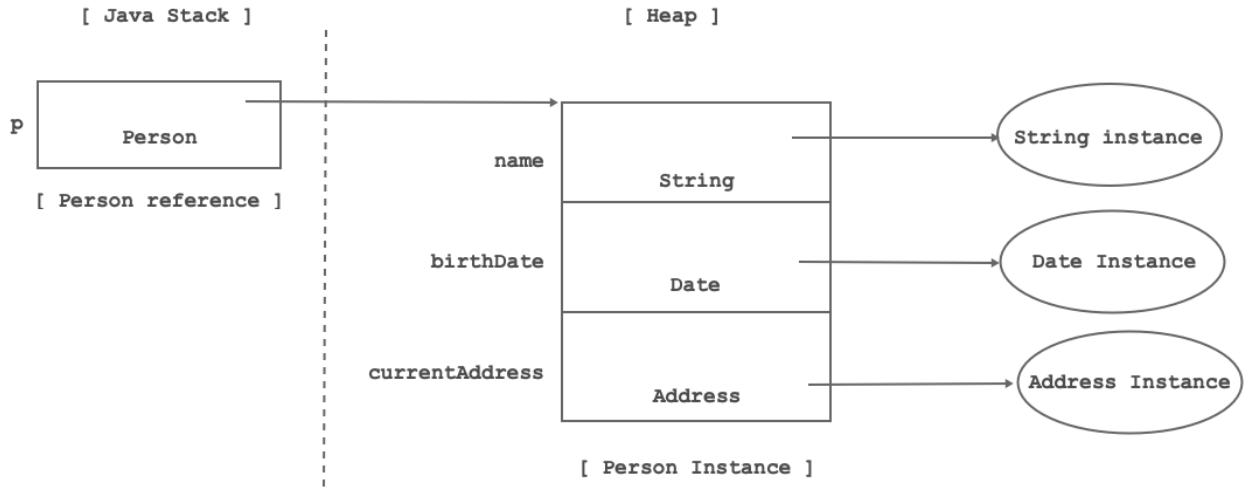
Day 22

Collection Framework

- Consider following example:

```
class Date{ }
class Addresss{ }
class Person{
  private String name = new String();
  private Date birthDate = new Date();
  private Address currentAddress = new Address();
}
class Program{
  public static void main(String[] args) {
    Person p = new Person( );
  }
}
```

- In Java, instance do not get space inside another instance. Rather instance contains reference of another instance.



Library

- In Java, .jar file is a library file.
- It can contain, menifest file, resources, packages.
- Package can contain sub package, interface, class, enum, exception, error, annotation types
- Example: rt.jar

Framework

- framework = collection of libraries + tools + rules/guidelines
- It is a development platform which contain reusable partial code on the top of it we can develop application.
- Examples:

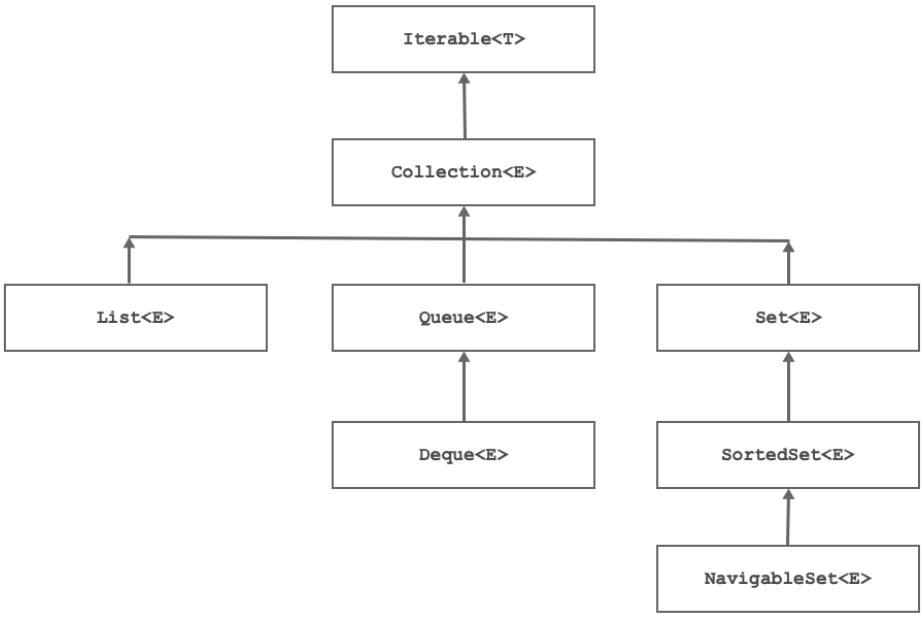
- JUnit: Unit testing framework which is used to write test case.
- Apache Log4j2: Logging framework which is used to record activities.
- AWT/Swing/Java-FX: GUI framework.
- JNI: Framework to access native code
- Struts: Readymade MVC based web application framework.
- Hibernate: ORM based automatic persistence framework
- Spring: Enterprise framework

Collection

- Any instance which contains multiple elements is called as collection.
- In java, data structure is also called as collection.

Collection Framework

- Collection framework is a library of data structure classes on the top of it we can develop Java application.
- In Java, collection framework talk about use not about implementation.
- In Java, when we use collection to store instance then it doesnt contain instance rather it contains reference of the instance.
- To use collection framework, we should import java.util package.



[Collection Framework Interface Hierarchy]

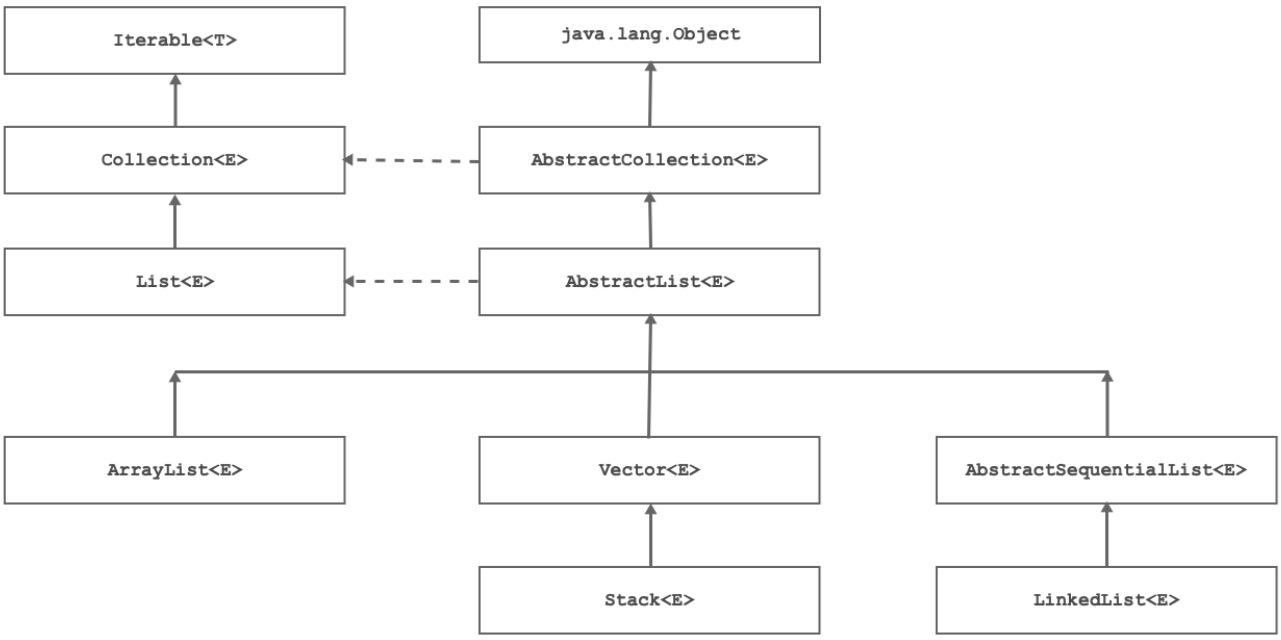
Iterable

- It is interface declared in java.lang package.
- It is introduced in jDK 1.5.
- Implementing this interface allows an object to be the target of the "for-each loop" statement.
- Methods:
  - Iterator iterator()
  - default Spliterator spliterator()
  - default void forEach(Consumer<? super T> action)

Collection

- Reference: https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html
- Value stored inside any collection(Array, Stack, Queue, LinkedList etc.) is called as element.
- It is interface declared in java.util package.
- It is root interface in the collection framework interface hierarchy.
- The JDK does not provide any direct implementations of Collection interface.
- Direct implementation classes of Collection interface are AbstractList, AbstractQueue, AbstractSet.
- List, Queue, Set are sub interfaces of java.util.Collection interface.
- Abstract methods of java.util.Collection interface:
    - boolean add(E e)
    - boolean addAll(Collection<? extends E> c)
    - void clear()
    - boolean contains(Object o)
    - boolean containsAll(Collection<?> c)
    - boolean isEmpty()
    - boolean remove(Object o)
    - boolean removeAll(Collection<?> c)
    - boolean retainAll(Collection<?> c)
    - int size()
    - Object[] toArray()
    - T[] toArray(T[] a)
- Default methods of java.util.Collection interface:
    - default Stream stream()
    - default Stream parallelStream()
    - default boolean removeIf(Predicate<? super E> filter)

