Collections

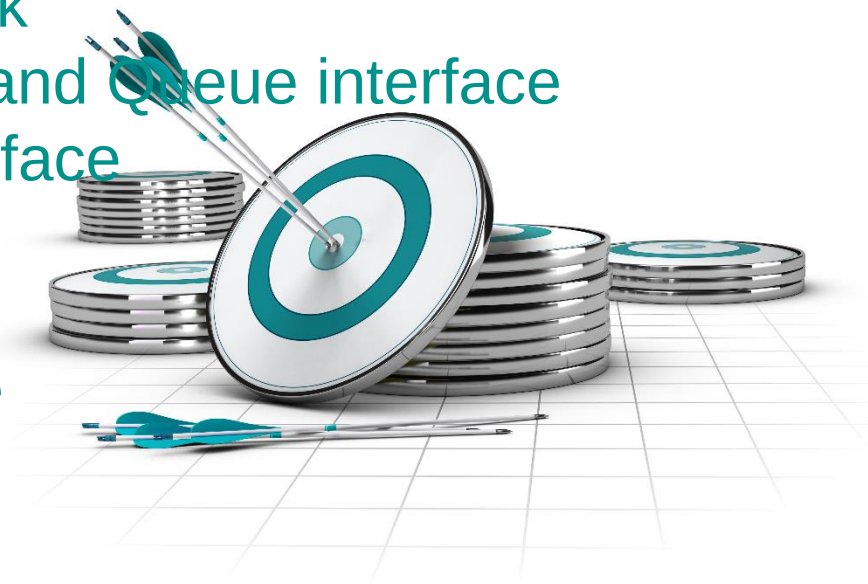
INTRODUCTION

Collections

LEARNING OBJECTIVES

At the end of this lesson, you will be able to:

- List the classes/interfaces in Collection Framework
- Distinguish and use concrete classes of List, Set and Queue interface
- Distinguish and use concrete classes of Map interface
- Understand Legacy classes in Collections API
- Use Collections Utility class
- Implement Comparable and Comparator Interface



