Limitations of an Array

- It can't contain elements of different types.
- It can not grow or reduce dynamically.
- These limitations are covered in collection.

The Java Collection Framework

The Java Collections Framework is a library of classes and interfaces for working with collections of objects.

A collection is an object which can store other objects, called elements. Collections provide methods for adding and removing elements, and for searching for a particular element within the collection.

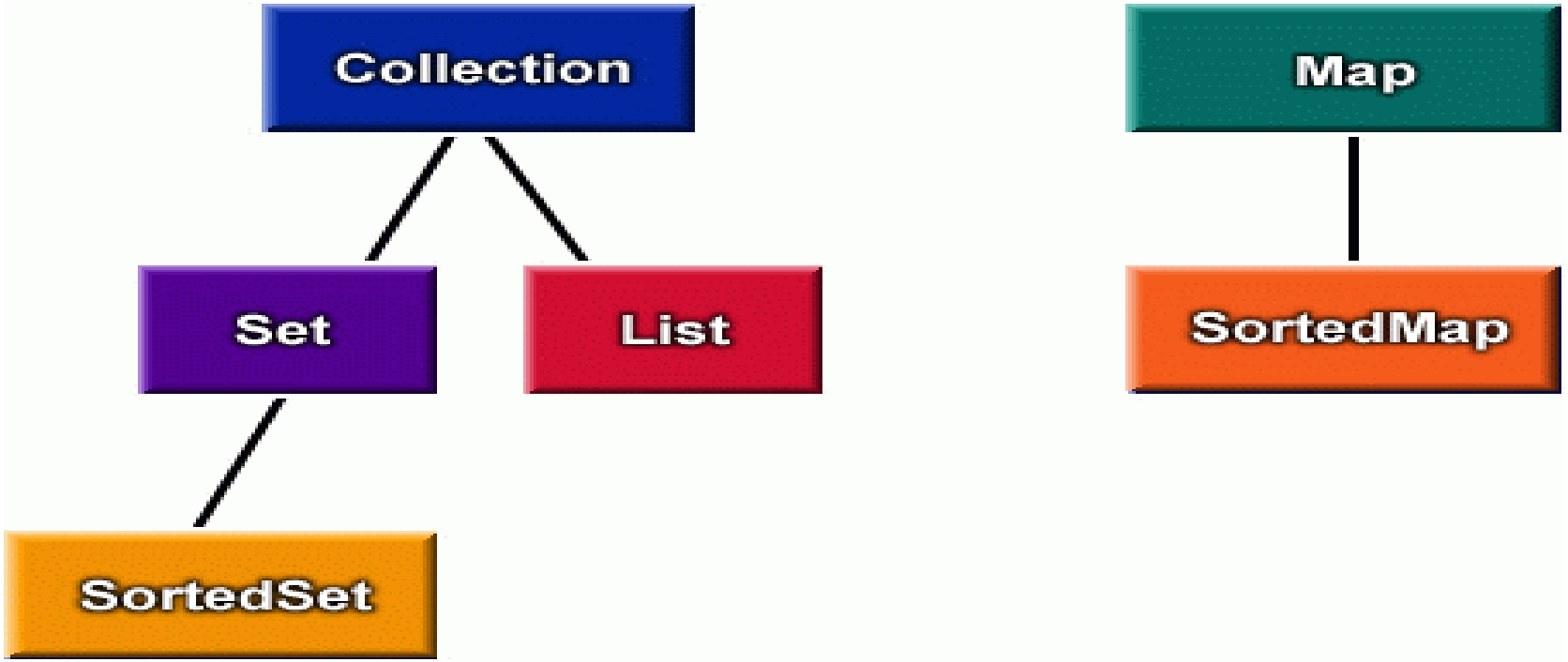
COLLECTION

- It is a group of objects treated as a single unit.
- It can not contains object of different types.
- Grow and reduce dynamically.
- Collections are defined in java.util

Collection Framework

- It is a well defined **architecture** which provides inbuilt **interfaces**, **classes**, **methods** to perform operation on collection.

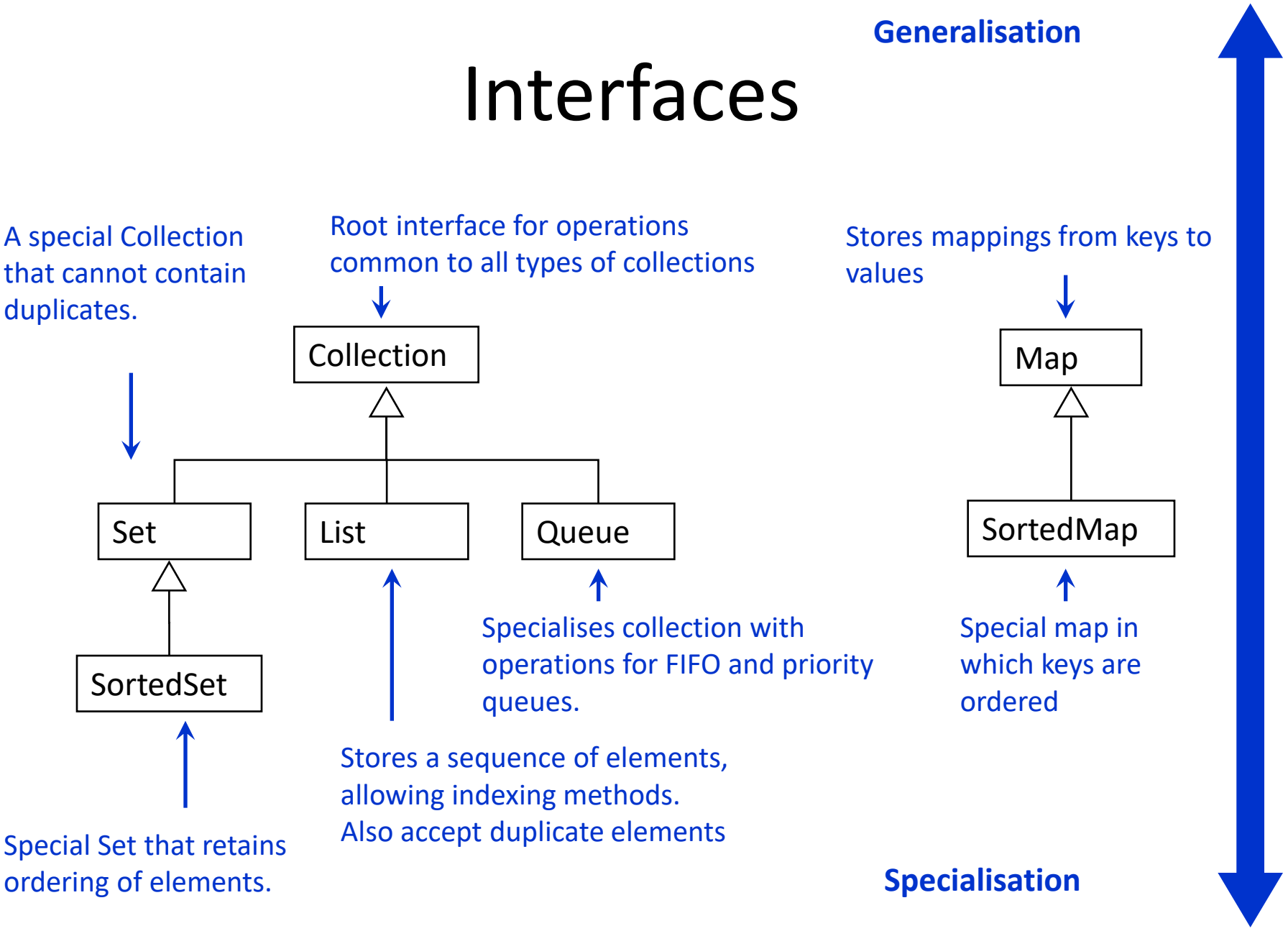
- Collection framework has following components.