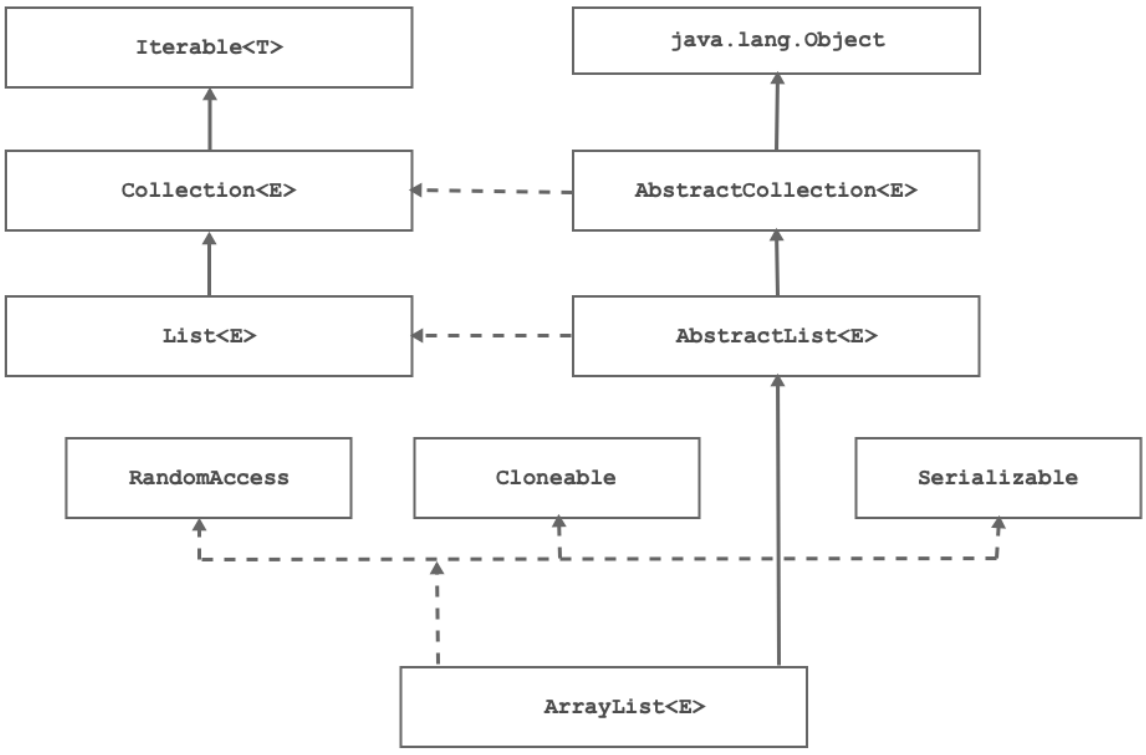
List



- This interface is a member of the Java Collections Framework and introduced in JDK 1.2

- It is sub interface of Collection interface. It means that all the methods of Collection interafce will be inherited into List interface.
- Direct implementation classes of List interfaces are AbstractList, ArrayList, Vector, Stack, LinkedList. These collection classes are called as List collections.
- Inside List collection we can store data in sequential fashion.
- We can store duplicate elements inside any List collection.
- We can store multiple null values inside List collection.
- With the help of integer index, we can access elements from List collection.
- We can traverse elements of any List collection using Iterator as well as ListIterator.
- This interface is a member of the Java Collections Framework.
- Abstract methods of java.util.List interface:
    - void add(int index, E element)
    - boolean addAll(int index, Collection<? extends E> c)
    - E remove(int index)
    - E get(int index)
    - E set(int index, E element)
    - int indexOf(Object o)
    - int lastIndexOf(Object o)
    - ListIterator listIterator()
    - ListIterator listIterator(int index)
    - List subList(int fromIndex, int toIndex)
- Default methods of java.util.List interface:
    - default void sort(Comparator<? super E> c)
    - default void replaceAll(UnaryOperator operator)
- Note: If we want to manage elements of non final type inside any List collection then we should override at least equals methods inside non final type.

ArrayList



- Array is collection of fixed elements. ArrayList is resizeable array.
- Implementation of ArrayList is based of array.
- ArrayList is List collection.
- Since ArrayList is List collection we can store elements sequentially.
- Since ArrayList is List collection, we can store duplicate elements as well as null elements inside ArrayList.
- Since ArrayList is List collection, we can access its elements using integer index.
- Since ArrayList is List collection, we can traverse its elements using Iterator as well as ListIterator.
- ArrayList implementation is unsynchronized. Using Collections.synchronizedList() method we can make it synchronized.
- If ArrayList is full then its capacity gets increased by half of existing capacity.
- This class is a member of the Java Collections Framework and introduced in JDK 1.2.
- Constructor Summary of ArrayList class:
    - public ArrayList()

      ```
      ArrayList<Intger> list = new ArrayList();
      ```

    - public ArrayList(int initialCapacity)

      ```
      ArrayList<Intger> list = new ArrayList( 15 );
      ```

    - public ArrayList(Collection<? extends E> c)

      ```
      Collection<Integer> c = new ArrayList<>( );
      c.add( 10 );
      c.add( 20 );
      c.add( 30 );

      ArrayList<Integer> list = new ArrayList<>( c );
      ```

- Method Summary of ArrayList class:
    - public void ensureCapacity(int minCapacity)
    - protected void removeRange(int fromIndex, int toIndex)
    - public void trimToSize()
- Instantiation:

```
public static void main(String[] args){
    Collection<Integer> collection = new ArrayList<>(); //OK:
Upcasting
    List<Integer> list = new ArrayList<>(); //OK: Upcasting
    ArrayList<Integer> arrayList = new ArrayList<>();   //OK
}
```

- How to add single element inside ArrayList?

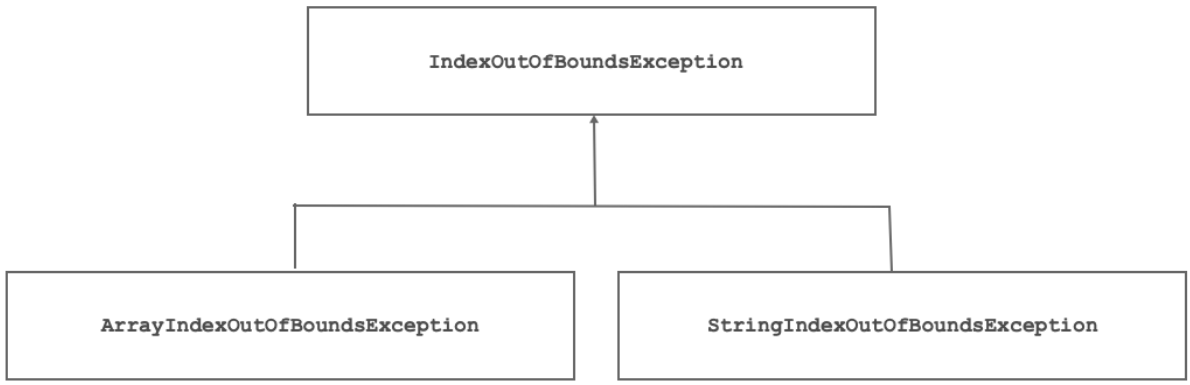
```
public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    list.add(10);
    list.add(20);
    list.add(40);
    list.add(50);
    list.add(2, 30);
    System.out.println( list.toString());
}
```

```
public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    list.add(10);
    list.add(20);
    list.add(30);
    list.add(40);
    list.add(50);
    return list;
}
public static void main(String[] args) {
    List<Integer> list = Program.getList();
    System.out.println( list.toString()); //[10, 20, 30, 40, 50]
}
```

```
public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    list.add(10);
    list.add(20);
    list.add(30);
    list.add(40);
    list.add(50);
    return list;
}
public static void main(String[] args) {
    List<Integer> list = Program.getList();
    Integer element = null;
    for( int index = 0; index < list.size(); ++ index ) {
```

```
        element = list.get( index );
        System.out.println(element);
    }
}
```

```
    public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    list.add(10);
    list.add(20);
    list.add(30);
    list.add(40);
    list.add(50);
    return list;
  }
  public static void main(String[] args) {
    int[] arr = new int[ ] { 10, 20, 30 };
    //int element = arr[ arr.length ];
//ArrayIndexOutOfBoundsException

    String str = "CDAC";
    //char ch = str.charAt(str.length());
//StringIndexOutOfBoundsException

    List<Integer> list = Program.getList();
    Integer element = list.get( list.size() );
//IndexOutOfBoundsException
  }
```




```
    public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    list.add(10);
    list.add(20);
    list.add(30);
    list.add(40);
```

```
    list.add(50);
    return list;
  }
  public static void main(String[] args) {
    List<Integer> list = Program.getList();
    Integer element = null;
    Iterator<Integer> itr = list.iterator();
    while( itr.hasNext()) {
      element = itr.next();
      System.out.println(element);
    }
  }
```

```
    public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    list.add(10);
    list.add(20);
    list.add(30);
    list.add(40);
    list.add(50);
    return list;
  }
  public static void main(String[] args) {
    List<Integer> list = Program.getList();
    for( Integer element : list )
      System.out.println( element );
  }
```

```
    public static void main(String[] args) {
    List<Integer> list = Program.getList();
    /* Consumer<Integer> action = System.out::println;
    list.forEach(action); */
    list.forEach( System.out::println );
  }
```

```
    public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    list.add(10);
    list.add(20);
    list.add(30);
    list.add(40);
    list.add(50);
    return list;
  }
  public static void main(String[] args) {
    List<Integer> list = Program.getList();
    ListIterator<Integer> itr = list.listIterator();
```

```
    Integer element = null;
    while( itr.hasNext()) {
      element = itr.next();
      System.out.print( element+"   ");
    }
    System.out.println();
    while( itr.hasPrevious()) {
      element = itr.previous();
      System.out.print( element+"   ");
    }
  }
```

```
//Object[] elementData;
private static int capacity(List<Integer> list) throws Exception{
    Class<?> c = list.getClass();
    Field field = c.getDeclaredField("elementData");
    field.setAccessible(true);
    Object[] elementData = (Object[]) field.get(list);
    return elementData.length;
}
public static void main(String[] args) {
    try {
        List<Integer> list = Program.getList();
        System.out.println("Size        :   "+list.size()); //5

        int capacity = Program.capacity( list );
        System.out.println("Capacity    :   "+capacity);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

- How to add multiple elements inside ArrayList?