CONCEPT

Collections API

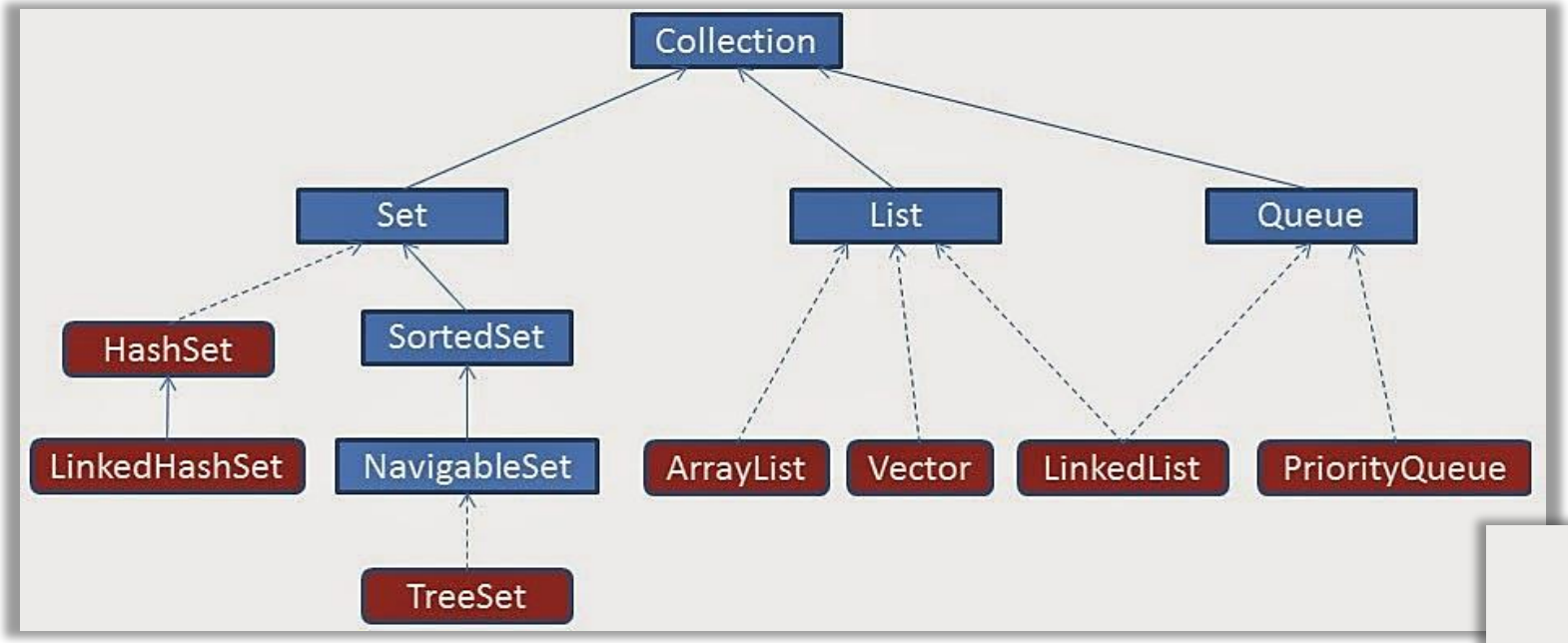
INTRODUCTION TO COLLECTIONS

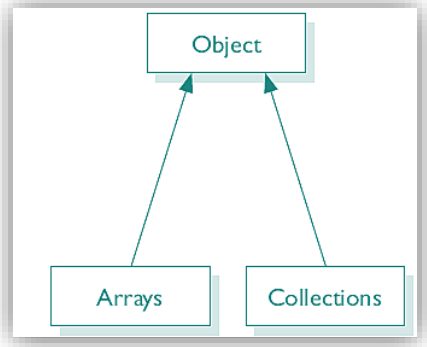
- Collection
  - represents a group of objects of similar type, known as its *elements*
  - provide flexibility to add or remove objects
  - provide capability to store objects in an ordered way
  - provide capability to prevent duplicate objects
  - provides capability to store objects with key-value pairing

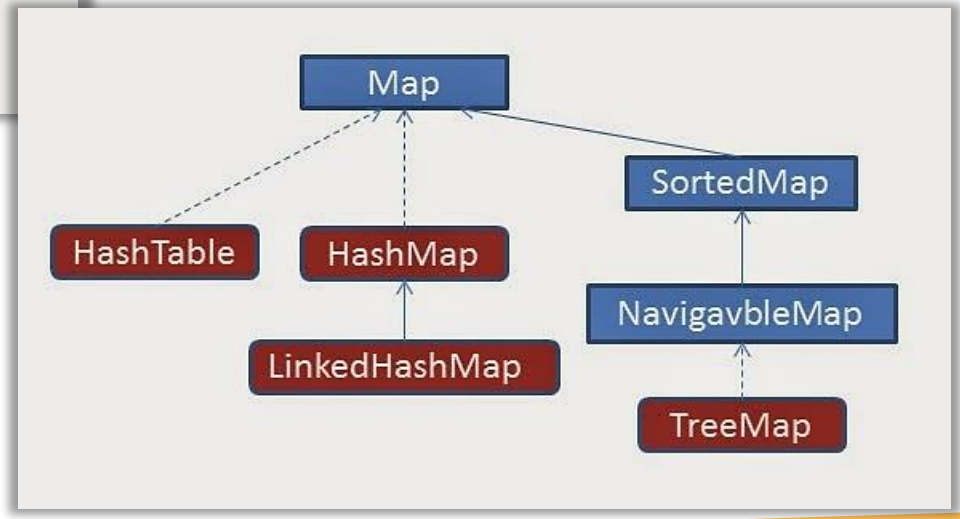
Basic operations

Add objects	Remove objects	Search objects	Retrieve objects	Iterate

Collections Framework







COLLECTION INTERFACE

- Root interface in Collection Hierarchy
- Does not have any direct implementation
- Extended by interface like List, Set and Queue
- Collection framework classes/interface belong to java.util package

