Parul® University

# Object Oriented Programming with JAVA

Ms.Vaishalee Joishar,Cyber Secutiy Trainer

Parul®
University

DIGITAL LEARNING CONTENT

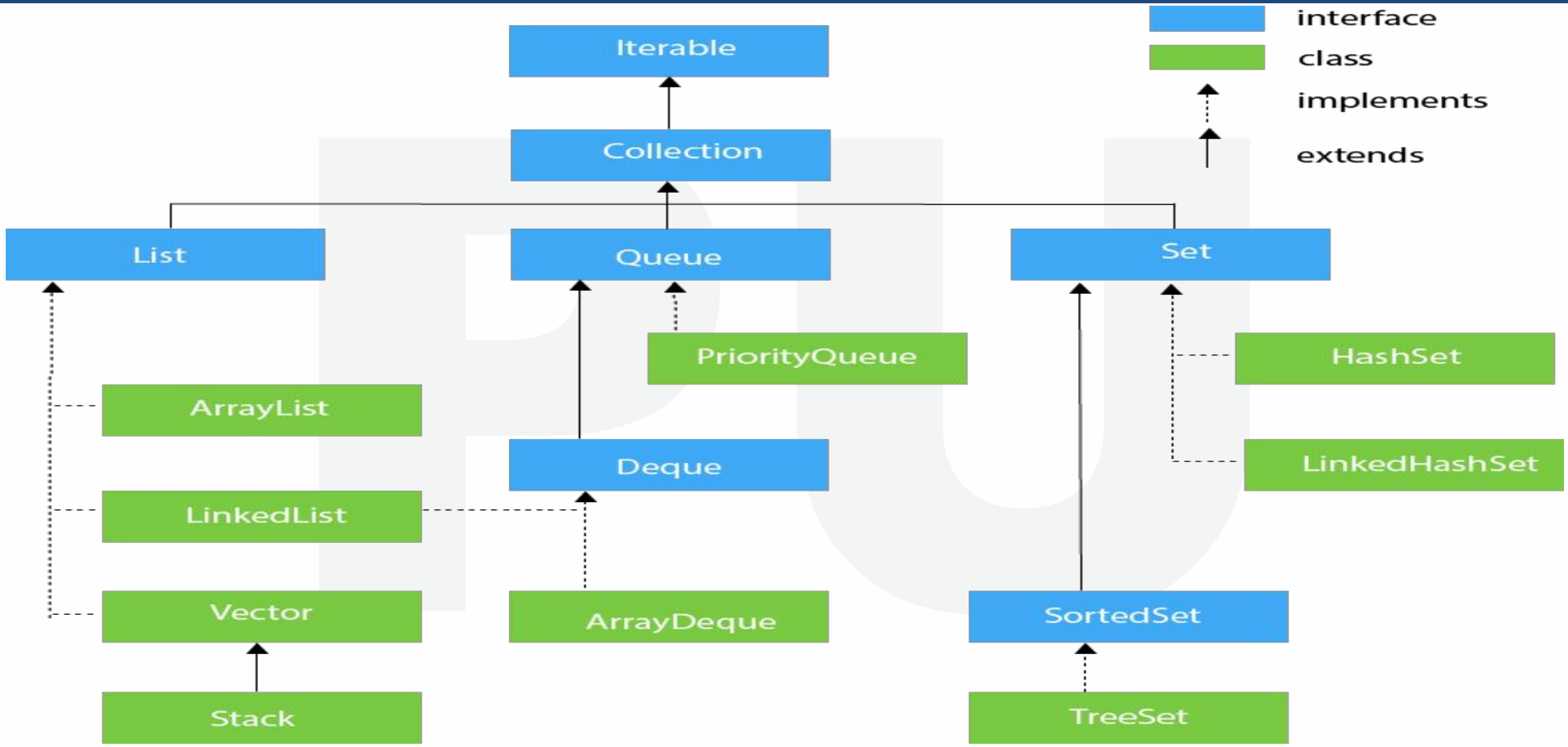
# CHAPTER-10

# Collections Framework


# Collection

- The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

# Hierarchy of Collection Framework



# Collection Interface - Methods

| Sr. | Method & Description |
|---|---|
| 1 | boolean add(E e)<br>It is used to insert an element in this collection. |
| 2 | boolean addAll(Collection<? extends E> c)<br>It is used to insert the specified collection elements in the invoking collection. |
| 3 | void clear()<br>It removes the total number of elements from the collection. |
| 4 | booelan contains(Object element)<br>It is used to search an element. |
| 5 | boolean containsAll(Collection<?> c)<br>It is used to search the specified collection in the collection. |
| 6 | boolean equals(Object obj) |