```
public static void main(String[] args){
    Collection<Integer> collection = new ArrayList<>();
    collection.add(30);
    collection.add(40);
    collection.add(50);

    //List<Integer> list = new ArrayList<>( collection );   //OK
    List<Integer> list = new ArrayList<>(  );
    list.add(10);
    list.add(20);
    list.addAll(collection);
    System.out.println(list);
}
```

```
public static void main(String[] args) {
    Collection<Integer> collection = new ArrayList<>();
    collection.add(30);
    collection.add(40);
    collection.add(50);

    List<Integer> list = new ArrayList<>(  );
    list.add(10);
    list.add(20);
    list.add(60);
    list.add(70);
    list.addAll(2, collection);
    System.out.println(list);
}
```

- How will you search single element inside ArrayList?

```
    public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    for( int count = 1; count <= 10; ++ count  )
      list.add( count * 10 );
    return list;
  }
  public static void main(String[] args) {
    List<Integer> list = Program.getList(); //[10, 20, 30, 40, 50, 60,
70, 80, 90, 100]
    Integer key = new Integer(500);
    if( list.contains(key)) {
      int index = list.indexOf(key);
      System.out.println( key+" found at index : "+index);
    }else
      System.out.println(key+" not found.");
  }
```

- How will you search and remove multiple elements

```
    public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    for( int count = 1; count <= 10; ++ count  )
      list.add( count * 10 );
    return list;
  }
    public static void main(String[] args) {
    List<Integer> list = Program.getList(); //[10, 20, 30, 40, 50, 60,
70, 80, 90, 100]
    Collection<Integer> keys = new ArrayList<>( );
    keys.add(30);
    keys.add(50);
    keys.add(70);
```

```
    if( list.containsAll(keys)) {
      list.removeAll(keys); //[10, 20, 40, 60, 80, 90, 100]
      //list.retainAll(keys);   //[30, 50, 70]
      System.out.println( list );
    }else
      System.out.println(keys+" not found.");
  }
```

- How will you search and remove single element from ArrayList?

```
    public static List<Integer> getList( ){
    List<Integer> list = new ArrayList<>();
    for( int count = 1; count <= 10; ++ count  )
      list.add( count * 10 );
    return list;
  }
  public static void main(String[] args) {
    List<Integer> list = Program.getList(); //[10, 20, 30, 40, 50, 60,
70, 80, 90, 100]
    Integer key = new Integer(50);
    if( list.contains(key)) {
      //list.remove(key);
      int index = list.indexOf(key);
      list.remove(index);
      System.out.println( list );   //[10, 20, 30, 40, 60, 70, 80, 90,
100]
    }else
      System.out.println(key+" not found.");
  }
```

- How will you sort ArrayList?

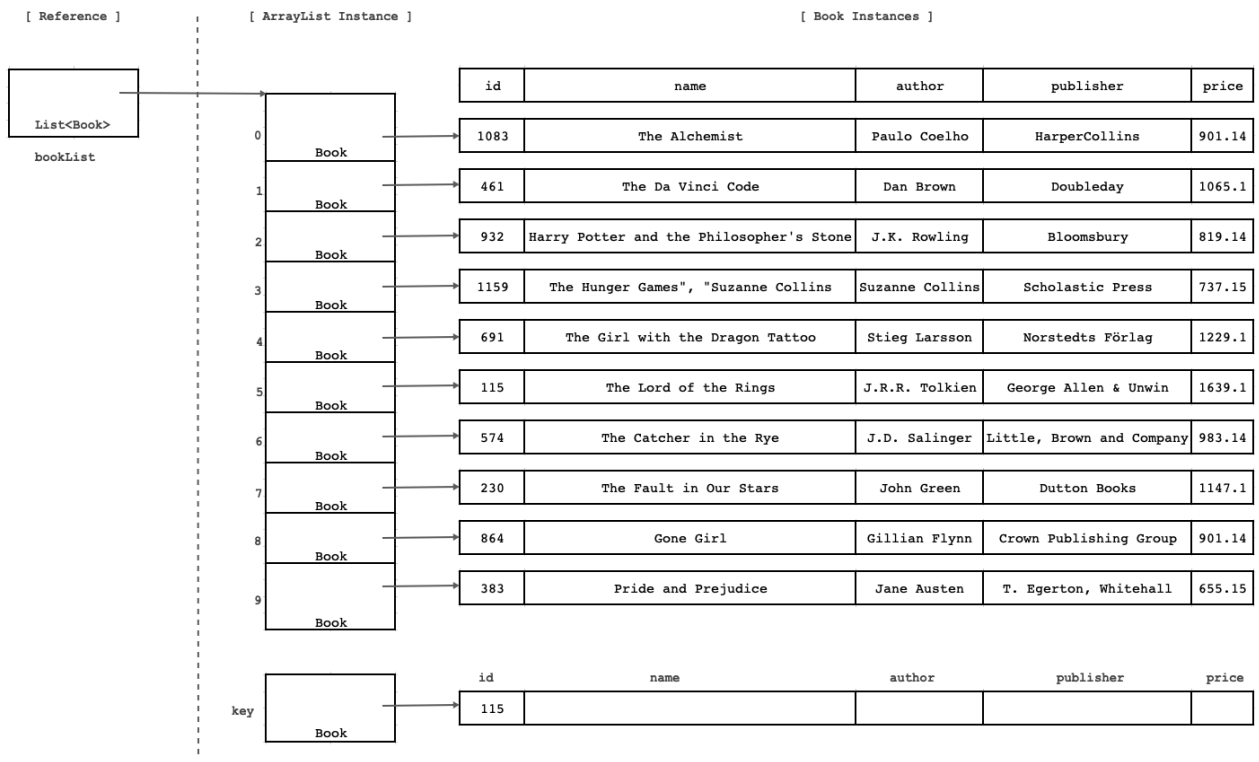
```
    public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    list.add(50);
    list.add(10);
    list.add(30);
    list.add(20);
    list.add(40);

    System.out.println(list);   //[50, 10, 30, 20, 40]
    //Collections.sort( list );
    list.sort(null);
    System.out.println(list);   //[10, 20, 30, 40, 50]
  }
```

- How will you convert ArrayList into array?

```
  public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    list.add(50);
    list.add(10);
    list.add(30);
    list.add(20);
    list.add(40);

    //Object[] arr = list.toArray();

    Integer[] arr = new Integer[ list.size() ];
    list.toArray(arr);