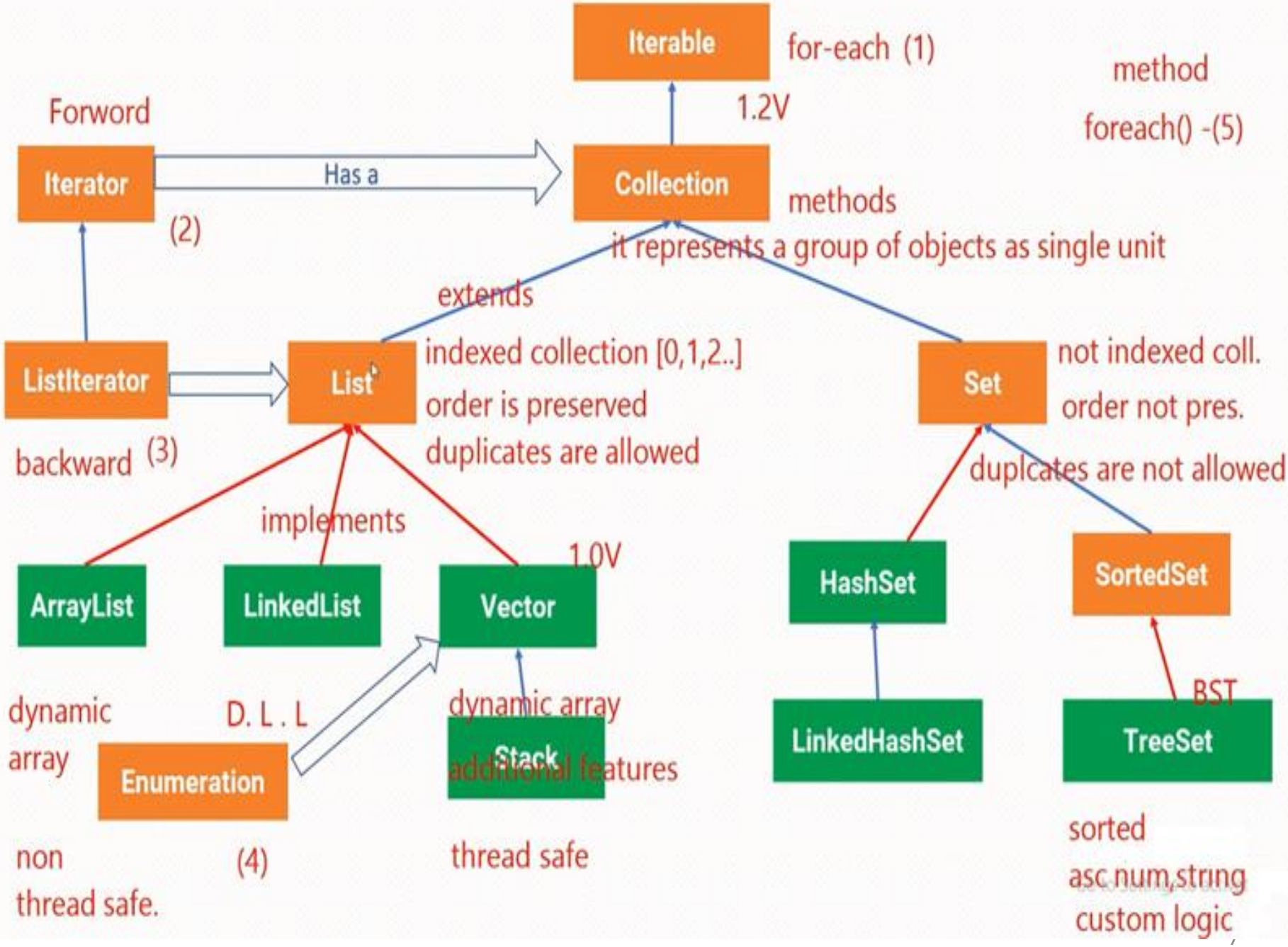


Collection: Basic operations

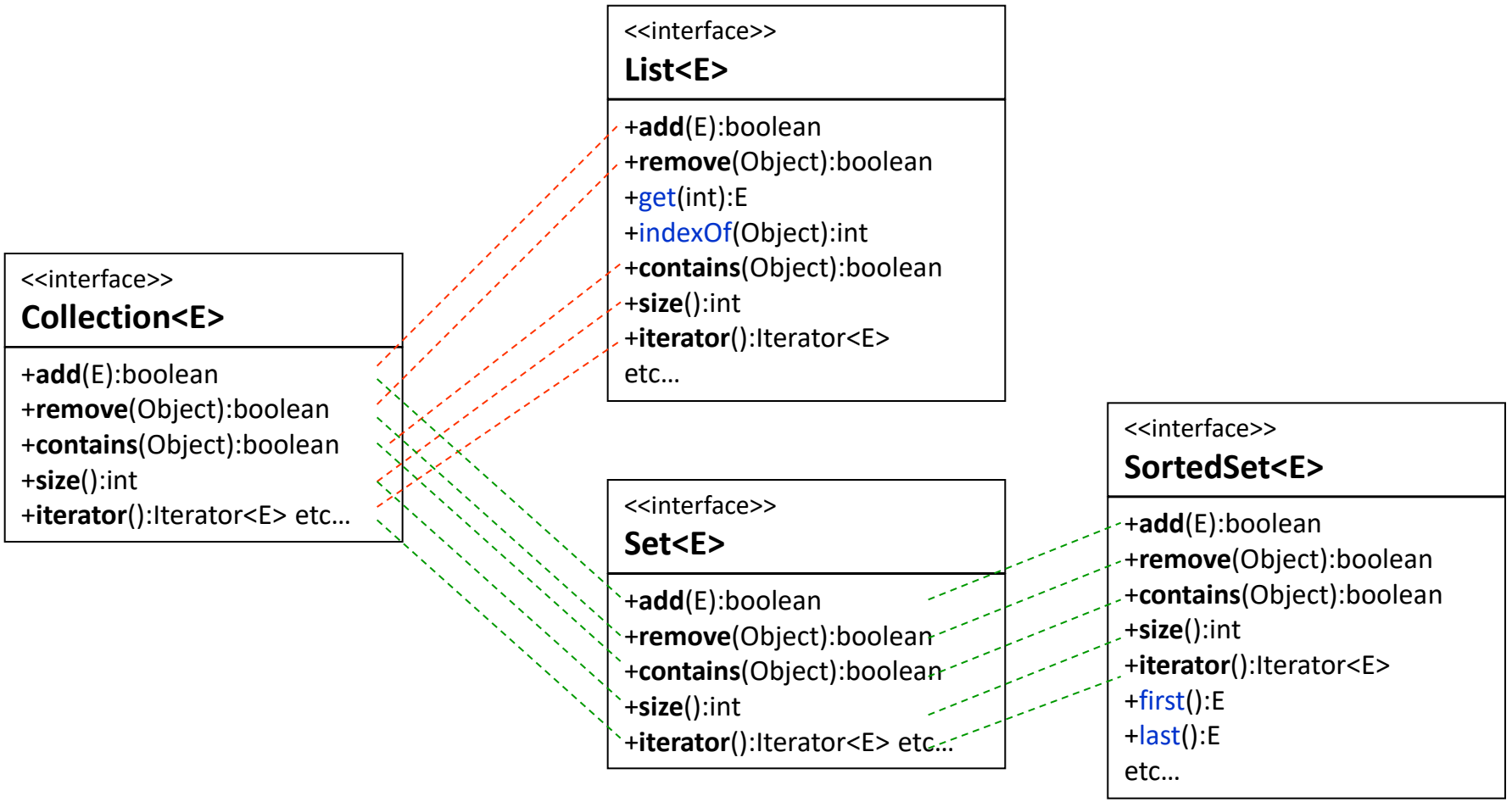
Data Type	Method
int	size( );
boolean	isEmpty( );
boolean	contains(Object element);
boolean	add(Object element);
boolean	remove(Object element);
Iterator	iterator( );

Interfaces


Generalisation
A special Collection that cannot contain duplicates.
Root interface for operations common to all types of collections
Stores mappings from keys to values
Collection
Map
Set
List
Queue
SortedMap
SortedSet
Specialises collection with operations for FIFO and priority queues.
Special map in which keys are ordered
Stores a sequence of elements, allowing indexing methods.
Also accept duplicate elements
Special Set that retains ordering of elements.
Specialisation

