c)** - It is used to build a new tree set that contains the elements of the collection c.
- **TreeSet(Comparator comp)** - It is used to construct an empty tree set that will be sorted according to given comparator.
- **TreeSet(SortedSet ss)** - It is used to build a TreeSet that contains the elements of the given SortedSet.

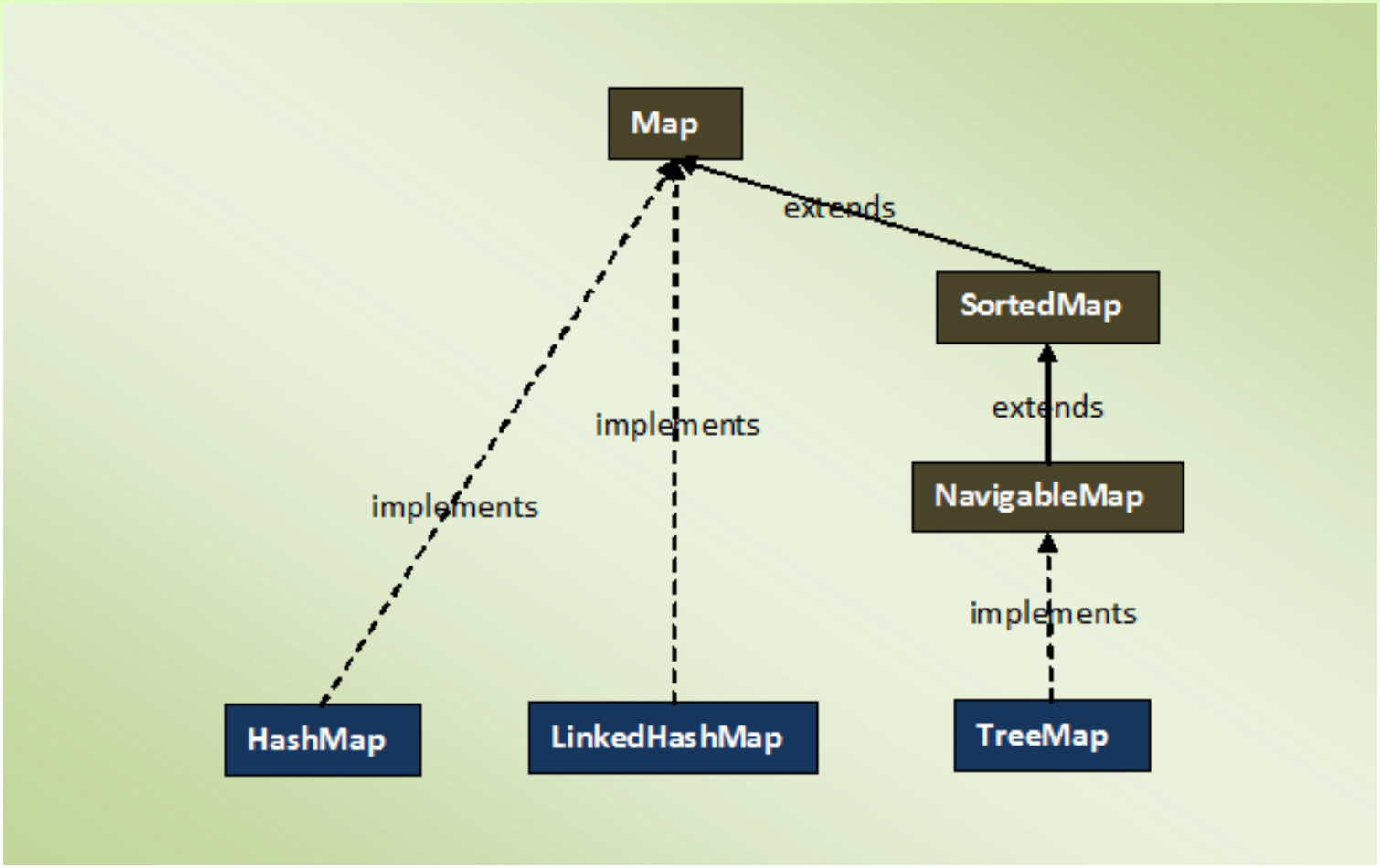
MAP INTERFACE

❖The **Map** interface in java is one of the four top level interfaces of Java Collection Framework along with List, Set and Queue interfaces.

❖But, unlike others, it doesn’t inherit from Collection 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.

MAP INTERFACE



MAP INTERFACE

❖ 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**.

MAP INTERFACE

❖ 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).**

HashMap Class - Constructor

- **HashMap()** - It is used to construct a default HashMap.

- **HashMap(Map m)** - It is used to initializes the hash map by using the elements of the given Map object m.

- **HashMap(int capacity)** - It is used to initializes the capacity of the hash map to the given integer value, capacity.

- **HashMap(int capacity, float fillRatio)** - It is used to initialize both the capacity and fill ratio of the hash map by using its arguments.

HashMap Class- Methods

❖**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.

HashMap Class- Methods

❖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.

HashMap Class- Methods

❖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.

HashMap Class- Methods

❖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.

HashMap Vs. HashTable

❖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.

❖ashTable.

HashMap Vs. HashTable

❖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.

HashMap Vs. 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.

HashCode() and Equals()

❖An object of a class that override the equals() method can be used as elements in a collection.