    System.out.println( Arrays.toString(arr));  //[50, 10, 30, 20, 40]
  }
```

- Using Arrays.asList() method

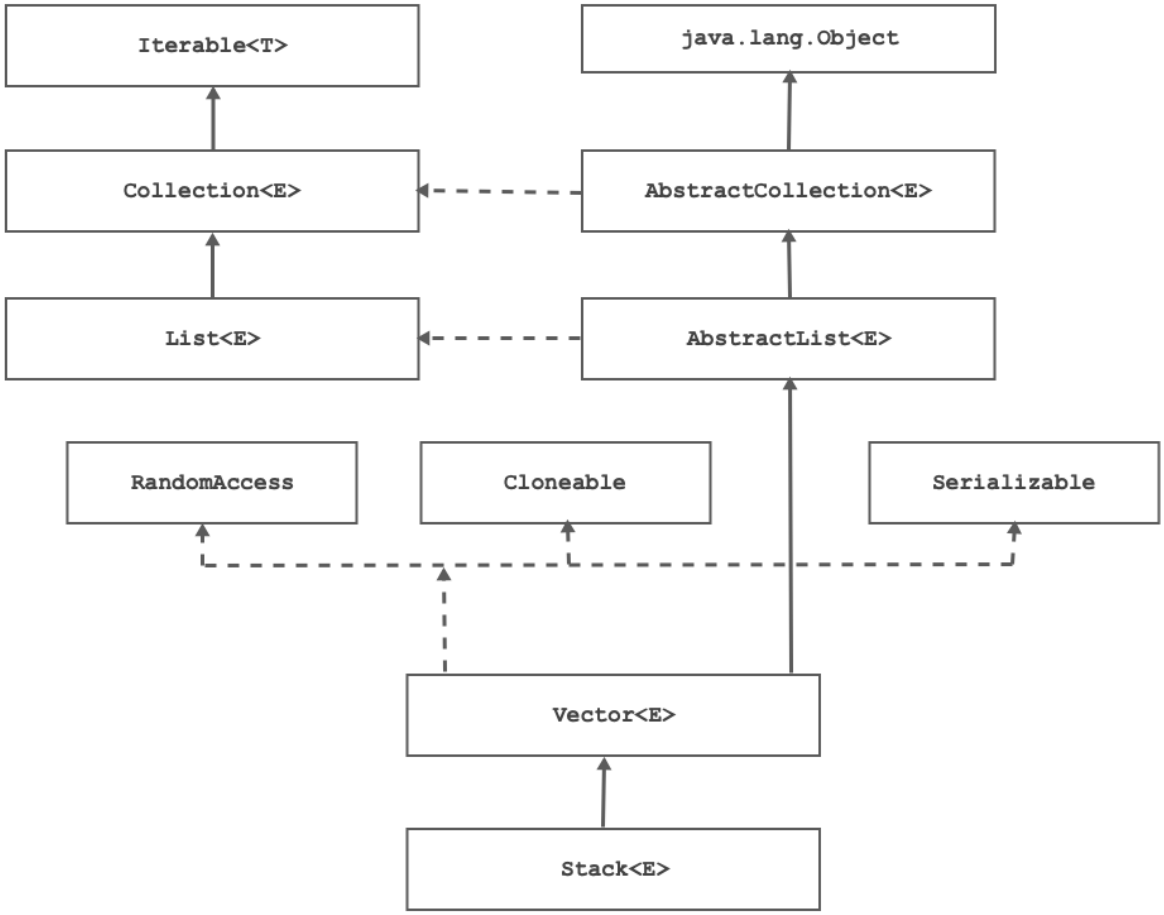
```
    public static void main(String[] args) {
    List<Integer> list = Arrays.asList(10, 20, 30, 40, 50 );
    System.out.println( list.getClass().getName());
//java.util.Arrays$ArrayList
    System.out.println( list ); //[10, 20, 30, 40, 50]
  }
```



Which collection classes are by default synchronized in Java?

- java.util.Vector
- java.util.Stack
- java.util.Hashtable
- java.util.Properties

Vector



- Vector is a class declared in java.util package.
- Vector is List collection whose implementation is based on array.
- Since Vector is List collection, it is ordered/sequential collection.
- Since Vector is List collection, it can contain duplicate elements as well as null elements
- Since Vector is List collection, we can traverse its elements using Iterator as well as ListIterator.
- We can traverse elements of Vector using java.util.Enumeration , java.util.Iterator as well as ListIterator.
- Vector is Synchronized collection.
- Default capacity is 10 elements. Once Vector is full it gets double capacity.
- It was introduced in JDK 1.0. Hence it is also called as legacy class.

Travsering using Enumeration

- Enumeration is interface declared in java.util package.
- It was introduced in JDK 1.0.
- Methods of Enumeration I/F:

    - boolean hasMoreElements()
    - E nextElement()

- Using Enumeration we can traverse limited collections. For Example: Vector, Hashtable etc.
- Using Enumeration, we can traverse collection only forward direction. During traversing we can not add, set or remove elements from underlying collection.

```
    public static void main(String[] args){
    Vector<Integer> v = new Vector<>();
    for( int count = 1; count <= 10; ++ count )
      v.add(count);

    Integer element = null;
    Enumeration<Integer> e = v.elements() ;
    while( e.hasMoreElements()) {
      element = e.nextElement();
      System.out.println(element);
    }
  }
```

Travsering using Iterator

- Iterator is interface declared in java.util package.
- This interface is a member of the Java Collections Framework.
- Methods of Iterator interface:
    - boolean hasNext()
    - boolean hasNext()
    - default void remove()
    - default void forEachRemaining(Consumer<? super E> action)
- Iterator takes the place of Enumeration in the Java Collections Framework. Iterators differ from enumerations in two ways:
    - Iterators allow the caller to remove elements from the underlying collection during the iteration.
    - Method names have been improved.

```
  public static void main(String[] args){
    Vector<Integer> v = new Vector<>();
    for( int count = 1; count <= 10; ++ count )
      v.add(count);

    Integer element = null;
    Iterator<Integer> itr = v.iterator();
    while( itr.hasNext()) {
      element = itr.next();
      System.out.println(element);
```

```
    }
  }
```

Travsering using ListIterator

- It is subinterface of Iterator interface which is declared in java.util package.
- We can use it to traverse only List collections(ArrayList, Vector, Stack, LinkedList etc.)
- We can use ListIterator to traverse collection in bidirection. During travsering, using iterator we can add/set/remove element from collection.
- Method Summary
  - void add(E e)
  - void set(E e)
  - void remove()
  - boolean hasNext()
  - E next()
  - boolean hasPrevious()
  - E previous()
  - int nextIndex()
  - int nextIndex()
- This interface is a member of the Java Collections Framework.
- It is introduced in JDK 1.2

```
    public static void main(String[] args){
    Vector<Integer> v = new Vector<>();
    for( int count = 1; count <= 10; ++ count )
      v.add(count);

    Integer element = null;
    ListIterator<Integer> itr = v.listIterator();
    //ListIterator<Integer> itr = v.listIterator( 4 );
    //ListIterator<Integer> itr = v.listIterator( v.size() );
    while( itr.hasNext()) {
      element = itr.next();
      System.out.print(element+"    ");
    }
    System.out.println();
    while( itr.hasPrevious()) {
      element = itr.previous();
      System.out.print(element+"    ");
    }
  }
```

What is the difference between Enumeration and Iterator

- Using Enumeration we can traverse collection only in forward direction. During traversing, using Enumeration, we can not add/set/remove element from underlying Collection. Using Iterator we can traverse collection only in forward direction. During traversing, using Iterator, we can not add/set element but we can remove element from underlying Collection.
- We can use Enumeration for few Collections only but we can use Iterator for any collection that implements Iterable interface.
- Enumeration method names are long bur Iterator methods names are short.
- Enumeration was introduced in JDK 1.0 whereas Iterator was introduced in JDK1.2.

What is the difference between Iterator and ListIterator

- Using Iterator we can traverse any Collection which implements Iterable interface but Using ListIterator we can traverse any List collection.
- Using Iterator we can traverse collection only in forward direction whereas using ListIterator we can traverse collection in bidirection.
- During traversing, using iterator, we can not add/set element from underlying collection but we can remove element. During traversing, using ListIterator, we can add/set/remove element from underlying collection.

What do you know about fail-fast and not fail-fast(i.e. fail-safe) iterator

or What do you know about ConcurrentModificationException?

- During traversing, without iterator, if we try to make changes in underlying collection and if we get ConcurrentModificationException then such iterator is called as fail-fast Iterator.

```
public static void main(String[] args){
    Vector<Integer> v = new Vector<>();
    for( int count = 1; count <= 10; ++ count )
        v.add(count);

    Integer element = null;
    Iterator<Integer> itr = v.iterator();
    while( itr.hasNext()) {
        element = itr.next();
        System.out.println(element);
        if( element == 10 )
            v.add(11);  //ConcurrentModificationException
    }
}
```

- During traversing, without iterator, if we try to make changes in underlying collection and if we do not get ConcurrentModificationException then such iterator is called as fail-safe Iterator. Such iterators works by creating copy of the Collection.

```
public static void main1(String[] args){
    Vector<Integer> v = new Vector<>();
```

```
        for( int count = 1; count <= 10; ++ count )
            v.add(count);

        Integer element = null;
        Enumeration<Integer> e = v.elements() ;
        while( e.hasMoreElements()) {
            element = e.nextElement();
            System.out.println(element);
            if( element == 10 )
                v.add(11);  //OK
        }
        System.out.println(v);
    }
```

What is the difference between ArrayList and Vector?

- Synchronization: ArrayList collection in unsynchronized whereas Vector is collection synchronized.
- Capacity: Incase of arrayList, capacity gets increased by half of existing capacity. In case of vector, capacity get increased by existing capacity.
- Traversing: We can traverse elements of ArrayList using Iterator and ListIterator whereas we can traverse elements of vector using Enumeration, Iterator and ListIterator.
- Legacy: ArrayList collection is introduced in JDK 1.2 whereas Vector collection is introduced in JDK 1.0.

Stack

- It is sub class of java.util.Vector class.
- IN Java, Stack is synchronized collection.
- If we want to perform operations in Last In First Out(LIFO) order/manner then we should use Stack.
- Method Summary Stack:
    - public boolean empty()
    - public E push(E item)
    - public E peek()
    - public E peek()
    - public int search(Object o)

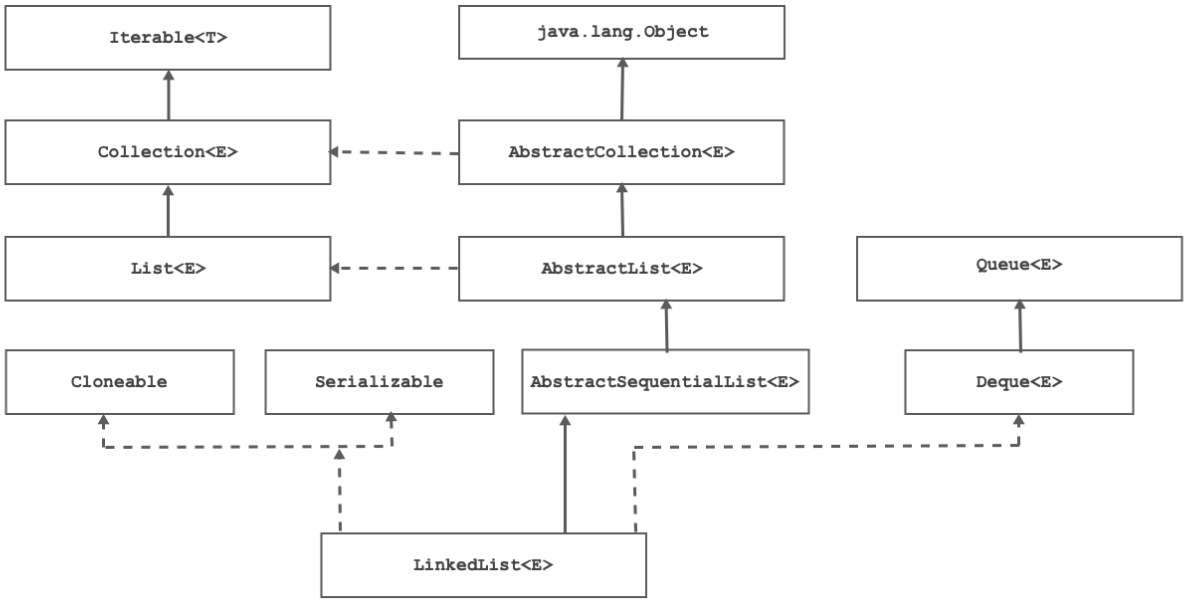
```
    public static void main(String[] args) {
        Stack<Integer> stk = new Stack<>();
        stk.push(10);
        stk.push(20);
        stk.push(30);
        stk.push(40);
        stk.push(50);