Key Methods :

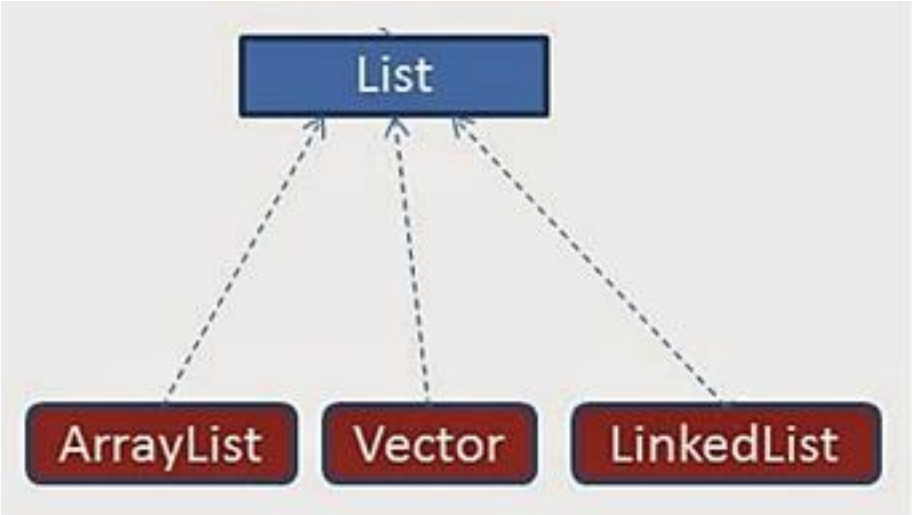
add(element e)	remove(Element o)
addAll(Collection c)	
removeAll(Collection c)	
contains(Element o)	isEmpty()
size()	toArray()
iterator()	clear()

LIST

- an ordered collection (aka *sequence*)
- orders objects based on the index
- allows duplicate elements
- precise control over where in the list each element is inserted
- provides methods for positional (indexed) access of list elements

```
                get(int index)
indexOf(Element o)
                add(int index, Element o)
          remove(int Index)
                lastIndexOf(Element o)
```

LIST CONCRETE IMPLEMENTATIONS



- ArrayList
  - Dynamically growing Array (initial capacity 10)
  - Elements stored contiguously in memory
  - Methods of ArrayList are not synchronized

- Features
  - Effecient iteration and random access
  - Insertion/Deletion of elements at specific indexes from a large list is expensive

LIST CONCRETE IMPLEMENTATIONS

- Vector
  - Same as an ArrayList, but has synchronized methods for thread safety

- Stack is a subclass of Vector that implements a standard last-in, first-out stack (LIFO).
  - Elements can retrieved, inserted, or removed from the top of the stack.

Methods of Stack :

+empty() : boolean
+peek() : Object
+pop() : Object
+push (element:Object) : void
+search (element:Object) : int

LIST CONCRETE IMPLEMENTATIONS

- LinkedList
  - Elements are doubly-linked to one another
  - Each Node holds Data and Address of previous and next node

- Features
  - Insertion/Deletion of elements is less expensive compared to ArrayList
  - Iteration over linked list is required, to retrieve any element
  - Gives extra methods for adding and removing elements from the beginning or end

DIFFERENCE BETWEEN ARRAYLIST AND VECTOR

ArrayList

- ArrayList is not synchronized.
- ArrayList increments 50%of current array size if number of element exceeds from its capacity.
- ArrayList is faster as its methods are non-synchronized
- ArrayList uses Iterator interface to traverse the elements

Vector

- Vector is synchronized.
- Vector increments 100% means doubles the array size if total number of element exceeds its capacity.
- Vector is slow because its methods are synchronized
- Vector uses Enumeration interface to traverse the elements. (can use Iterator also)