# Cont...

| Sr. | Method & Description |
|---|---|
| 7 | boolean isEmpty()<br>Returns true if the invoking collection is empty. Otherwise returns false. |
| 8 | Iterator iterator()<br>It returns an iterator. |
| 9 | boolean remove(Object obj)<br>Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false. |
| 10 | boolean removeAll(Collection<?> c)<br>It is used to delete all the elements of the specified collection from the invoking collection. |
| 11 | boolean retainAll(Collection<?> c)<br>It is used to delete all the elements of invoking collection except the specified collection. |

# List Interface

- The **`List`** interface extends **`Collection`** and declares the behavior of a collection that stores a sequence of elements.
- Elements can be inserted or accessed by their position in the list, using a zero-based index.
- A list may contain duplicate elements.
- List is a generic interface with following declaration

```
interface List<E>
```

  - where `E` specifies the type of object.

# List Interface - Methods

| Sr. | Method & Description |
|---|---|
| 1 | void add(int index, Object obj)<br>Inserts obj into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. |
| 2 | boolean addAll(int index, Collection c)<br>Inserts all elements of c into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise. |
| 3 | Object get(int index)<br>Returns the object stored at the specified index within the invoking collection. |
| 4 | int indexOf(Object obj)<br>Returns the index of the first instance of obj in the invoking list. If obj is not an element |

# List Interface – Methods (Cont.)

| Sr. | Method & Description |
|---|---|
| 5 | ListIterator listIterator( )<br>Returns an iterator to the start of the invoking list. |
| 6 | ListIterator listIterator(int index)<br>Returns an iterator to the invoking list that begins at the specified index. |
| 7 | Object remove(int index)<br>Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one |
| 8 | Object set(int index, Object obj)<br>Assigns obj to the location specified by index within the invoking list. |
| 9 | List subList(int start, int end)<br>Returns a list that includes elements from start to end-1 in the invoking list. Elements in |

# List Interface (example)

```
1.  import java.util.*;
2.  public class CollectionsDemo {
3.     public static void main(String[] args)
    {
4.        List a1 = new ArrayList();
5.        a1.add("Sachin");
6.        a1.add("Sourav");
7.        a1.add("Shami");
8.              System.out.println("ArrayList
    Elements");
9.        System.out.print("\t" + a1);
10.       List l1 = new LinkedList();
11.       l1.add("Mumbai");
12.       l1.add("Kolkata");
13.       l1.add("Vadodara");
14.       System.out.println();
15.         System.out.println("LinkedList
    Elements");
16.       System.out.print("\t" + l1);
17.    }     }
```

```
G:\Darshan\Java 2019\PPTs\HAD\Programs>ja
ArrayList Elements
        [Sachin, Sourav, Shami]
LinkedList Elements
        [Mumbai, Kolkata, Vadodara]
G:\Darshan\Java 2019\PPTs\HAD\Programs>
```

# Iterator

- Iterator interface is used to cycle through elements in a collection, eg. displaying elements.
- ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.
- Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.
- To use an iterator to cycle through the contents of a collection, follow these steps:

1. Obtain an iterator to the start of the collection by calling the collection's iterator( ) method.
2. Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.
3. Within the loop, obtain each element by calling next( ).

# Iterator - Methods

| Sr. | Method & Description |
|---|---|
| 1 | boolean hasNext()<br>Returns true if there are more elements. Otherwise, returns false. |
| 2 | E next()<br>Returns the next element. Throws NoSuchElementException if there is not a next element. |
| 3 | void remove()<br>Removes the current element. Throws IllegalStateException if an attempt is made to call remove() that is not preceded by a call to next() |

# Iterator - Example

```
1.  import java.util.*;
2.  public class Main {
3.  public static void main(String
    args[]) {
4.  ArrayList<String> al = new
    ArrayList<String>();
5.  al.add("C");
6.  al.add("A");
7.  al.add("E");
8.  al.add("B");
9.  al.add("D");
10. al.add("F");
11. System.out.println(al);
12. System.out.println("Contents
    of list: ");
13. Iterator<String> itr =
    al.iterator();
14. while(itr.hasNext()) {
15. Object element = itr.next();
16. System.out.println(element + "
    "); }
17.  }
18. }
```

```
G:\Darshan\Java 2019\PPTs\HAD\Pr
Contents of list: C A E B D F
```

# Comparator

- Comparator interface is used to set the sort order of the object to store in the sets and lists.