        Integer element = null;
        while( !stk.empty()) {
            element = stk.peek();
            System.out.println("Removed element is  :   "+element);
```

```
            stk.pop();
        }
    }
```

- If we want, unsynchronized implementation of Stack then we should use Deque implementation

```
public static void main(String[] args) {
    Deque<Integer> stk = new ArrayDeque<>();
    stk.push(10);
    stk.push(20);
    stk.push(30);
    stk.push(40);
    stk.push(50);

    Integer element = null;
    while( !stk.isEmpty()) {
        element = stk.peek();
        System.out.println("Removed element is  :   "+element);
        stk.pop();
    }
}
```

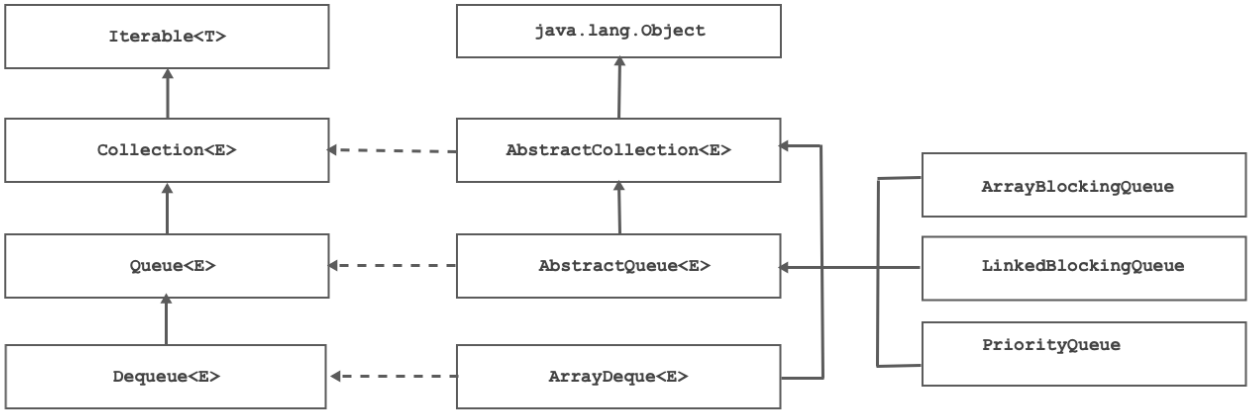
LinkedList



- It is a class declared in java.util package. Its implementation is based on Doubly LinkedList.
- LinkedList class implements List as well as Deque interface.
- Since it is List collection, It stored elements in sequential manner.
- Since it is List collection, It can contain duplicate elements as well as null elements
- Since it is List collection, We can access its elements using integer index.
- Since it is List collection, We can traverse its elements using Iterator and ListIterator
- LinkedList collection is unsynchronized. Using Collections.synchronizedList() method we can make it synchronized.

- This class is a member of the Java Collections Framework.It is introduced in JDK 1.2

Queue



- It is sub interface of Collection interface.
- If we want to perform operations in First In First Out order then we should use Queue implementation.
- This interface is a member of the Java Collections Framework.
- It is introduced in JDK 1.5
- Method Summary of Queue interface:
    - boolean add(E e)
    - boolean offer(E e)
    - E remove()
    - E poll()
    - E element()
    - E peek()
- Consider following code:

```
public static void main(String[] args) {
    Queue<Integer> que = new ArrayDeque<>();
    que.add(10);
    que.add(20);
    que.add(30);
    que.add(40);
    que.add(50);
    //que.add(null);    //Not Allowed

    Integer element = null;
    while( !que.isEmpty() ) {
        element = que.element();
        System.out.println("Removed element is  :   "+element);
        que.remove();
    }
}
```

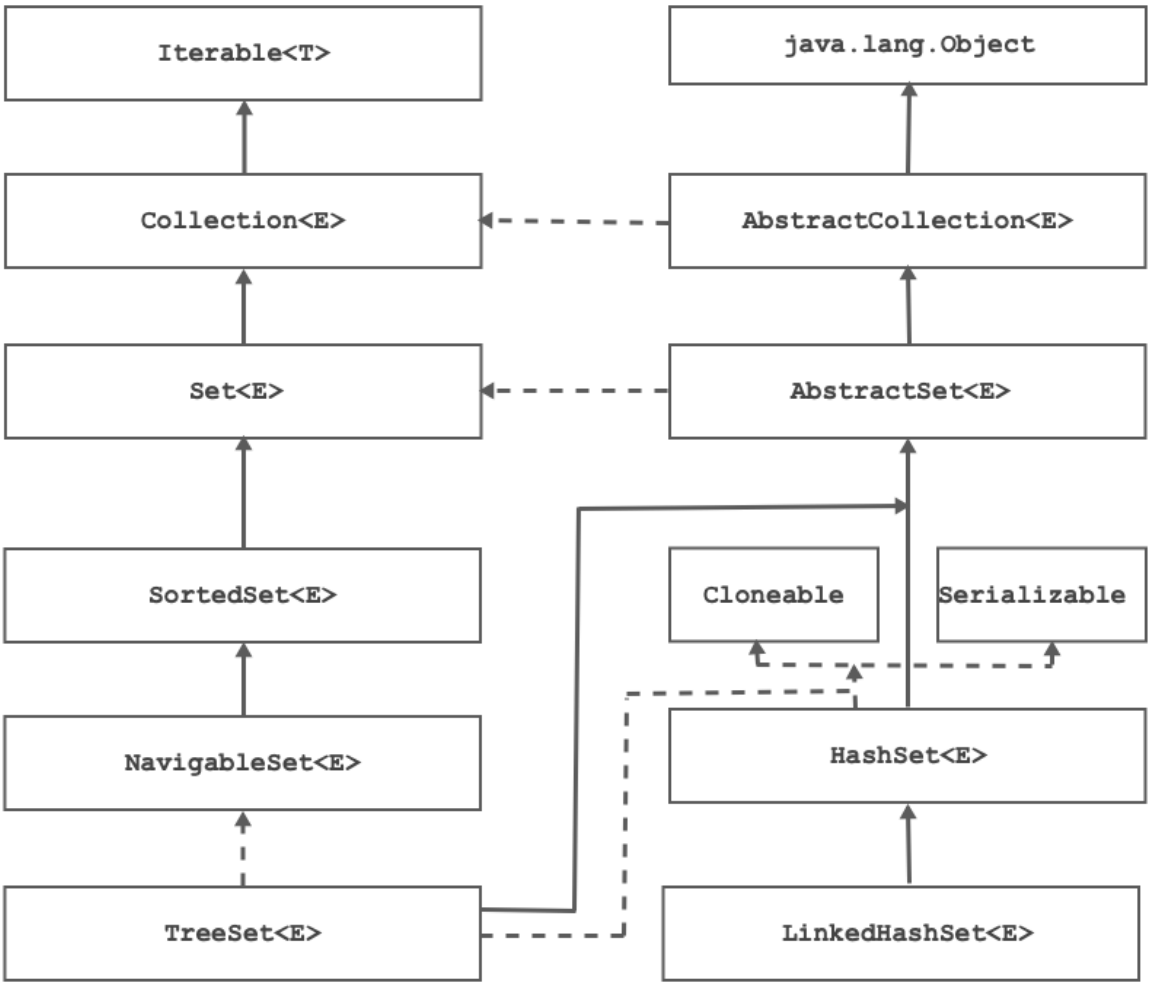
- Consider following code:

```
public static void main(String[] args) {
    Queue<Integer> que = new ArrayDeque<>();
    que.offer(10);
    que.offer(20);
    que.offer(30);
    que.offer(40);
    que.offer(50);
    //que.offer(null);  //Not Allowed

    Integer element = null;
    while( !que.isEmpty() ) {
        element = que.peek();
        System.out.println("Removed element is  :   "+element);
        que.poll();
    }
}
```

Deque

- It is sub interface of Queue interface.
- The name deque is short for "double ended queue" and is usually pronounced "deck".
- This interface is a member of the Java Collections Framework.
- It is introduced in JDK 1.6

Set



- It is sub interface of java.util.Collection interface.
- HashSet, LinkedHashSet, TreeSet, EnumSet are Set collections.
- Set Collections do not contain duplicate elements.
- This interface is a member of the Java Collections Framework.
- It is introduced in JDK 1.2
- Method names of Collection and Set interface are same. No new method is added in Set interafce.

What is the difference between List and Set?

- ArrayList, Vector, LinkedList are List Collections and HashSet, LinkedHashSet, TreeSet are Set collections
- List collections can contain duplicate elements but Set collections do not contain duplicate elements.
- List collections can contain null elements but not all Set collections contain null elements.
- We can traverse elements of List collection using ListIterator as well as Iterator but we can we can traverse elements of Set collection using Iterator only.
- All List collections are sequential collections but we can not give gurantee of order of elements in Set collection.

TreeSet

- It is a Set collection.
- It can not contain duplicate elements.

- It can not contain null elements.
- It contains data in sorted order.
- TreeSet implementation is based on TreeMap<K,V>.
- It is unsynchronized collection. Using Collections.synchronizedSortedSet() method we can make it synchronized.
- This class is a member of the Java Collections Framework.
- It is introduced in JDK 1.2
- Note: If we want to use TreeSet to store elements of non final type then non final type should implement Comparable interface.
- How to create instance of TreeSet

```
    public static void main(String[] args) {
    TreeSet<Integer> treeSet = new TreeSet<>();

    Set<Integer> set = new TreeSet<>(); //Upcasting

    Collection<Integer> collection = new TreeSet<>();   //Upcasting
  }
```

- How will you add elements inside TreeSet?