ARRAYLIST – BEFORE JAVA VERSION 5

- ArrayList class as defined in Java API for holding objects before version 5

```
Public class ArrayList extends AbstractList implements List
      public boolean add(Object o)
            Appends the specified element to the end of this list
      public Object get(int index)
            Returns the element at the specified position in this list.
```

```
ArrayList myList = new ArrayList();
```

ARRAYLIST – BEFORE JAVA VERSION 5

```
import java.util.*;
class TestList{
 public static void main(String[] args) {
   Integer i1 = new Integer(10);
   String s1  = "Hello";
   ArrayList list = new ArrayList();
   list.add(i1);
   list.add(s1);

   Object o1 = list.get(1);
   String s2 = (String)o1;
   System.out.println(s2.length());
 }
}
```

DrawBacks

1. Any object can be added to the list
2. If the List is supposed to hold one particular type of object, it cant be restricted
3. The object retrieved from the list has to be casted back to required type before use
4. Type casting is unsafe

ARRAYLIST WITH GENERICS

Generics

- Provides capability to parameterize Types
- Helps define class or methods independent of c
- Diamond operator (<>) is used for type Inferenc

E is Type parameter
Elements added to the List can be of only Type E or its subtypes
If the List is of Type String, Only Strings can be stored in the list
No Type casting required as compiler knows the type of object stored

ArrayList class using Generics is defined in JAVA A

```
Public class ArrayList<E> extends AbstractList<E> implements List<E>
        public boolean add(E e)
                Appends the specified element to the end of this list
        public E get(int index)
                Returns the element at the specified position in this list.
```

```
ArrayList<String> myList = new ArrayList<String>();
```

```
ArrayList<String> myList = new ArrayList<>(); // From Java ver 7
```

ARRAYLIST WITH GENERICS

DEMO

```
import java.util.*;
class TestList{
        public static void main(String[] args) {
          Integer i1 = new Integer(10);
          String s1  = "Hello";
          ArrayList<String> list = new ArrayList<String>();
         //list.add(i1);    - Compile Error
          list.add(s1);

                    String s3 = list.get(1);
        System.out.println(s3.length()); - No Casting required
    }
}
```

ITERATOR INTERFACE

- Enables traversing through a collection in forward direction
- Remove elements from the collection selectively while iterating
- The Iterator Interface has three methods
  - boolean hasNext() - Check the existense of next element
  - object next() - Retrieves next element
  - void remove() - Removes last element obtained using next() method

```
List<String> myList = new ArrayList<String>();
Iterator it = myList.iterator();
```

- ListIterator -
  - An iterator for lists to traverse the list in either direction and modify the list during iteration

SORTING LISTS

- Collections.sort() method is used to sort Lists of objects
- A class must implement Comparable interface, so that objects its objects can be compared
- Comparable interface has one method compareTo which must be overridden by the class for comparing objects
- JVM calls the compareTo method automatically when objects need to be compared
- Comparable interfaces provides the natural ordering of elements in a collection

SORTING LISTS

- compareTo method defines the criteria for comparing objects. It takes object to be compared as parameter.

  compareTo(T o)

- The compareTo() method returns an int
  - negative thisObject < passedObject
  - zero thisObject == passedObject
  - positive thisObject > passedObject

```
class Employee implements Comparable<Employee> {
       public int compareTo(Employee e1) {
              return (this.empId - e1.empId);
  } }
```