- The Comparator interface defines two methods: compare( ) and equals( ).

```
int compare(Object obj1, Object obj2)
```

  - `obj1` and `obj2` are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if `obj1` is greater than `obj2`. Otherwise, a negative value is returned.

```
boolean equals(Object obj)
```

  - `obj` is the object to be tested for equality. The method returns true if `obj` and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false.

# Comparator Example

```
public class ComparatorDemo {
    public static void main(String args[]){
        ArrayList<Student> al=new ArrayList<Student>();
        al.add(new Student("Vijay",23));
        al.add(new Student("Ajay",27));
        al.add(new Student("Jai",21));
        System.out.println("Sorting by age");
        Collections.sort(al,new AgeComparator());
        Iterator<Student> itr2=al.iterator();
        while(itr2.hasNext()){
                Student st=(Student)itr2.next();
                System.out.println(st.name+" "+st.age);
        }
    }
}
```

```
class AgeComparator implements
Comparator<Object>{
    public int compare(Object o1,Object o2){
        Student s1=(Student)o1;
        Student s2=(Student)o2;
        if(s1.age==s2.age) return 0;
        else if(s1.age>s2.age) return 1;
        else return -1;
    }
}
```

```
import java.util.*;
class Student {
    String name;
    int age;
    Student(String name, int
    age){
        this.name = name;
        this.age = age;
    }
}
```

# Vector Class

- `Vector` implements a dynamic array.

- It is similar to `ArrayList`, but with two differences:
  - Vector is synchronized.
  - Vector contains many legacy methods that are not part of the collection framework

- Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

- Vector is declared as follows:

```
Vector<E> = new Vector<E>;
```

# Vector - Constructors

| Sr. | Constructor & Description |
|---|---|
| 1 | Vector( )<br>This constructor creates a default vector, which has an initial size of 10 |
| 2 | Vector(int size)<br>This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size: |
| 3 | Vector(int size, int incr)<br>This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward |
| 4 | Vector(Collection c)<br>creates a vector that contains the elements of collection c |

# Vector - Methods

| Sr. | Method & Description |
|---|---|
| 5 | boolean containsAll(Collection c)<br>Returns true if this Vector contains all of the elements in the specified Collection. |
| 6 | Enumeration elements()<br>Returns an enumeration of the components of this vector. |
| 7 | Object firstElement()<br>Returns the first component (the item at index 0) of this vector. |
| 8 | Object get(int index)<br>Returns the element at the specified position in this Vector. |
| 9 | int indexOf(Object elem)<br>Searches for the first occurence of the given argument, testing for equality using the equals method. |

# Vector – Method (Cont.)

| Sr. | Method & Description |
|---|---|
| 10 | Object lastElement()<br>Returns the last component of the vector. |
| 11 | int lastIndexOf(Object elem)<br>Returns the index of the last occurrence of the specified object in this vector. |
| 12 | Object remove(int index)<br>Removes the element at the specified position in this Vector. |
| 13 | boolean removeAll(Collection c)<br>Removes from this Vector all of its elements that are contained in the specified Collection. |

# Stack

- Stack is a subclass of Vector that implements a standard last-in, first-out stack. LIFO
- Stack only defines the default constructor, which creates an empty stack.
- Stack includes all the methods defined by Vector and adds several of its own.
- Stack is declared as follows:
  - Stack<E> st = new Stack<E>();
- where E specifies the type of object.

# Stack - Methods

- Stack includes all the methods defined by Vector and adds several methods of its own.

| Sr. | Method & Description |
|---|---|
| 1 | boolean empty()<br>Returns true if the stack is empty, and returns false if the stack contains elements. |
| 2 | E peek()<br>Returns the element on the top of the stack, but does not remove it. |
| 3 | E pop()<br>Returns the element on the top of the stack, removing it in the process. |
| 4 | E push(E element)<br>Pushes element onto the stack. Element is also returned. |

# Queue

- Queue interface extends Collection and declares the behaviour of a queue, which is often a first-in, first-out list.
- LinkedList and PriorityQueue are the two classes which implements Queue interface
- Queue is declared as follows:
  - Queue<E> q = new LinkedList<E>();
  - Queue<E> q = new PriorityQueue<E>();
- where E specifies the type of object.