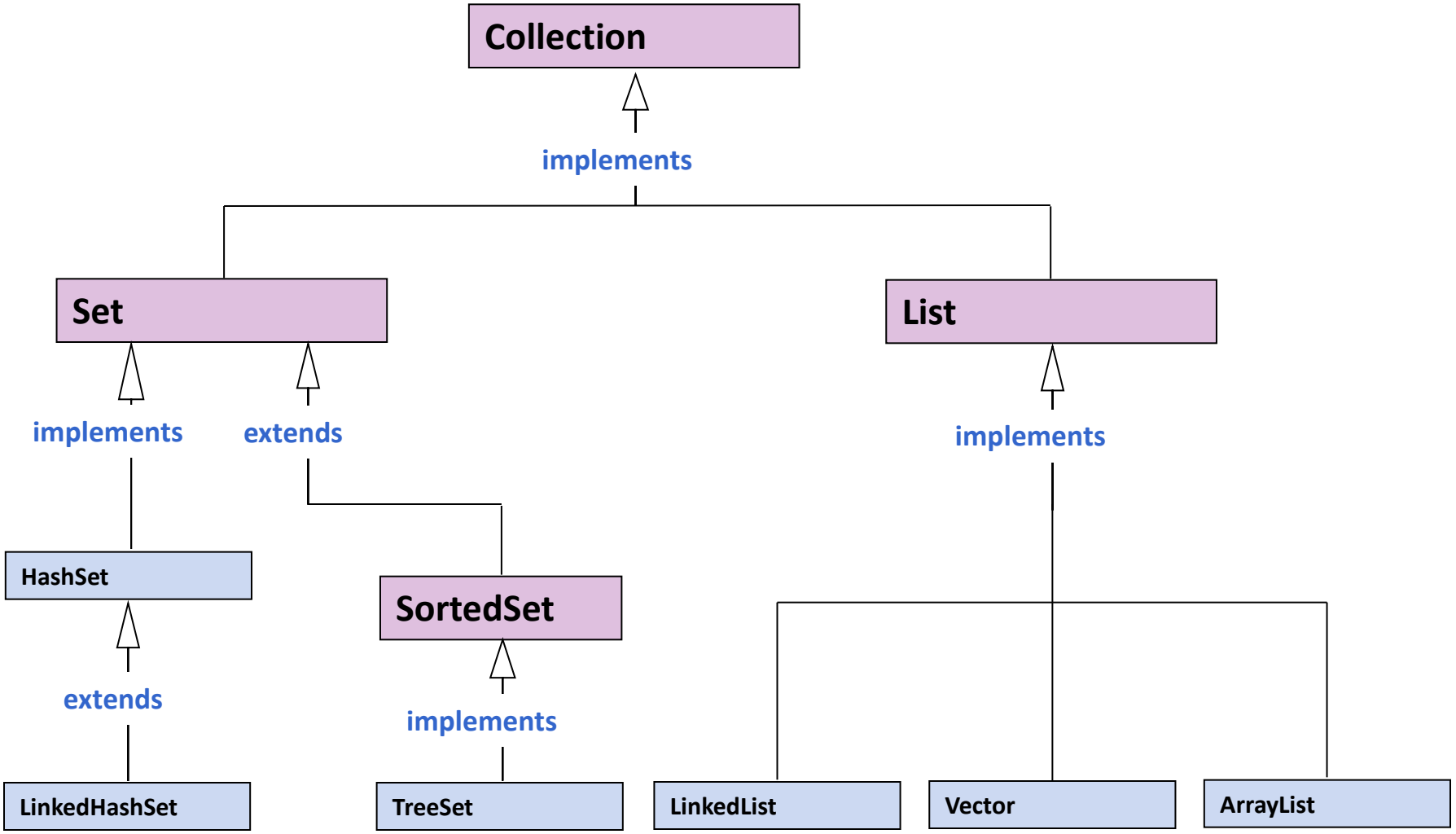

Iterable
for-each (1)
1.2V
method
foreach() -(5)
Forword
Iterator
Has a
(2)
Collection
methods
it represents a group of objects as single unit
extends
ListIterator
List
indexed collection [0,1,2..]
order is preserved
duplicates are allowed
Set
not indexed coll.
order not pres.
duplcates are not allowed
backward (3)
implements
1.0V
ArrayList
LinkedList
Vector
HashSet
SortedSet
dynamic array
D. L . L
dynamic array
BST
Stack
LinkedHashSet
TreeSet
Enumeration
additional features
non thread safe.
(4)
thread safe
sorted
asc num,string
custom logic

Expansion of contracts



Concrete Collections

concrete collection	implements	description
`HashSet`	`Set`	hash table
`TreeSet`	`SortedSet`	balanced binary tree
`ArrayList`	`List`	resizable-array
`LinkedList`	`List`	linked list
`Vector`	`List`	resizable-array
`HashMap`	`Map`	hash table
`TreeMap`	`SortedMap`	balanced binary tree
`Hashtable`	`Map`	hash table


Collection
implements
Set
List
implements
extends
implements
HashSet
SortedSet
extends
implements
LinkedHashSet
TreeSet
LinkedList
Vector
ArrayList

Framework Interfaces

1. **Set:-** It is used to store objects in random order & accept unique data.

2. **Sorted Set:-** It is used to store objects in ascending order.

3. **List:-**

   - It is used to store even **duplicate objects** but retain the sequence in which these objects are being added.

4. **Map:-**

   - It is used to store objects in pair(**Key & value**).
   - In random order according to key.
   - Here each & every object has unique key

5. **Sorted Map:-**

   - Here objects are sorted in pair(key & value) in ascending order according to key by default.
   - Each and every object must have unique key.

Set : Accept uniques elements

HashSet :

Accept Unique elements
Store elemnets randomly
Accept Null value

LinkedHashSet:

Accept Unique elements
Mainain the order of elements
Accept Null value

TreeSet:

Accept Unique elements

It sorts the elements in ascending order
Never Accept Null value

Legacy Classes - Java Collections

Early version of java did not include the Collections framework.

It only defined several classes and interfaces that provide methods for storing objects.

When Collections framework were added in J2SE 1.2, the original classes were reengineered to support the collection interface.

These classes are also known as Legacy classes.

All legacy classes and interface were redesign by JDK 5 to support Generics.

Legacy Classes - Java Collections

The following are the legacy classes defined by **java.util** package

- Dictionary
- HashTable
- Properties
- Stack
- Vector

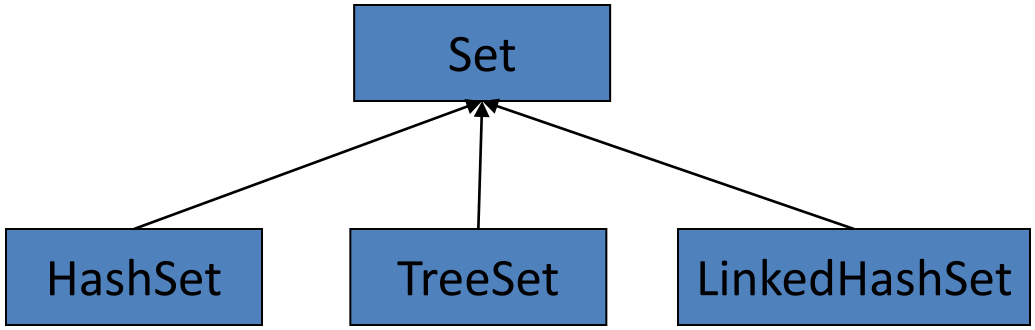
There is only one legacy interface called **Enumeration**

Set

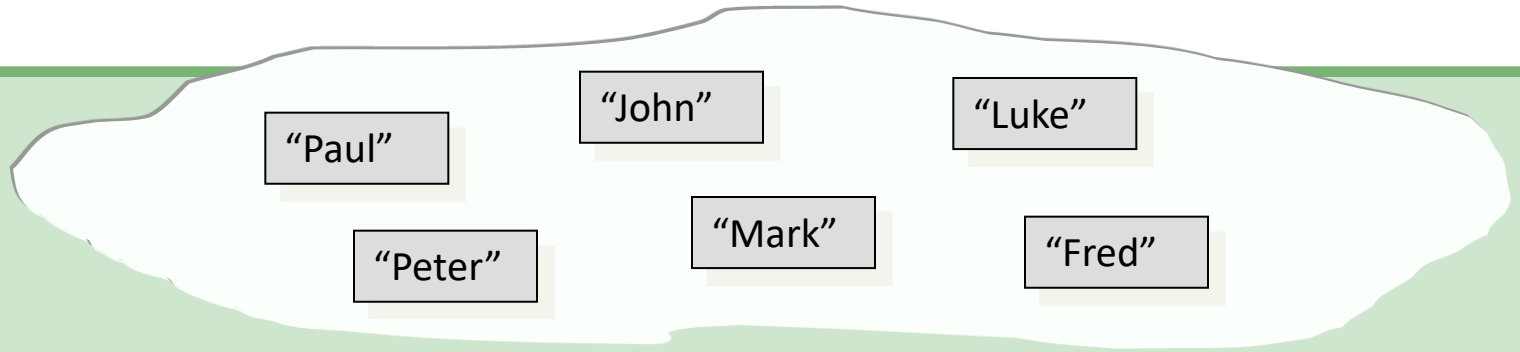
- The set interface extends the collection interface and forbids duplicates within the collection.