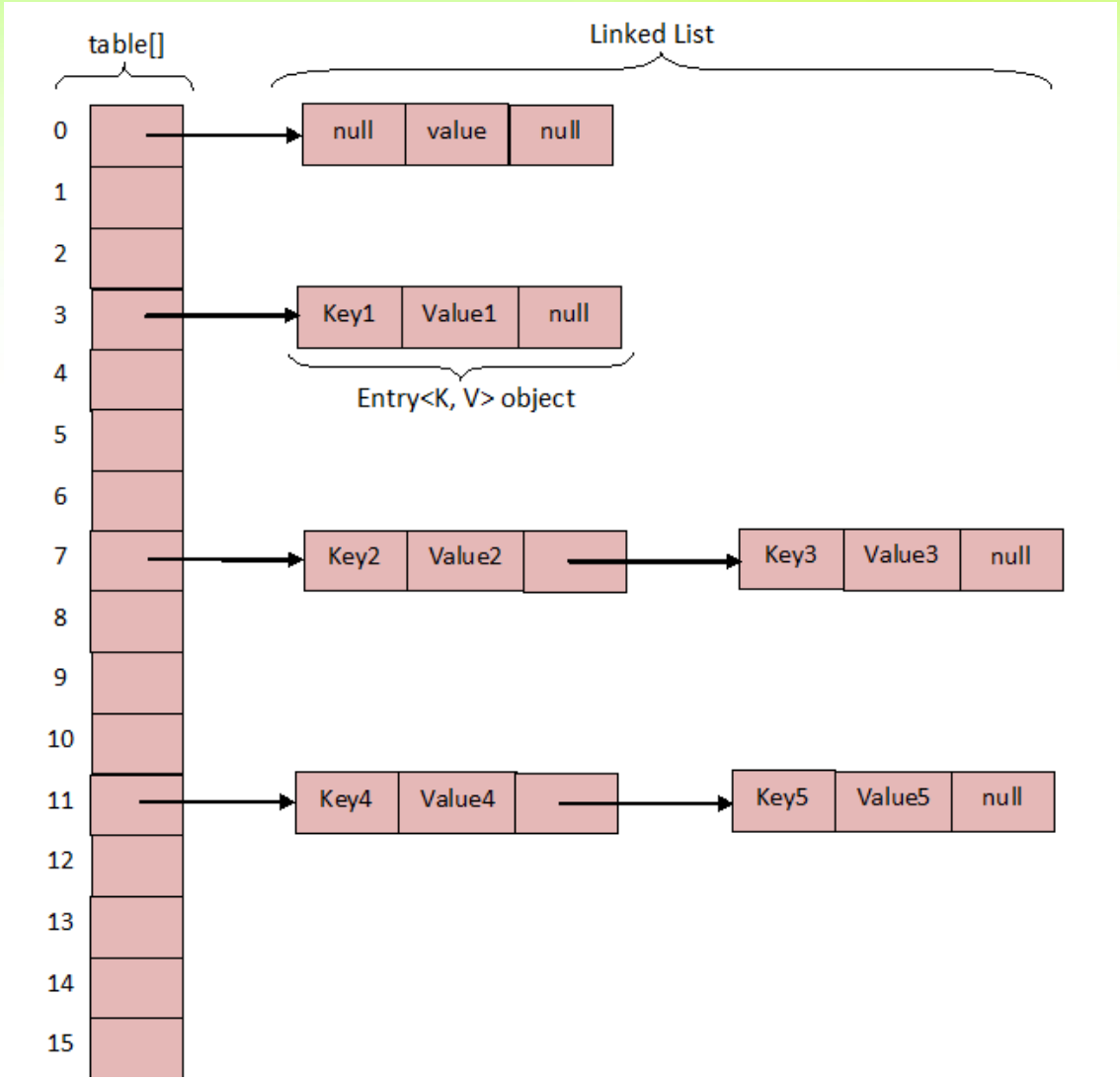
❖And if they override the hashCode() method also, they can be used as elements as keys in a HashMap.

HASHCODE() AND EQUALS()

❖If two objects are equal according to the **equals(Object)** method, then calling the **hashCode** method on each of the two objects must produce the same integer result.

❖It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results.

HASHMAP INTERNAL STRUCTURE



COLLECTIONS UTILITY 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.

COLLECTIONS

- What is Collection's Hierarchy?
- What is the difference between Collection and Collections?
- Why Map interface does not extend Collection interface?
- What is the difference between comparable and comparator?
- Which methods you need to override to use any object as key in HashMap ?
- What is the importance of hashCode() and equals() methods?
- How can we sort a list of Objects?
- What is the difference between Fail- fast iterator and Fail-safe iterator

COLLECTIONS

- There are two objects a and b with same hashcode. I am inserting these two objects inside a hashmap.

- hMap.put(a,a);
- hMap.put(b,b);

- where a.hashCode()==b.hashCode()
- Now tell me how many objects will be there inside the hashmap?

LAB 6

- Q1. Write a Java program to create a new array list, add some colors (string) and print out the collection.
- Q2. Write a Java program to iterate through all elements in a array list.
- Q3. Write a Java program to insert an element into the array list at the first position.
- Q4. Write a Java program to get the number of elements in a hash set.
- Q5. Write a Java program to test a hash set is empty or not.
- Q6. Write a Java program to iterate a HashMap.
- Q7. Write a Java program to count the number of key-value (size) mappings in a map
- Q8. Write a program to remove duplicate elements from ArrayList

LAB 6

- Q1. Create a class Employee with following fields:
  - EmpId
  - Name
  - Salary
- Write a program to create a Hashmap which stores unique Employee objects.
- Create a utility which performs following tasks:
  - search function which searches for a particular employee based on their EmpId and print the employee details.
  - Add a new Employee
  - Delete an existing employee
  - Modify the details based on the EmpId

LAB 6

❖Q2. Write a program to read a text file and print the file content.

❖Q3. Write a program to read a text file containing numbers from 1 to 10 in any order and store the content in an Arraylist.

❖Q4. write a program to sort the above created Arraylist and remove duplicate elements.