# Queue-Methods

| Sr. | Method & Description |
|---|---|
| 1 | E element()<br>Returns the element at the head of the queue. The element is not removed. It throws NoSuchElementException if the queue is empty. |
| 2 | boolean offer(E obj)<br>Attempts to add obj to the queue. Returns true if obj was added and false otherwise. |
| 3 | E peek()<br>Returns the element at the head of the queue. It returns null if the queue is empty. The element is not removed. |
| 4 | E poll()<br>Returns the element at the head of the queue, removing the element in the process. It returns null if the queue is empty. |
| 5 | E remove()<br>Returns the element at the head of the queue, returning the element in the process. It throws |

# Queue Example

```
1.  import java.util.*;
2.  public class QueueDemo {
3.      public static void main(String[] args) {
4.          Queue<String> q = new LinkedList<String>();
5.          q.add("Tom");
6.          q.add("Jerry");
7.          q.add("Mike");
8.          q.add("Steve");
9.          q.add("Harry");
10.         System.out.println("Elements in Queue:"+q);
11.         System.out.println("Removed element: "+q.remove());
12.         System.out.println("Head: "+q.element());
13.         System.out.println("poll(): "+q.poll());
14.         System.out.println("peek(): "+q.peek());
15.         System.out.println("Elements in Queue:"+q);
16.     }
```

```
G:\Darshan\Java 2019\PPTs\HAD\Programs>java QueueDemo
Elements in Queue:[Tom, Jerry, Mike, Steve, Harry]
Removed element: Tom
Head: Jerry
poll(): Jerry
peek(): Mike
Elements in Queue:[Mike, Steve, Harry]
```

# PriorityQueue

- PriorityQueue extends AbstractQueue and implements the Queue interface.
- It creates a queue that is prioritized based on the queue's comparator.
- PriorityQueue is declared as follows:
  - PriorityQueue<E> = new PriorityQueue<E>;
- It builds an empty queue with starting capacity as 11.

# PriorityQueue - Example

```
import java.util.*;
public class PriorityQueueExample {
    public static void main(String[] args) {
                    PriorityQueue<Integer>  numbers  =  new
PriorityQueue<>();
        numbers.add(750);
        numbers.add(500);
        numbers.add(900);
        numbers.add(100);
System.out.println(numbers);
        }
    }
}
```

# List v/s Sets

| List | Set |
|---|---|
| Lists allow duplicates. | Sets allow only unique elements. |
| List is an ordered collection. | Sets is an unordered collection. |
| Popular implementation of List interface includes ArrayList, Vector and LinkedList. | Popular implementation of Set interface includes HashSet, TreeSet and LinkedHashSet. |

**When to use List and Set?**

- Lists - If insertion order is maintained during insertion and allows duplicates.
- Sets – If unique collection without any duplicates without maintaining order.

# Maps

- A map is an object that stores associations between keys and values, or key/value pairs.
- Given a key, you can find its value. Both keys and values are objects.
- The keys must be unique, but the values may be duplicated. Some maps can accept a null key and null values, others cannot.
- Maps don't implement the Iterable interface. This means that you cannot cycle through a map using a for-each style for loop. Furthermore, you can’t obtain an iterator to a map.

# Map Interfaces

| Interface | Description |
|---|---|
| Map | Maps unique keys to values. |
| Map.Entry | Describes an element (a key/value pair) in a map. This is an inner class of Map. |
| NavigableMap | Extends SortedMap to handle the retrieval of entries based on closest-match searches. |
| SortedMap | Extends Map so that the keys are maintained in ascending order. |

# Map Classes

| Class | Description |
|---|---|
| AbstractMap | Implements most of the Map interface. |
| EnumMap | Extends AbstractMap for use with enum keys. |
| HashMap | Extends AbstractMap to use a hash table. |
| TreeMap | Extends AbstractMap to use a tree. |
| WeakHashMap | Extends AbstractMap to use a hash table with weak keys. |
| LinkedHashMap | Extends HashMap to allow insertion-order iterators. |