- A special set interface for maintaining elements In a sorted order called SortedSet.

- Implementations:



Set



A Set cares about uniqueness, it doesn’t allow duplicates.

HashSet	LinkedHashSet	TreeSet

HashSet

```
import java.util.*;

public class MyHashSet {

   public static void main(String args[ ]) {

        Set hash = new HashSet( );

        hash.add("a");
        hash.add("b");
        hash.add("c");
        hash.add("d");

        Iterator iterator = hash.iterator( );

        while(iterator.hasNext( ))
        {
           System.out.println(iterator.next( ));
        }
   }
}
```

```
d
a
c
b
```

Difference between *HashSet* and *TreeSet*

HashSet:

- it does not guarantee that the order of elements will remain constant over time
  iteration performance depends on the ***initial capacity*** and the ***load factor*** of the HashSet.

TreeSet:

- Guarantees log(n) time cost for the basic operations (add, remove and contains)
  guarantees that elements of set will be sorted (ascending, natural, or the one specified by you via it's constructor)

What is common in *HashSet* and *TreeSet* in Java

1)Both ***HashSet*** and ***TreeSet*** implements ***java.util.Set*** interface which means they follow contract of ***Set*** interface and doesn't allow any duplicates.

2)Both ***HashSet*** and ***TreeSet*** are not thread-safe and not synchronized. Though you can make them synchronized by using ***Collections.synchronizedSet()*** method.

TreeSet

```
import java.util.TreeSet;
import java.util.Iterator;

public class MyTreeSet
{
   public static void main(String args[ ])
   {

        TreeSet<String> tree = new TreeSet<String>();

        tree.add("Jody");
        tree.add("Remiel");
        tree.add("Reggie");
        tree.add("Philippe");

        Iterator iterator = tree.iterator( );

        while(iterator.hasNext( ))
       {
           System.out.println(iterator.next( ).toString( ));
        }
   }
}
```

```
Jody
Philippe
Reggie
Remiel
```

The SortedSet Interface

- TreeSet implements the SortedSet interface.

- SortedSet methods allow access to the least and greatest elements in the collection.
- SortedSet methods allow various views of the collection, for example, the set of all elements greater than a given element, or less than a given element.

Syntax:-

```
SortedSet<String> ss=new TreeSet<String>();

ss.add("a");

ss.add("e");

 ss.add("g");

ss.add("b");

 ss.add("c");
```

Set Implementations

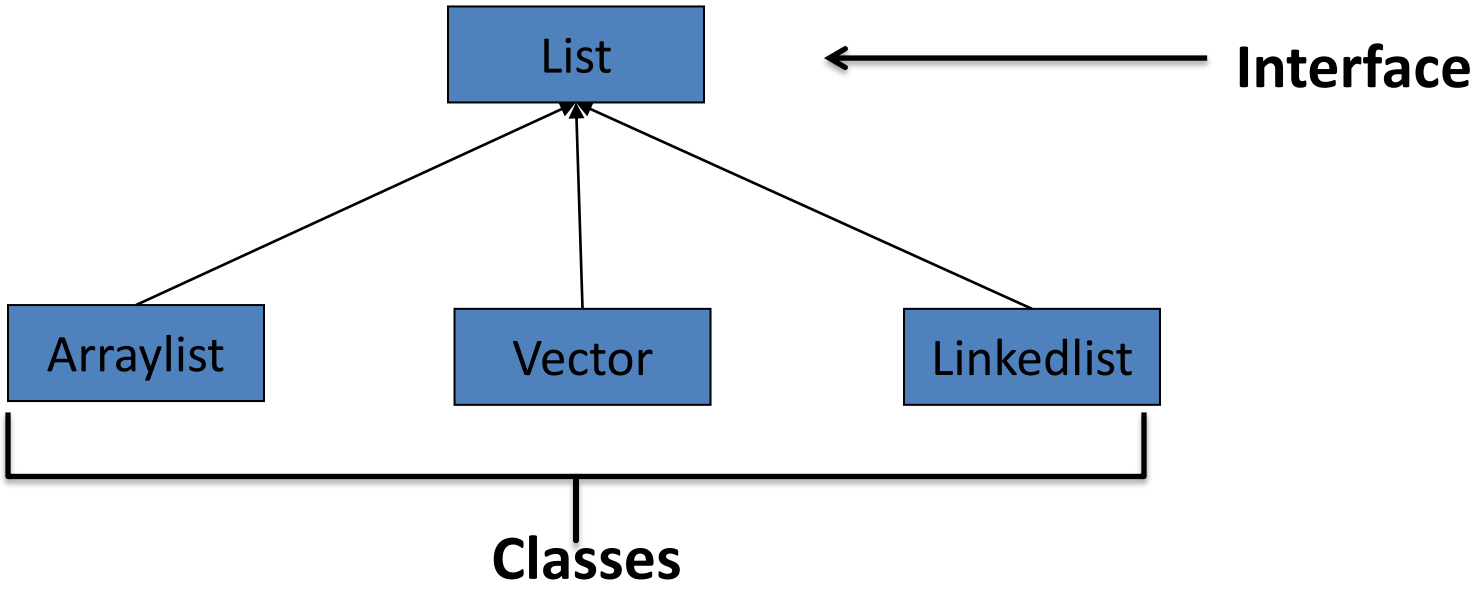
LinkedHashSet

```
import java.util.LinkedHashSet;

public class MyLinkedHashSet {

   public static void main(String args[ ]) {

         LinkedHashSet<String> lhs = new LinkedHashSet<String>();

         lhs.add(new String("One"));
         lhs.add(new String("Two"));
         lhs.add(new String("Three"));

         Object array[] = lhs.toArray( );

         for(int x=0; x<3; x++) {
            System.out.println(array[x]);
         }
   }
}
```

```
One
Two
Three
```

List

- The List interface extends the collection interface to define an ordered collection, permitting duplicates.
- The user can access elements by their integer index(position in the list) and search for element in the list.
- List Implementation



  - ArrayList offers better performance compared to Linkedlist.

Difference between ArrayList and LinkedList in Jav

ArrayList	LinkedList
ArrayList internally uses a dynamic array to store the elements	LinkedList internally uses a doubly linked list to store the elements
Manipulation with ArrayList is slow because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
ArrayList consumes less memory than LinkedList	A LinkedList consumes more memory than an ArrayList because it also stores the next and previous references along with the data.
An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
ArrayList is better for storing and accessing data.	LinkedList is **better for manipulating** data.

List

Method	Description
list.get(index)	Return the object at position index in the list, where index is an integer, The parameter must be in this range. Or an **IndexOfBoundsException** is thrown.
List.set(index,obj)	Stores an object obj at position number index in the list, replacing the object that was there previously.
List.add(index,obj)	Insert an object into the list at position number index.
List.remove(index)	Remove the object at position number index.
List.indexOf(obj)	Return an int gives the position of obj in the list, it it occurs.