```
    public static void main(String[] args) {
    Set<Integer> set = new TreeSet<>(); //Upcasting
    set.add(50);
    set.add(10);
    set.add(30);
    set.add(20);
    set.add(40);
    System.out.println( set );  //[10, 20, 30, 40, 50]
  }
```

- Can we add duplicate elements inside TreeSet

```
    public static void main(String[] args) {
    Set<Integer> set = new TreeSet<>(); //Upcasting
    set.add(50);
    set.add(10);
    set.add(30);
    set.add(20);
    set.add(40);

    set.add(50);
```

```
        set.add(10);
        set.add(30);
        set.add(20);
        set.add(40);

        System.out.println( set );  //[10, 20, 30, 40, 50]
    }
```

- Can we add null element inside TreeSet?

```
    public static void main(String[] args) {
    Set<Integer> set = new TreeSet<>(); //Upcasting
    set.add(50);
    set.add(10);
    set.add(30);
    set.add(20);
    set.add(40);
    set.add(null);  //NullPointerException

    System.out.println( set );
    }
```

- Can we convert TreeSet into ArrayList?

```
Set<Integer> set = new TreeSet<>();   //Upcasting
set.add( 10 );
set.add( 20 );
set.add( 30 );
List<Integer> list = new ArrayList( set );
```

Algorithm

- In Computer science algorithm and data structure are two different branches.
- Data structure describes 2 things:
    - How to organize data inside RAM?
    - Which operations should be used to organize data inside RAM.
- Data structure can be linear / non linear.
- Well defined set of statements that we can use to solve real world common problems is called as algorithms.
- In the context of data structure algorithms can be searching / sorting algorithms.

Searching

- Searching refers to the process of finding location(index / reference) of an element in Collection.
- The collection of data may be in array, list, database, or any other data structure.
- There are several searching algorithms, each with its own strengths and weaknesses:

- Linear search
- Binary search
- Hashing
- Interpolation search
- Exponential search

Linear Search

- Linear search is a simple searching algorithm that sequentially searches each element in a collection until the desired item is found.
- Consider following code:

```
public static int linearSearch(int[] arr, int key) {
  if( arr != null ){
    for (int index = 0; index < arr.length; index++) {
        if (arr[ index ] == key)
            return index;
    }
  }
  return -1;
}
```

- Pros of linear search:
  - Simplicity: Linear search is easy to understand and implement, making it a good choice for small collections.
  - Flexibility: Linear search can be used with any type of collection, including arrays and linked lists.
  - No requirement for sorted data: Unlike some other search algorithms, linear search does not require the data to be sorted.
- Cons of linear search:
  - Time complexity: The worst-case time complexity of linear search is O(n), where n is the number of elements in the collection.
  - Inefficiency with large collections: Linear search is inefficient for large collections, as it requires checking every element in the collection
  - Not suitable for complex data structures: For complex data structures, such as trees or graphs

Binary search

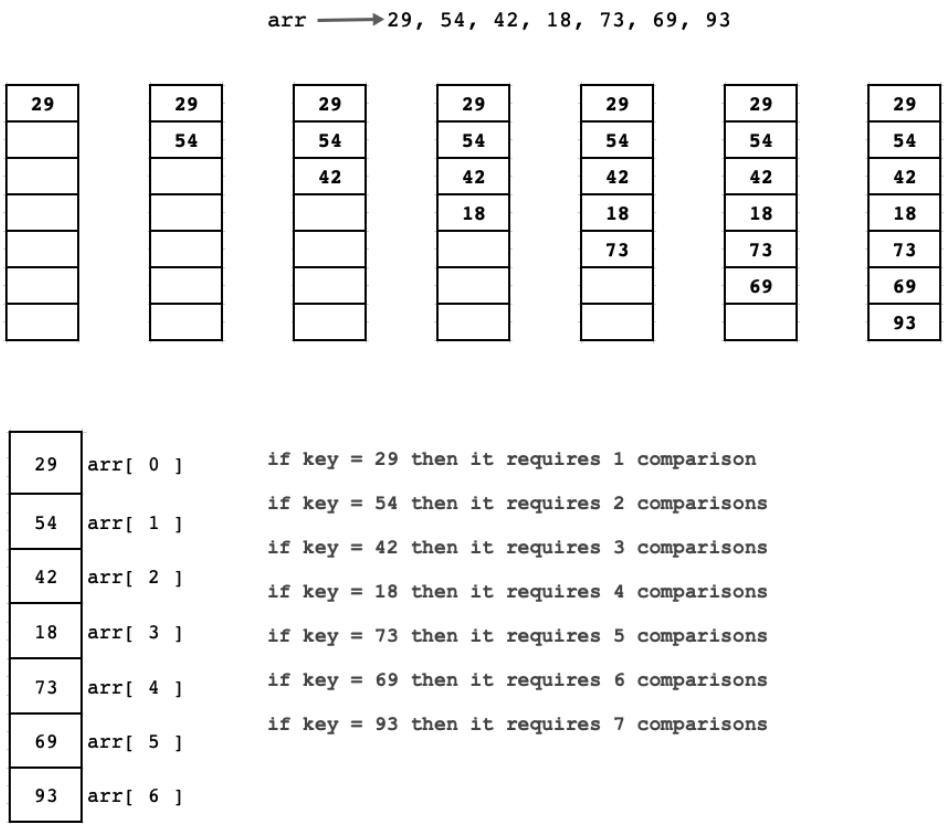
- Binary search is a searching algorithm used to search for a specific element in a sorted array.
- The basic idea of binary search is to divide the array into halves repeatedly until the desired element is found.
- Consider following exaple:

```
public static int binarySearch(int[] arr, int key) {
  int left = 0;
  int right = arr.length - 1;
  while (left <= right) {
    int mid = (left + right) / 2;
    if (arr[mid] == key)
        return mid;
    else if (arr[mid] < key)
        left = mid + 1;
      else
        right = mid - 1;
  }
  return -1;
}
```

- Pros of Binary Search:
    - Efficient: Binary search is an efficient algorithm, with a time complexity of O(log n) for a sorted array of size n. This is much faster than linear search, which has a time complexity of O(n).
    - If the array is already sorted, it can be a very efficient way to search for an element.
- Cons of Binary Search:
    - Requires Sorted Array: As mentioned, binary search requires the array to be sorted in advance. If the array is unsorted or frequently updated, then binary search may not be the best choice.
    - Requires Random Access: Binary search requires random access to the array elements, which is not available in some data structures such as linked lists.
    - Limited Applicability: Binary search is limited to searching for an element in a one-dimensional sorted array. It cannot be used to search for an element in a multidimensional array or a database table.

Hashing

- Consider numbers, 29,54,42,18,73,69,93 and insert it into array.



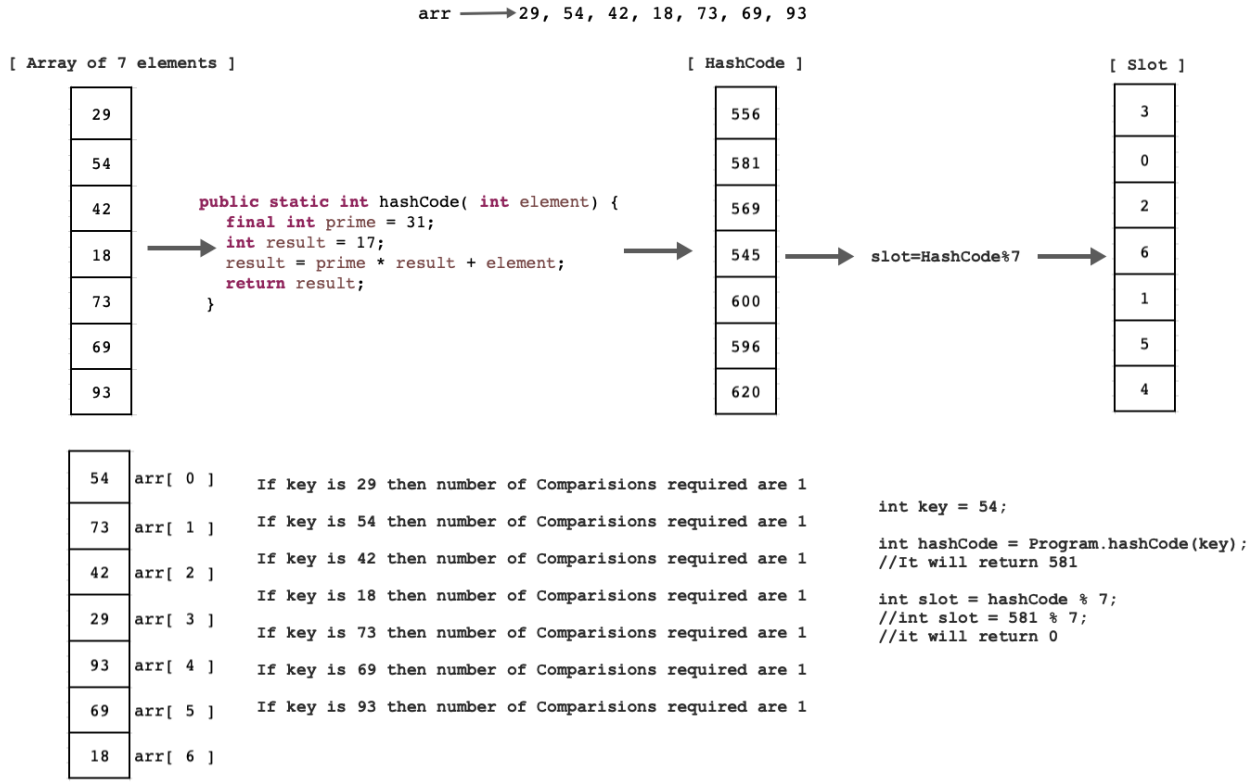
Note: Since we are adding elements sequentially, time required to search every element is different

- Hashing is a technique used to store and retrieve data in a fast and efficient manner.
- Hashing searching is based on hash code.
- Hashcode is not an index / address / reference. It is a logical integer number that can be generated by processing state of the instance.
- To generate hash code we should use hash function. Consider following example

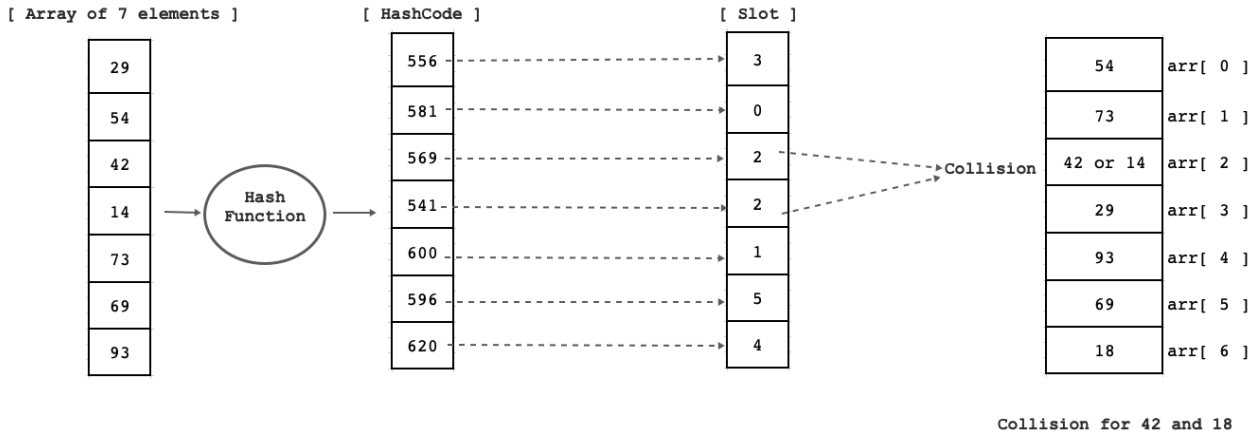
```
  public int hashCode( int element) {
  final int prime = 31;
  int result = 17;
  result = prime * result + element;
  return result;
}
```

- In the context of hashing, a slot refers to a location in the hash table where a key-value pair can be stored.

- In other words, hash code is required to generate index which is called as slot in hashing.

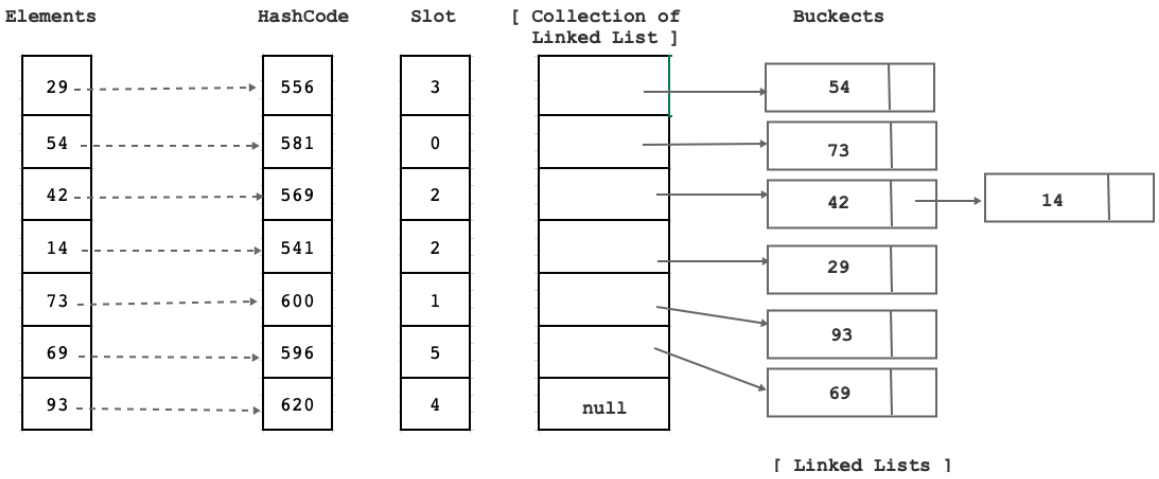


- By Processing state of the instance, if we get same slot then it is called as collision.



- To avoid collision, we should use collision resolution techniques:
    - Separate Chaining / Open Hashinng
    - Open Addressing /Close Hashing
        - Linear Probing
        - Qudratic Probing
        - Double Hashing

- Separate Chaining



- To avoid collision, we can maintain one collection(LinkedList/BST) per slot. It is called as bucket.
- Note: If we want to store any element of non primitive type inside Hashcode based collection then we should override equals and hashCode method inside non primitive type.
- Pros of Hashing
    - Fast retrieval: Hashing allows for constant-time retrieval of data, regardless of the size of the data set. This makes it a very efficient searching algorithm.
    - Flexibility: Hashing can be used to search for any type of data, including strings, numbers, and custom objects.
    - Easy to implement
    - Supports dynamic resizing: Hash tables can be resized dynamically to accommodate more data, without having to recreate the entire data structure.
- Cons of Hashing:
    - Hash collisions: Hash collisions occur when two different keys are mapped to the same hash value. This can result in slower lookup times and may require additional processing to resolve the collision.
    - Memory usage: Hash tables can consume a lot of memory, particularly if the data set is large. This can lead to performance issues if the available memory is limited.
    - Hash functions: The efficiency of hashing depends on the quality of the hash function used. Poor hash functions can lead to more collisions, slower retrieval times, and other issues.
- equals and hashCode are non final methods of java.lang.Object class.
- If we do not override equals method then super class's equals method will call. equals method of Object class do not compare state of instances. It compares state of references.
- Consider equals method definition of Object class:

```
public boolean equals(Object obj) {
    return (this == obj);
```

```
}
```

- hashCode is non final but native method of java.lang.Object class.
- Syntax:

```
public native int hashCode( );
```

- hashCode() method of java.lang.Object class convert memory address of instance into integer value. Hence even though state of the instances are same we get different hashCode. If we want hashCode based on state of the instance then we should override hashCode method inside class.
- Consider implementation:

```
  @Override
public int hashCode() {
  int prime = 31;
  int result = 1;
  result = result * prime + this.empid;
  return result;
}
```

HashSet

- It is hashCode based collection whose implementation is based on Hashtable.
- In HashSet elements get space according its hash code hence It doesn't give any gurantee about order of the elements.
- Since it is Set collection, it doesn't contain duplicate elements.
- HashSet can contain null element.
- It is unsynchronized collection. Using Collections.synchronizedSet() method we can consider it synchronized.
- This class is a member of the Java Collections Framework.
- It is introduced in JDK 1.2
- If we want to use any element of non final type inside HashSet then non final type should override equals() and hashCode() method.
- Instantiation of HashSet:

```
  public static void main(String[] args) {
  //HashSet<Integer> set = new HashSet<>();
```

```
    Set<Integer> set = new HashSet<>();
  }
```

- How will you add element inside HashSet?

```
  public static void main(String[] args) {
  Set<Integer> set = new HashSet<>();
  set.add(101);
  set.add(125);
  set.add(13);
  set.add(314);
  set.add(215);
  System.out.println(set);    //[101, 215, 314, 125, 13]
  }
```

- Is it possible to store duplicates and null elements?