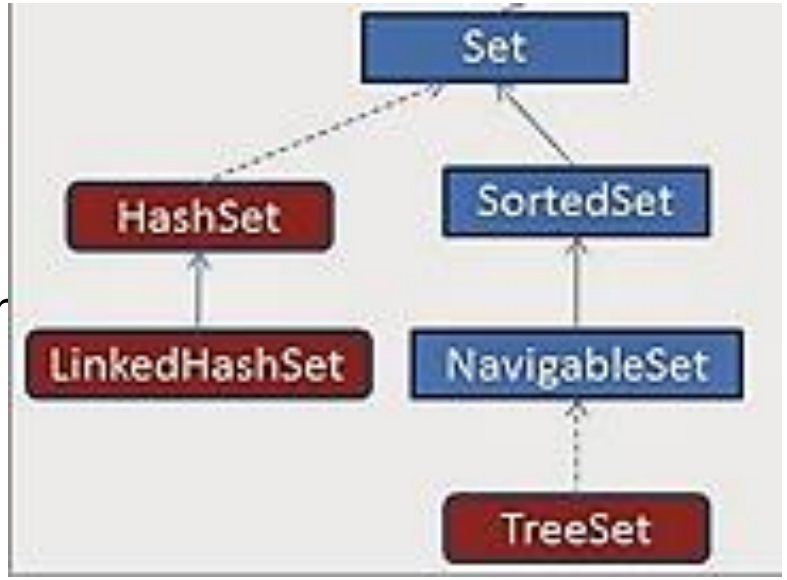
STACK

```
Stack<String> st = new Stack<String>();

st.push("a");
st.push("b");
st.push("c");

System.out.println(st.pop());
System.out.println(st.pop());
System.out.println(st.pop());
```

SET

- Set is a Collection that cannot contain duplicate elements
- Sets contain no pair of elements e1 and e2 such that e1.equals(e2)
- Java platform contains three general-purpose Set implementations: HashSet, TreeSet and LinkedHashSet
- SortedSet is a Set that provides a total ordering on its elements. The elements are ordered using their natural ordering

SET CONCRETE IMPLEMENTATIONS



- HashSet
  - Unsorted, Unordered Set
  - Use when duplicates are not allowed and order is not im
- LinkedHashSet
  - Ordered, Unsorted Set
  - HashSet that maintains a doubly-linked List across all elements
  - Use when duplicates are not allowed and order is important
- TreeSet
  - Keeps the elements in Sorted Order
  - Elements will be in ascending order, according to natural order
  - Use when duplicates are not allowed and sorted order is important

SET CONCRETE IMPLEMENTATIONS

- HashSet
  - Uses hashcode and equals() to prevent duplicates
  - Uses hashcode of object to insert and search elements
  - Elements of Hashset need to override hashCode() method and equals() method to provide object equality

- TreeSet
  - Uses Comparable /Comparator for keeping the elements in sorted Order and checking the equality

WHAT SET TO CHOOSE AND WHEN

- HashSet is a good choice for representing sets if element ordering is not important
- if ordering is important, LinkedHashSet or TreeSet are better choices. However they come with an additional speed and space cost.
- Iteration over a LinkedHashSet is generally faster than iteration over a HashSet.
- Tree-based data structures get slower as the number of elements get larger.
- Since TreeSet keeps its elements sorted, it can offer other features, such as the first and last methods, that is, the lowest and highest elements in a set, respectively.

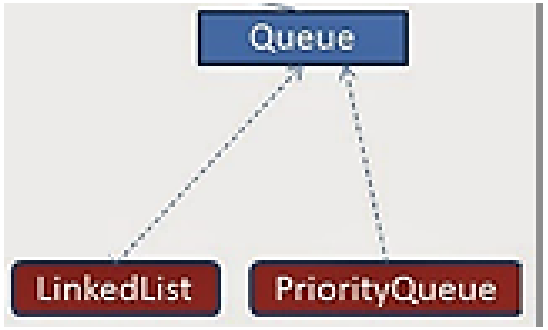
QUEUE

- Objects arranged in the order in which they are to be processed
- Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner
- Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations

	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

offer– used to add element to the tail of the Queue
poll – used to get elements from the head of the Queue
peek – used to review the element at the head of the Queue

QUEUE CONCRETE IMPLEMENTATIONS



- LinkedList
  - First in First out

```
Queue<String> myQueue = new LinkedList<String>();
myQueue.offer("first");
```

- Priority Queue
  - Stores element based on priority set by the Comparable/Comparator

```
PriorityQueue<String> myQueue = new PriorityQueue<String>();
myQueue.offer("first");
```

MAP

- An object maps a unique key to a specific value, where key and value are objects
- Elements of Map are stored as key-value pair
- map cannot contain duplicate keys