Tip: Summary of Adding to an `ArrayList`

- The **`add`** method is usually used to place an element in an **`ArrayList`** position for the first time (at an **`ArrayList`** index)
- The simplest **`add`** method has a single parameter for the element to be added, and adds an element at the next unused index, in order

Example 1

```
import java.util.*;
class arraylist
{
public static void main(String args[])
{
ArrayList<String> arr = new ArrayList(10);
arr.add("A");
arr.add("B");
arr.add("C");
arr.add("X");
arr.add("Y");
arr.add("Z");
}
}
```

Constructor is of unsafe Type. You have to use –Xlint option to compile

```
E:\oop>javac  arraylist.java
Note: arraylist.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

```
E:\oop>javac -Xlint arraylist.java
arraylist.java:6: warning: [unchecked] unchecked conversion
found   : java.util.ArrayList
required: java.util.ArrayList<java.lang.String>
ArrayList<String> arr = new ArrayList(10);
                        ^
1 warning
```

Example 2

```
import java.util.*;
class arraylist
{
public static void main(String args[])
{
ArrayList<String> arr = new ArrayList<String>(10);
arr.add("A");
arr.add("B");
arr.add("C");
arr.add("X");
arr.add("Y");
arr.add("Z");
}
}
```

Safe Constructor

```
E:\oop>javac  arraylist.java

E:\oop>
```

List Implementations

ArrayList

```
import java.util.ArrayList;

public class MyArrayList {

   public static void main(String args[ ]) {

        ArrayList alist = new ArrayList( );

        alist.add(new String("One"));
        alist.add(new String("Two"));
        alist.add(new String("Three"));

        System.out.println(alist.get(0));
        System.out.println(alist.get(1));
        System.out.println(alist.get(2));
   }
}
```

```
One
Two
Three
```

List Implementations

LinkedList

```
import java.util.ArrayList;

public class MyLinkedList
{
   public static void main(String args[ ])
   {

        LinkedList alist = new LinkedList( );

        alist.add(new String("One"));
        alist.add(new String("Two"));
        alist.add(new String("Three"));

        System.out.println(alist.get(0));
        System.out.println(alist.get(1));
        System.out.println(alist.get(2));
   }
}
```

```
One
Two
Three
```

Vector class

Vector is similar to **ArrayList** which represents a dynamic array.

There are two differences between **Vector** and **ArrayList**.

First, Vector is synchronized while ArrayList is not, and Second, it contains many legacy methods that are not part of the Collections Framework.

With the release of JDK 5, Vector also implements Iterable.

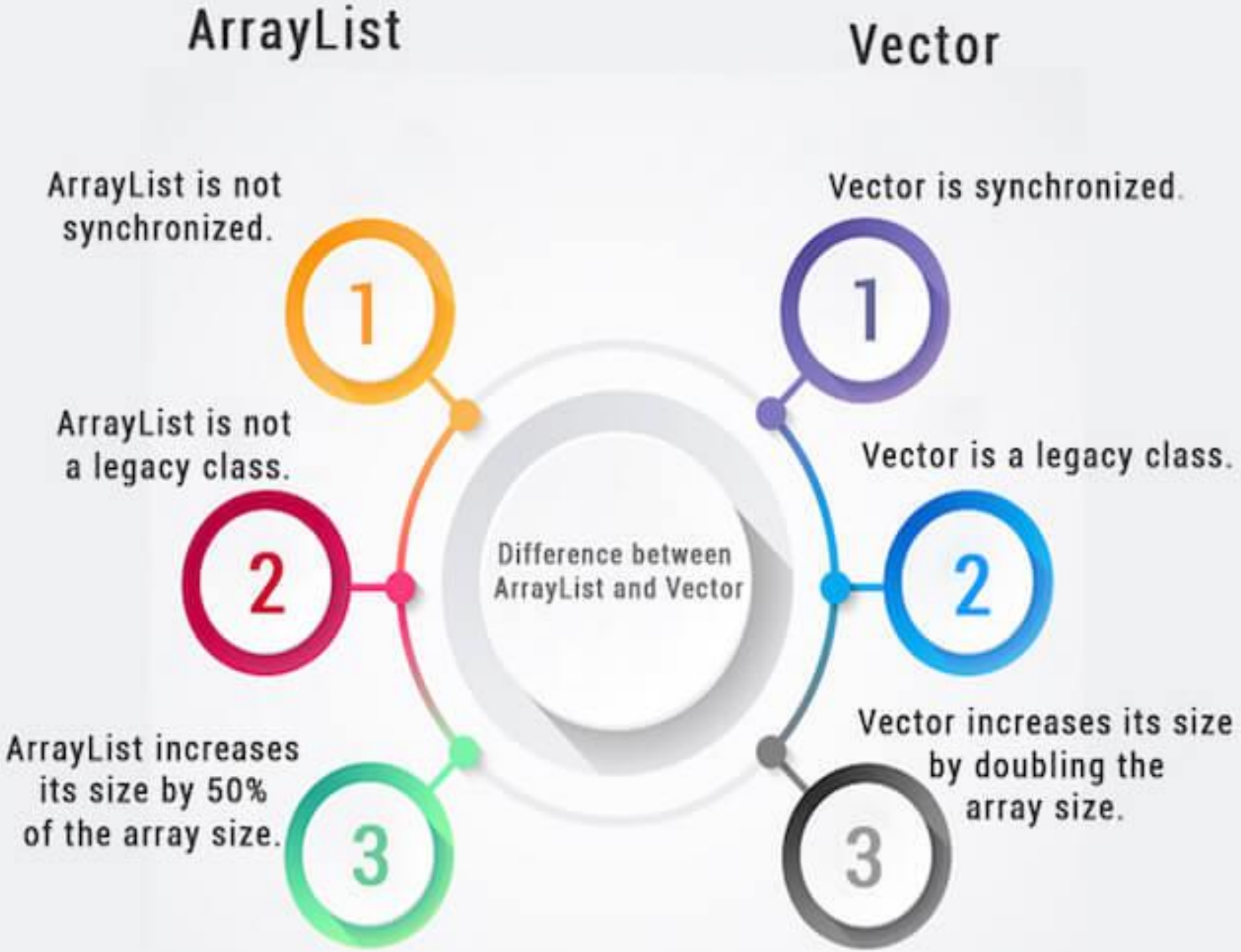
This means that Vector is fully compatible with collections, and a Vector can have its contents iterated by the for-each loop.

Vector

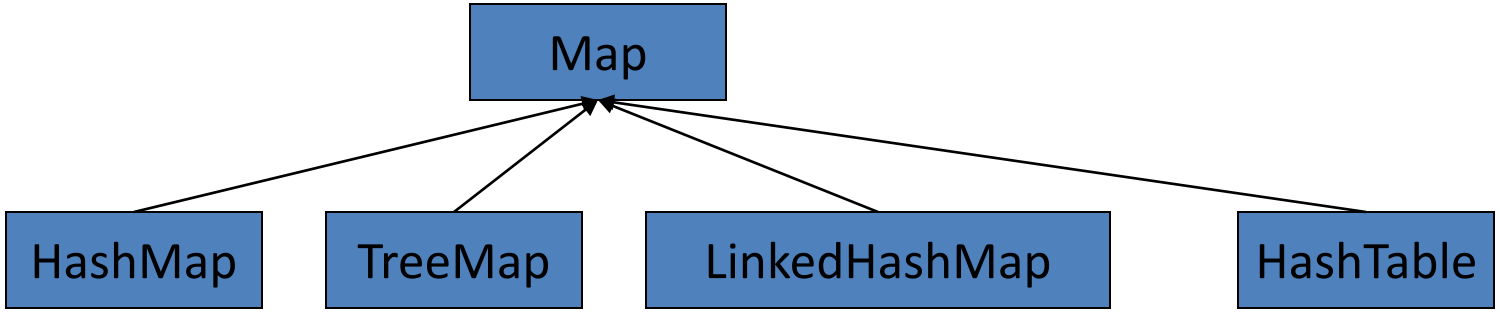
```
import java.util.Vector;

public class VectorExample
{

   public static void main(String[] args) {

      Vector<String> vc=new Vector<String>();
      //    <E> Element type of Vector e.g. String, Integer, Object ...

      // add vector elements
      vc.add("Vector Object 1");
      vc.add("Vector Object 2");
      vc.add("Vector Object 3");
      vc.add("Vector Object 4");
      vc.add("Vector Object 5");

      // add vector element at index
      vc.add(3, "Element at fix position");