```
  public static void main(String[] args) {
  Set<Integer> set = new HashSet<>();
  set.add(101);
  set.add(125);
  set.add(13);
  set.add(314);
  set.add(215);
  set.add(null);

  set.add(101);
  set.add(125);
  set.add(13);
  set.add(314);
  set.add(215);
  set.add(null);
  System.out.println(set);    //[null, 101, 215, 314, 125, 13]
  }
```

LinkedHashSet

- It is hashCode based collection whose implementation is based on Hashtable and LinkedList.
- During traversing it maintains order of lements.
- Since it is Set collection, it doesn't contain duplicate elements.
- HashSet can contain null element.
- It is unsynchronized collection. Using Collections.synchronizedSet() method we can consider it synchronized.
- This class is a member of the Java Collections Framework.
- It is introduced in JDK 1.4
- If we want to use any element of non final type inside HashSet then non final type should override equals() and hashCode() method.

Dictionary<K,V>

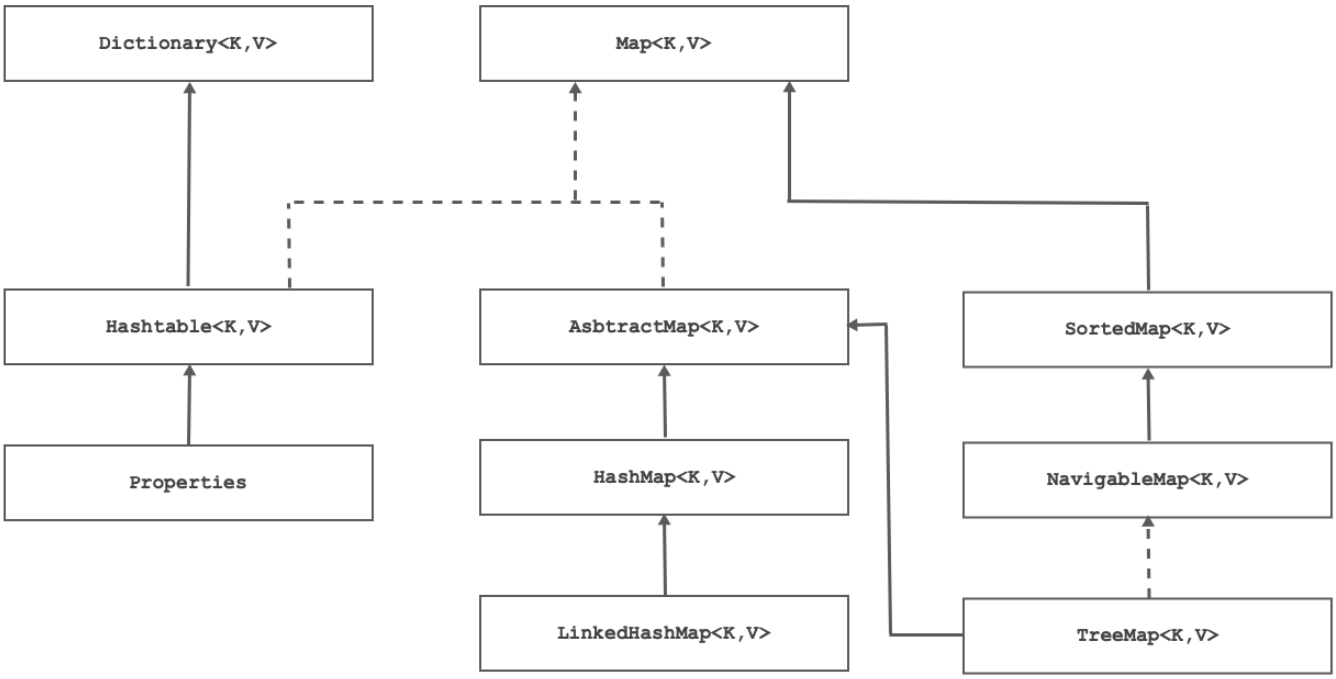
- Dictionary<K,V> is abstract class declared in java.util package.
- We can use it store elements in key value pair format.
- Hashtable is sub class of Dictionary<K,V> class.
- It was introduced in JDK 1.0
- Method Summary
    - public abstract V put(K key, V value)
    - public abstract int size()
    - public abstract V get(Object key)
    - public abstract V remove(Object key)
    - public abstract boolean isEmpty()
    - public abstract Enumeration keys()
    - public abstract Enumeration elements()
- Consider example of Dictionary class:

```
import java.util.Dictionary;
import java.util.Enumeration;
import java.util.Hashtable;
public class Program {
  public static Dictionary<Integer, String> getDictionary( ){
    Dictionary<Integer, String> d  = new Hashtable<>(); //Upcasting
    d.put(1,"PreDAC");
    d.put(2,"DAC");
    d.put(3,"DMC");
    d.put(4,"DIVESD");
    d.put(5,"DESD");
    d.put(6,"DBDA");
    return d;
  }
  private static void countAndPrintEntries(Dictionary<Integer, String>
d) {
    System.out.println("Count of entries    :   "+d.size());
  }
  private static void printKeys(Dictionary<Integer, String> d) {
    Enumeration<Integer> keys = d.keys();
    Integer key = null;
    while( keys.hasMoreElements()) {
      key = keys.nextElement();
      System.out.println( key );
    }
  }
  private static void printValues(Dictionary<Integer, String> d) {
    Enumeration<String> values = d.elements();
    String value = null;
    while( values.hasMoreElements()) {
      value = values.nextElement();
      System.out.println(value);
    }
  }
  private static void printValue(Dictionary<Integer, String> d, int
```

```
courseId) {
    Integer key = new Integer( courseId );
    String value = d.get(key);
    if( value != null )
      System.out.println(key+"  "+value);
    else
      System.out.println("Invalid key");
  }
  private static void removeEntry(Dictionary<Integer, String> d, int
courseId) {
    Integer key = new Integer( courseId );
    String value = d.remove(key);
    if( value != null )
      System.out.println(key+"  "+value+" is removed");
    else
      System.out.println("Invalid key");
  }
  public static void main(String[] args) {
    Dictionary<Integer, String> d = Program.getDictionary();
    //Program.countAndPrintEntries( d );
    //Program.printKeys( d );
    //Program.printValues( d );
    //Program.printValue( d, 2 );
    Program.removeEntry( d, 200);
  }
}
```

- This class is obsolete. New implementations should implement the Map interface, rather than extending this class.

Map<K,V>



- It is Key/Value pair interface which is declared in java.util packgae.

- It is member of Collection framework but it doesn't extend Collection interface.
- This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.
- Hashtable<K,V>, HashMap<K,V>, LinkedHashMap<K,V>, TreeMap<K,V> etc. are map collections.
- If we want to insert any element/value inside mau collection then we should provide key for it.
- In map collection we can insert duplicate value but we can not insert duplicate key.
- This interface is a member of the Java Collections Framework.
- It was introduced in JDK 1.2
- Map.Entry<K,V> is nested interface of Map interface.
- (key-value pair) instance is also called as entry.
- Methods of Map.Entry<K,V> interface
    - K getKey()
    - V getValue()
    - V setValue(V value)
- Method Summary of Map<K,V> interface
    - boolean isEmpty()
    - V put(K key, V value)
    - void putAll(Map<? extends K,? extends V> m)
    - int size()
    - boolean containsKey(Object key)
    - boolean containsValue(Object value)
    - V get(Object key)
    - V remove(Object key)
    - void clear()
    - Set keySet()
    - Collection values()
    - Set<Map.Entry<K,V>> entrySet()

Hashtable<K,V>

- It is a sub class of Dictionary<K,V> class and it implements Map<K,V>
- Since it is Map collection, we can not insert duplicate keys but we can insert duplcate value.
- In Hashtable key and value can not be null.
- It is synchronized collection.
- It is introduced in JDK 1.0.
- If we want to use any element of non primitive type inside Hashtable then non primitive type should override equals and hashCode method.

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Hashtable;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
```

```
import java.util.Set;

public class Program {
  private static Map<Integer, String> getMap() {
    Map<Integer, String> map = new Hashtable<>();
    map.put(1, "DAC");
    map.put(2, "DMC");
    map.put(3, "DESD");
    map.put(4, "DBDA");
    map.put(5, "DIVESD");
    return map;
  }
  private static void printAndCountEntries(Map<Integer, String> map) {
    System.out.println("Entries Count   :   "+map.size());
  }
  private static void printKeys(Map<Integer, String> map) {
    Set<Integer> keys = map.keySet();
    for (Integer key : keys) {
      System.out.print(key+"    ");
    }
    System.out.println();
  }
  private static void printValues(Map<Integer, String> map) {
    Collection<String> values = map.values();
    for (String value : values) {
      System.out.print(value+"  ");
    }
    System.out.println();
  }
  private static void printEntries(Map<Integer, String> map) {
    Set<Entry<Integer, String>> entries = map.entrySet();
    for (Entry<Integer, String> entry : entries)    {
      System.out.println(entry.getKey()+"   "+entry.getValue());
    }
  }
  private static void printEntry(Map<Integer, String> map, int id) {
    Integer key = new Integer(id);
    if( map.containsKey(key)) {
      String value = map.get(key);
      System.out.println( key+" "+value);
    }else
      System.out.println(key+" not found");
  }
  private static void removeEntry(Map<Integer, String> map, int id) {
    Integer key = new Integer(id);
    if( map.containsKey(key)) {
      String value = map.remove(key);
      System.out.println( key+" "+value+" is removed");
    }else
      System.out.println(key+" not found");
  }
  public static void main(String[] args) {
    Map<Integer, String> map = Program.getMap( );
    //Program.printAndCountEntries( map );
```

```
        //Program.printKeys( map );
        //Program.printValues( map );
        //Program.printEntries( map );
        //Program.printEntry( map, 3 );
        //Program.removeEntry( map, 3 );

        List<String> list = new ArrayList<>(map.values());
        System.out.println(list);
    }
}
```