# HashMap Class

- The HashMap class extends AbstractMap and implements the Map interface.
- It uses a hash table to store the map.
- This allows the execution time of get() and put() to remain constant even for large sets.
- HashMap is a generic class that has declaration:
    - class HashMap<K,V>

# HashMap - Constructors

| Sr. | Constructor & Description |
|---|---|
| 1 | HashMap()<br>Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75). |
| 2 | HashMap(int initialCapacity)<br>Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75). |
| 3 | HashMap(int initialCapacity, float loadFactor)<br>Constructs an empty HashMap with the specified initial capacity and load factor. |
| 4 | HashMap(Map<? extends K,? extends V> m)<br>Constructs a new HashMap with the same mappings as the specified Map. |

# Example

```
import java.util.*;
class HashMapDemo {
   public static void main(String args[]) {
        // Create a hash map.
        HashMap<String, Double> hm = new HashMap<String, Double>();
        // Put elements to the map
        hm.put("John Doe", new Double(3434.34));
        hm.put("Tom Smith", new Double(123.22));
        hm.put("Jane Baker", new Double(1378.00));
        hm.put("Tod Hall", new Double(99.22));
        hm.put("Ralph Smith", new Double(-19.08));
        // Get a set of the entries.
        Set<Map.Entry<String, Double>> set = hm.entrySet();
        // Display the set.
```

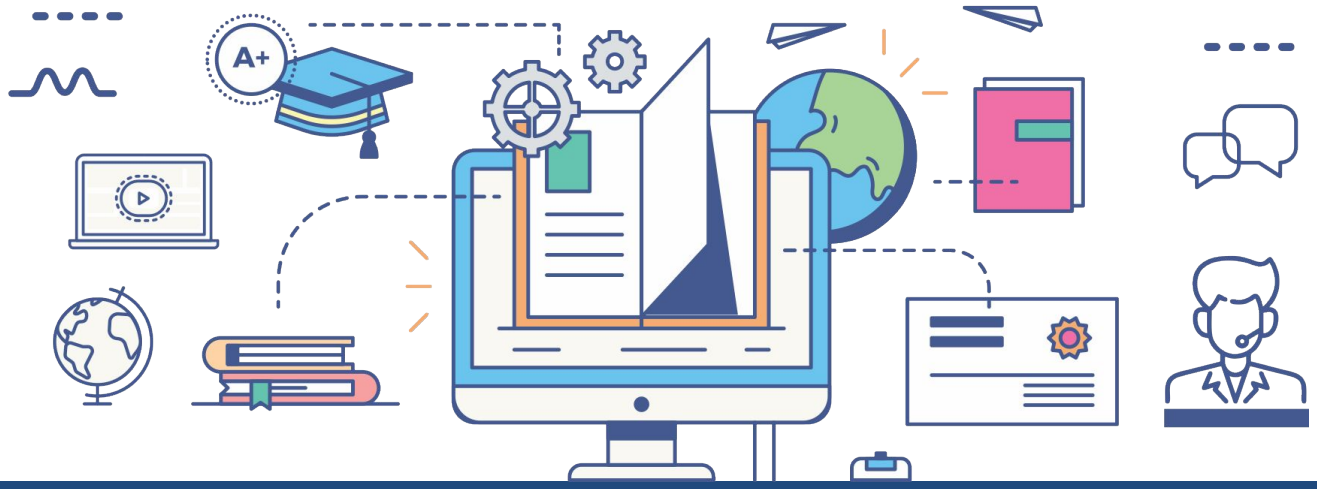
# Cont...

```
    for(Map.Entry<String, Double> me : set) {
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
    }
            System.out.println();
            //Deposit 1000 into John Doe's account.
            double balance = hm.get("John Doe");
            hm.put("John Doe", balance + 1000);
            System.out.println("John Doe's new balance: " +
            hm.get("John Doe"));
    }
}
```

```
C:\Users\hardi\Desktop>java HashMapDemo
Tod Hall: 99.22
John Doe: 3434.34
Ralph Smith: -19.08
Tom Smith: 123.22
Jane Baker: 1378.0

John Doe's new balance: 4434.34
```

# DIGITAL LEARNING CONTENT




Parul® University

www.paruluniversity.ac.in