      // vc.size() inform number of elements in Vector
      System.out.println("Vector Size :"+vc.size());
   }
}
```

Vector defines several legacy methods. Lets see some important legacy methods defined by **Vector** class.

Method	Description
void addElement(E element)	adds element to the Vector
E elementAt(int index)	returns the element at specified index
Enumeration elements()	returns an enumeration of element in vector
E firstElement()	returns first element in the Vector
E lastElement()	returns last element in the Vector
void removeAllElements()	removes all elements of the Vector

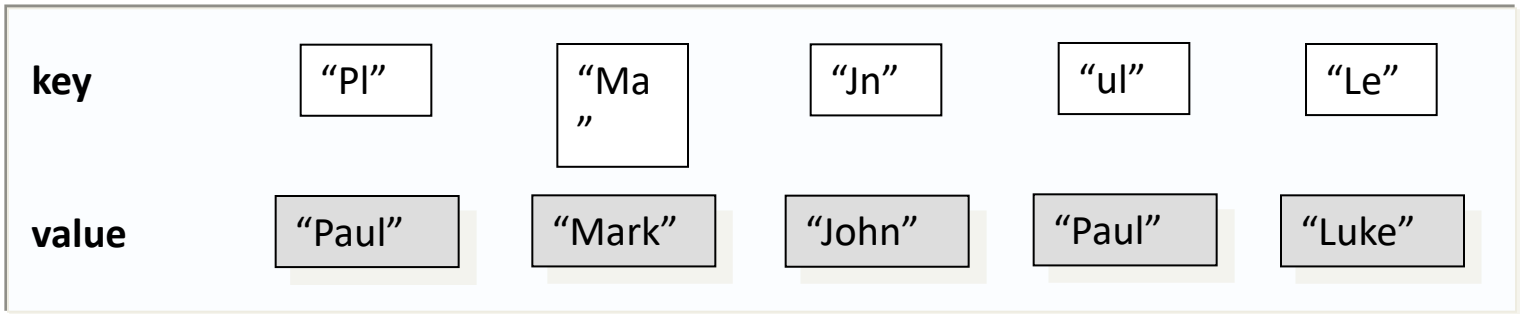

ArrayList
Vector
ArrayList is not synchronized.
1
ArrayList is not a legacy class.
2
ArrayList increases its size by 50% of the array size.
3
Difference between ArrayList and Vector
Vector is synchronized.
1
Vector is a legacy class.
2
Vector increases its size by doubling the array size.
3

Map Implementations



Performance Increases

Map

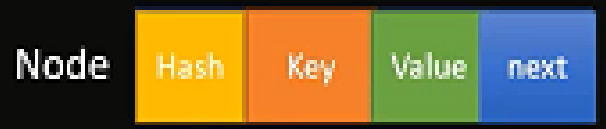


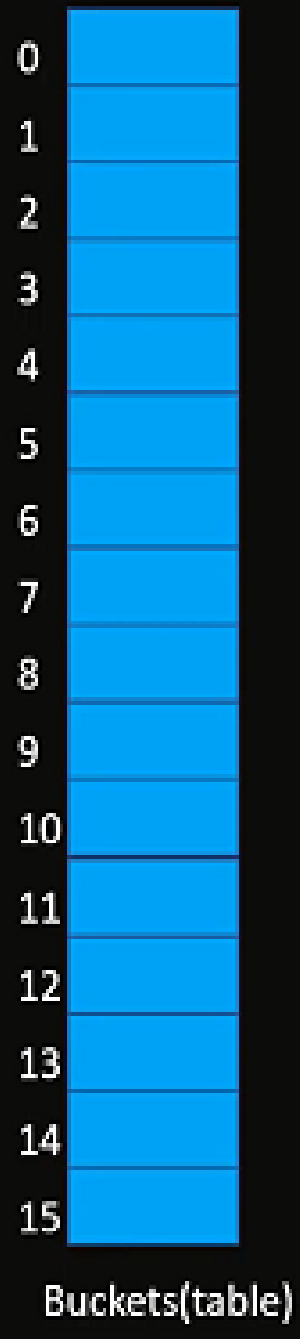
A Map cares about unique identifiers.

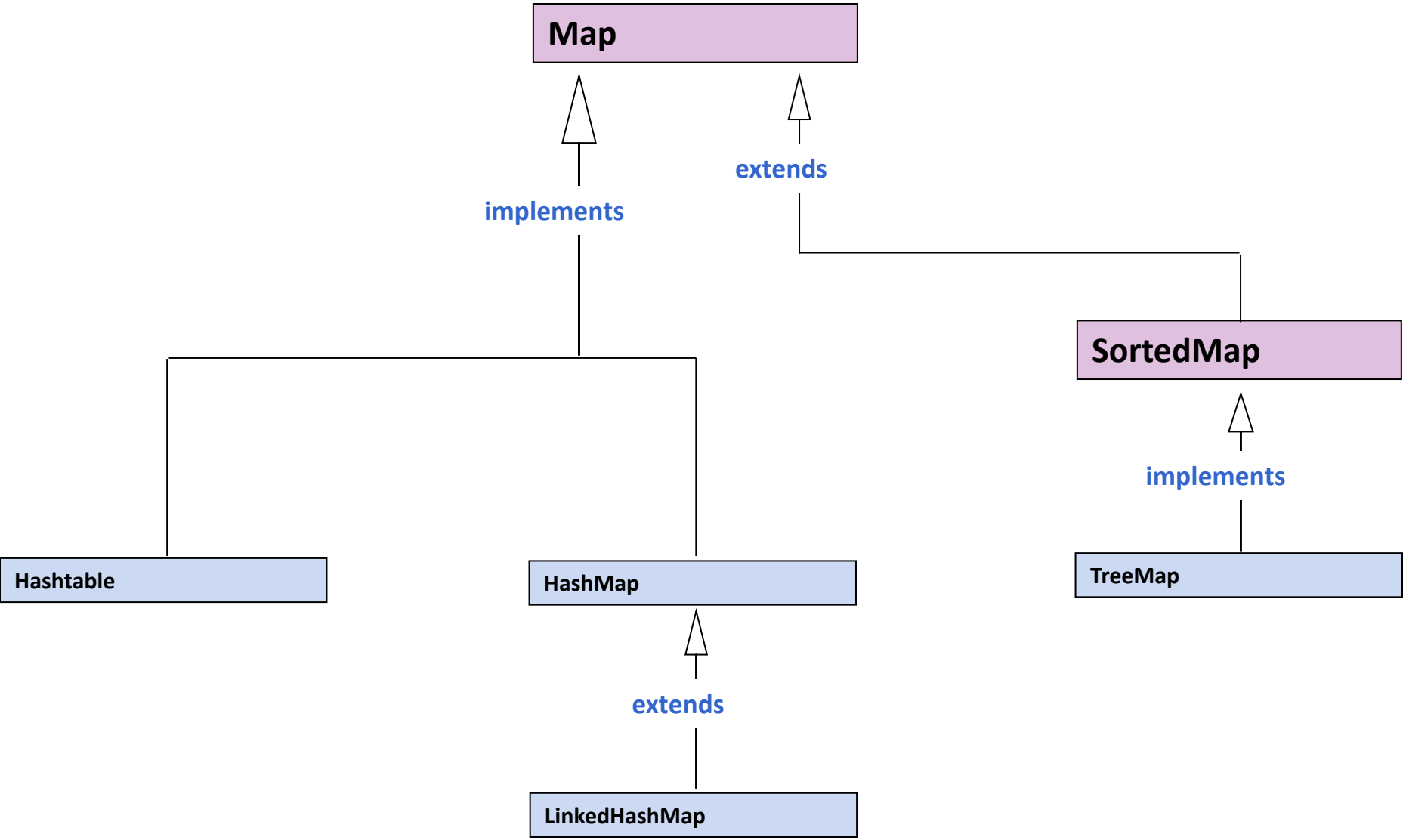
HashMap	Hashtable	LinkedHashMap	TreeMap

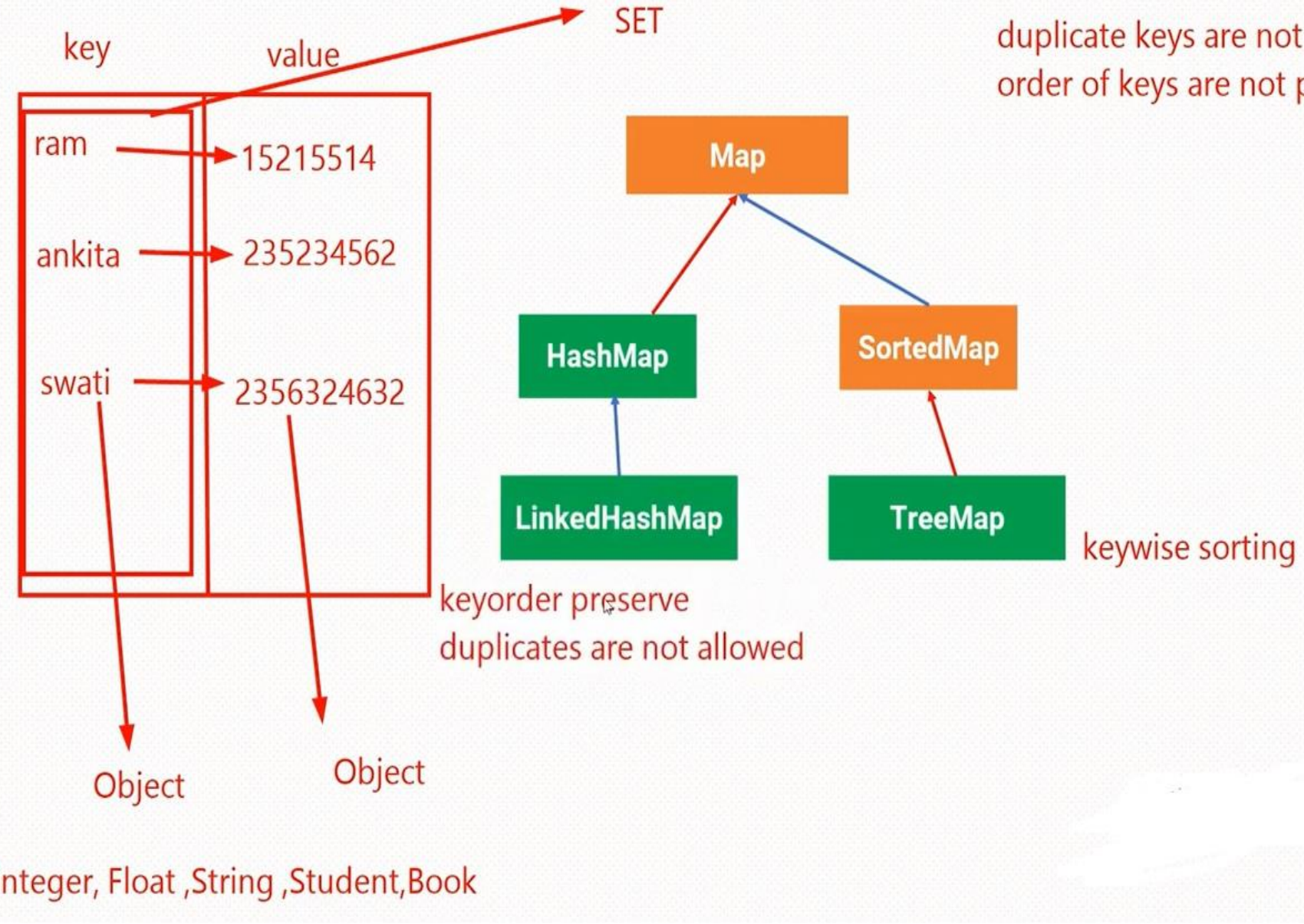
```
HashMap<String , Integer> map = new HashMap<>( ) ;