- Allows to
  - Search for a value based on the key
  - Get a collection of just Keys using keySet method
  - Get a collection of just values using values method

Key Methods :

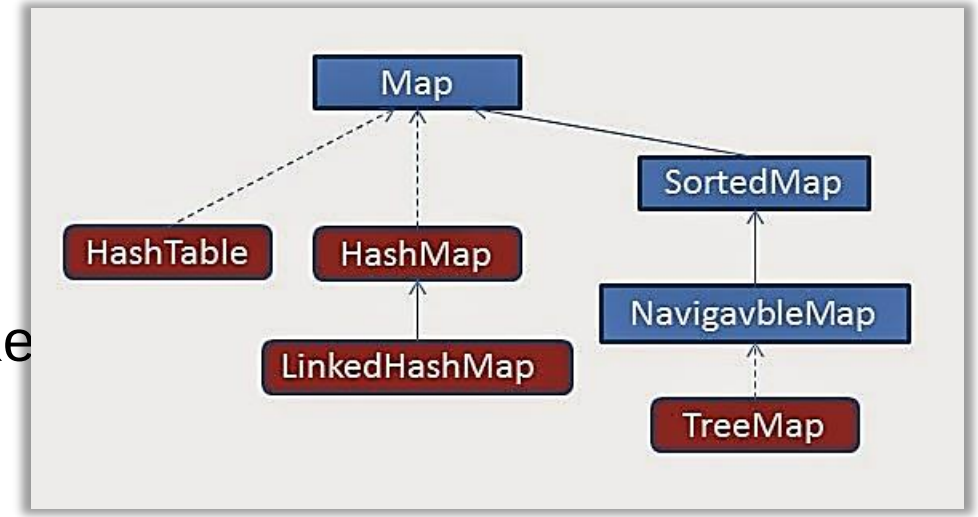
put(Key k, Value v) get(Key)

remove(Key)

keySet()

values()

MAP CONCRETE IMPLEMENTATIONS



- HashMap
  - Unsorted, Unordered Map
  - Relys on equals() and hashCode for equality of Ke
- LinkedHashMap
  - Unsorted Map,Ordered on keys
  - Relys on equals() and hashCode for equality of Keys
- TreeMap
  - Sorted based on Keys
  - Sorted using comparable or comparator

SORTING USING COMPARATOR

- Comparator interface provides the alternate ordering to sort objects in a collection
- Class implementing Comparator interface has to override compare method
- Compare method is used to define the criteria of comparison between two objects
- It takes two objects as parameter and returns an int
  - int compare(T o1, T o2)

ENUMERATION INTERFACE

- Enumeration interface defines a way to traverse all the members of a collection of objects.
- Enumeration is only used in legacy classes such as Vector and Properties, this interface has been superceded by Iterator
- **Methods**
  - hasMoreElements()
    - Checks to see if there are more elements
    - Then returns a Boolean
  - nextElement()
    - Checks to see if there are more elements
    - If yes, then returns the next element as an object

If there aren't any more elements when nextElement() is called, the runtime NoSuchElementExceptionwill be thrown.

Generics

- Multiple Parameters
  - A class can use multiple type parameters

```
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>{
        public V put(K key, V value)
        ……}
```

- A generic can restrict the type of object that can be used as the parameter by using Bounded Types
  - UpperBound
    - ArrayList<? extends Employee> means ArrayList of any object of class which extends Employee class can be passed as argument to printName Method
  - LowerBound
    - ArrayList<? super Manager> means ArrayList of any object of class Manager and its supertypes can be passed as argument to printName

Generics

- Method declarations can be made generic even if the class doesn't use generic types

```
public static <T extends Employee> printName(T t){
                System.out.println(t.name);
        }
```

- Generic Interfaces

```
interface List<T>{
                public void put(T element, int position);
                public T get(int position);
        }
```

- Generic Type parameter cannot be of primitive type

SUMMARY

In this lesson, you've learned to:

- Describe Collection Framework
- Distinguish between various Collection Implementations
- Use appropriate Collection implementations based on the need
- Compare and Sort objects in a Collection