map.put("physics", 34);
```

```
put(k,v)
{
hash=hashCode(k)

Index=hash&(n-1)

}
```






Map
implements
extends
SortedMap
implements
Hashtable
HashMap
TreeMap
extends
LinkedHashMap


key
value
SET
ram
15215514
ankita
235234562
swati
2356324632
Object
Object
duplicate keys are not allowed
order of keys are not preserved
Map
HashMap
SortedMap
LinkedHashMap
TreeMap
keywise sorting
keyorder preserve
duplicates are not allowed
Integer, Float ,String ,Student,Book

The Map interface

- A Map is an object that maps keys to values
- A map cannot contain duplicate keys
- Each key can map to at most one value
- Examples: dictionary, phone book, etc.

Map implementations

- `Map` is an interface; you can't say `new Map( )`
- Here are two implementations:
  - `HashMap` is the faster
  - `TreeMap` guarantees the order of iteration
- It's poor style to expose the implementation unnecessarily, so:
- Good: `Map map = new HashMap( );`
  Fair: `HashMap map = new HashMap( );`

Map: Basic operations

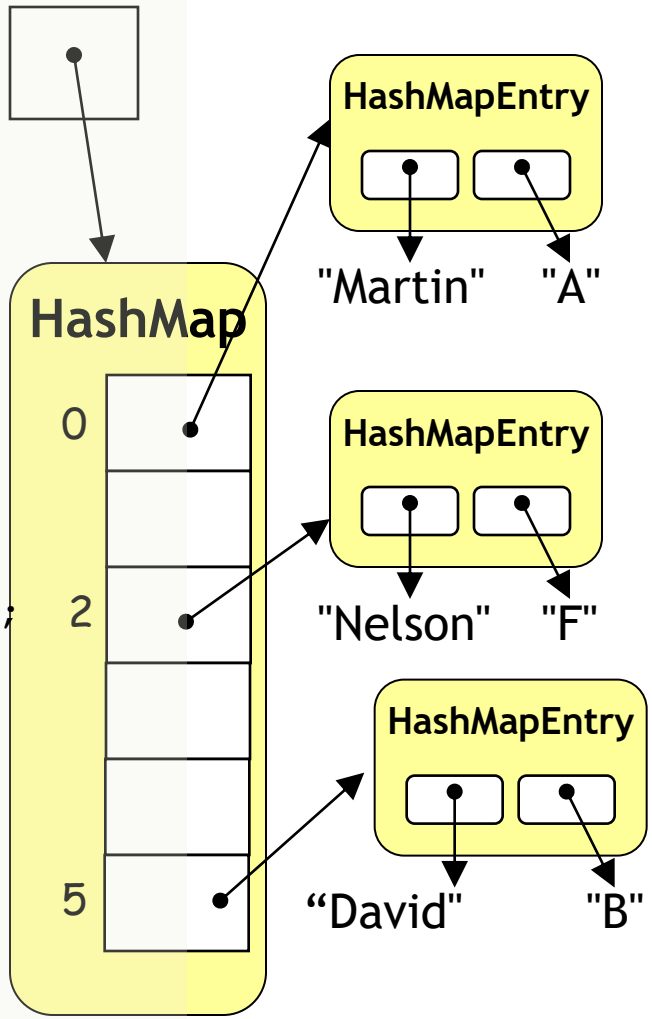
```
Object put(Object key, Object value);
Object get(Object key);
Object remove(Object key);
boolean containsKey(Object key);
boolean containsValue(Object value);
int size( );
boolean isEmpty( );
```

More about put

- If the map already contains a given key, put(key, value) replaces the value associated with that key
- This means Java has to do equality testing on keys
- With a HashMap implementation, you need to define equals and hashCode for all your keys
- With a TreeMap implementation, you need to define equals and implement the Comparable interface for all your keys

HashMap example

```
import java.util.HashMap;

public class MyHashMap
{

   public static void main(String args[ ])
   {

        HashMap<String,String> map = new
                HashMap<String,String>();
        map.put("Martyn", "A");
        map.put("Nelson", "F");
        map.put("Davis", "B");

System.out.println("Name: " +map.get("Martyn"));

System.out.println("ID: " + map.get("Nelson"));

System.out.println("Address: " +
map.get("David"));

   }
}
```



Map Example

```
import java.util.*;
class MapDemo
{
        public static void main(String args[])
        {Map<Integer,String> obj = new HashMap<Integer,String>();
                obj.put(new Integer(103),"David");
                obj.put(new Integer(101),"Ravi");
                obj.put(new Integer(102),"Amit");
                obj.put(new Integer(104),“Ram");

                //System.out.println("Map Demo.."+obj);

                Set s =obj.entrySet();

                Iterator itr = s.iterator();
                while(itr.hasNext())
                {
                        Map.Entry me =(Map.Entry)itr.next();
                        System.out.println(me.getKey()+"   "+me.getValue());
                }
        }
}
```

```
One
102     Amit
103     David
101     Ravi
104     Ram
```

Map: Bulk operations

- void putAll(Map t);
  - Copies one Map into another
  - Example: newMap.putAll(oldMap);
- void clear();
  - Example: oldMap.clear();

Map: Collection views

- public Set keySet( );
- public Collection values( );
- public Set entrySet( );
  - returns a set of Map.Entry (key-value) pairs

- You can create iterators for the key set, the value set, or the entry set (the set of entries, that is, key-value pairs)
- The above views provide the *only* way to iterate over a Map

Map example

```
import java.util.*;

public class MapExample
 {
    public static void main(String[] args)
    {

Map<String, String> fruit = new HashMap<String, String>();

        fruit.put("Apple", "red");

        fruit.put("Pear", "yellow");

        fruit.put("Plum", "purple");

        fruit.put("Cherry", "red");

        for (String key : fruit.keySet())
        {
           System.out.println(key + ": " + fruit.get(key));

        }
    }
}
```

Output

```
Plum:   purple
Apple:  red
Pear:   yellow
Cherry:  red
```

Map.Entry

Interface for entrySet elements

- public interface Entry { // Inner interface of Map
  Object getKey( );
  Object getValue( );
  Object setValue(Object value);
  }
- This is a small interface for working with the Collection returned by entrySet( )
- Can get elements *only* from the Iterator, and they are only valid during the iteration

Hashtable class

Like HashMap, Hashtable also stores key/value pair. However neither **keys** nor **values** can be **null**.

There is one more difference

between **HashMap** and **Hashtable** that is Hashtable is synchronized while HashMap is not.

ConcurrentHashMap	Synchronized HashMap
ConcurrentHashMap is a class that implements the ConcurrentMap and serializable interface.	We can synchronize the HashMap by using the **synchronizedMap()** method of java.util.Collections class.
It locks some portion of the map.	It locks the whole map.
ConcurrentHashMap allows performing concurrent read and write operation. Hence, performance is relatively better than the Synchronized Map.	In Synchronized HashMap, multiple threads can not access the map concurrently. Hence, the performance is relatively less than the ConcurrentHashMap.
ConcuurentHashMap doesn't allow inserting null as a key or value.	Synchronized HashMap allows inserting null as a key.
ConccurentHashMap doesn't throw ConcurrentModificationException.	Synchronized HashMap throw **ConcurrentModificationException**.

Difference between HashMap and Hashtable

Hashtable	HashMap
Hashtable class is synchronized.	HashMap is not synchronized.
Because of Thread-safe, Hashtable is slower than HashMap	HashMap works faster.
Neither **key** nor **values** can be null	Both **key** and **values** can be null
Order of table remain constant over time.	does not guarantee that order of map will remain constant over time.

Map Implementations

Hashtable

```
Table of Contents:
{address=Manila, name=Jody, id=1001}
```

```
import java.util.Hashtable;

public class MyHashtable {

   public static void main(String args[ ])
  {

        Hashtable table = new Hashtable( );
        table.put("name", "Jody");
        table.put("id", new Integer(1001));
        table.put("address", new String("Manila"));

        System.out.println("Table of Contents:" + table);
    }
}
```

TreeMap

```
import java.util.*;

public class MyTreeMap
{
   public static void main(String args[])
   {
       TreeMap treeMap = new TreeMap( );
       treeMap.put("name", "Jody");
       treeMap.put("id", new Integer(446));
       treeMap.put("address", "Manila");

       Collection values = treeMap.values()
       Iterator iterator = values.iterator( );
       System.out.println("Printing the
       VALUES....");

       while (iterator.hasNext())
       {
          System.out.println(iterator.next( ));
       }
   }
```

```
the VALUES....

Manila

446

Jody
```

Iterator:

- An iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively.
- The **java.util.Iterator<E>** interface provides for one way traversal and **java.util.ListIterator<E>** provides two way traversal.
- Iterator <E> is a replacement for the older Enumeraion class which was used before collection through an iterator.
- Each of the collection classes provides an **iterator()** method that return an iterator to the start of the collection.
- By using this **iterator** object , we can access each element in the collection, one element at a time.

Iterator: (Interface)

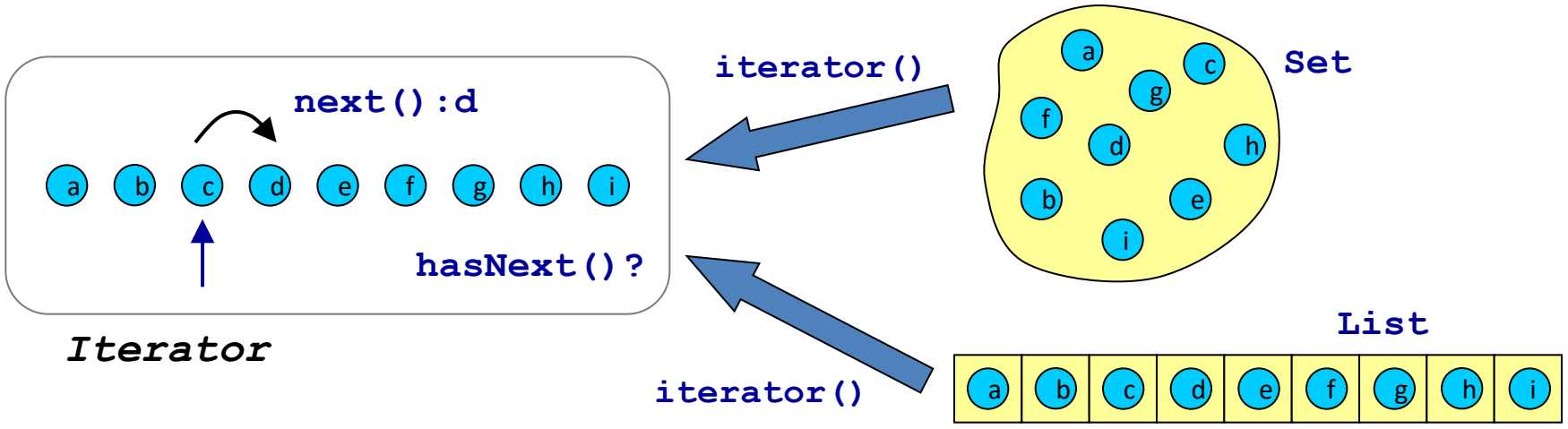
Method	Description
Boolean **hasNext()**	Method return true if the iterator has more elements.
Object **next()**	Method returns the next element in the iterator.
Void **remove()**	Is the safe way to modify a collection during iterator.

java.util.Iterator<E>

- **Think about typical usage scenarios for Collections**
  - **Retrieve the list of all patients**
  - **Search for the lowest priced item**

- **More often than not you would have to traverse every element in the collection – be it a List, Set, or your own data structure.**

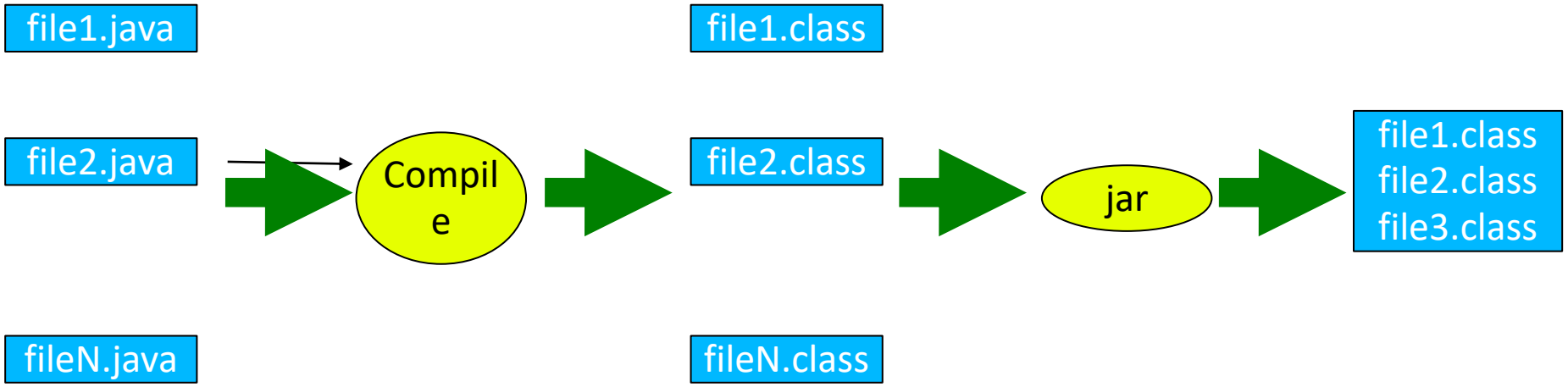
- **Iterators provide a generic way to traverse through a collection regardless of its implementation**



```
Public class IteratorTest
{
        public static void main(String args[])
        {
                ArrayList          al = new ArrayList();

                al.add("C");        //add elements to the array list.
                al.add("A");
                al.add("E");
                al.add("B");
        //Display all contents of arraylist al.

         System.out.println("Original elements of al...."+al);

        //Use iterator to display content of arraylist al.
        Iterator  itr = al.iterator();
        While(itr.hasNext())
        {
                Object element = itr.next();
                System.out.println("Element..."+element);
        }
```

```
Public class IteratorTest
{
        public static void main(String args[])
        {
                ArrayList          al = new ArrayList();

                al.add("C");        //add elements to the array list.
                al.add("A");
                al.add("E");
                al.add("B");
        //Display all contents of arraylist al.

         System.out.println("Original elements of al...."+al);

        //Use iterator to display content of arraylist al.
        Iterator  itr = al.iterator();
        While(itr.hasNext())
        {
                Object element = itr.next();
                System.out.println("Element..."+element);
        }
```

.class and jar files



Typically a jar file contains class files of a package.