

Collections Framework

Pradeep LN

Objectives

- Dynamic Collections vs Arrays
- What is Collections Framework?
- Collection Interface
- Set Interface
- List Interface
- Map Interface
- Generics
- Using Generics
- Collections API
- The Collection<E> and List<E> Interface
- The ArrayList<E> and LinkedList<E> Classes
- Looping over Collections: Iterable<E> Interface
- Collecting Primitive Values: Auto-Boxing

Objectives

- Using Wildcards with Generic Types
- Iterators and `Iterator<E>` Interface
- Maps and `Map<K,V>` Interface
- Sorted Collections
- The `SortedSet<E>` and `SortedMap<E>` Interface
- The Collections Class utility
- Algorithms
- Conversion utilities

What is a Collection?



- A collection (sometimes called a container) is an object that groups multiple elements into a single unit.

Why Collections?

- Many a times it is necessary to hold a group of objects in a single unit.
- Collections are used to store, retrieve and manipulate data, and to transmit data from one method to another.
- Collections typically represent data items that form a natural group, a mail folder, a telephone directory...

Dynamic Collections vs Arrays

- Arrays
 - The Array is the simplest way that Java provides to store and randomly access a sequence of objects.
 - The Array is a simple linear sequence, which makes quick insertion and element access fast.
 - The speed of accessing data comes with a drawback, the size of an array is fixed and cannot be changed for its lifetime.
 - The other drawback includes insertion of an element at a particular position is slow.

Dynamic Collections vs Arrays

- Collections
 - Unlike arrays the collection classes do not have a fixed size, the size is dynamically expanded.
 - Handle insertion, deletion and resizing better, even though few implementations will be backed by an array.
 - The drawbacks include, random access of elements is not always as fast when compared to an array.

Collections Framework

- Java's support for collection is provided by the Java's Collection Framework.
- The Java collections framework is made up of a set of interfaces and classes for working with groups of objects
- The Java Collections Framework provides
 - **Interfaces**

These are abstract data types representing collections.
 - **Implementations**

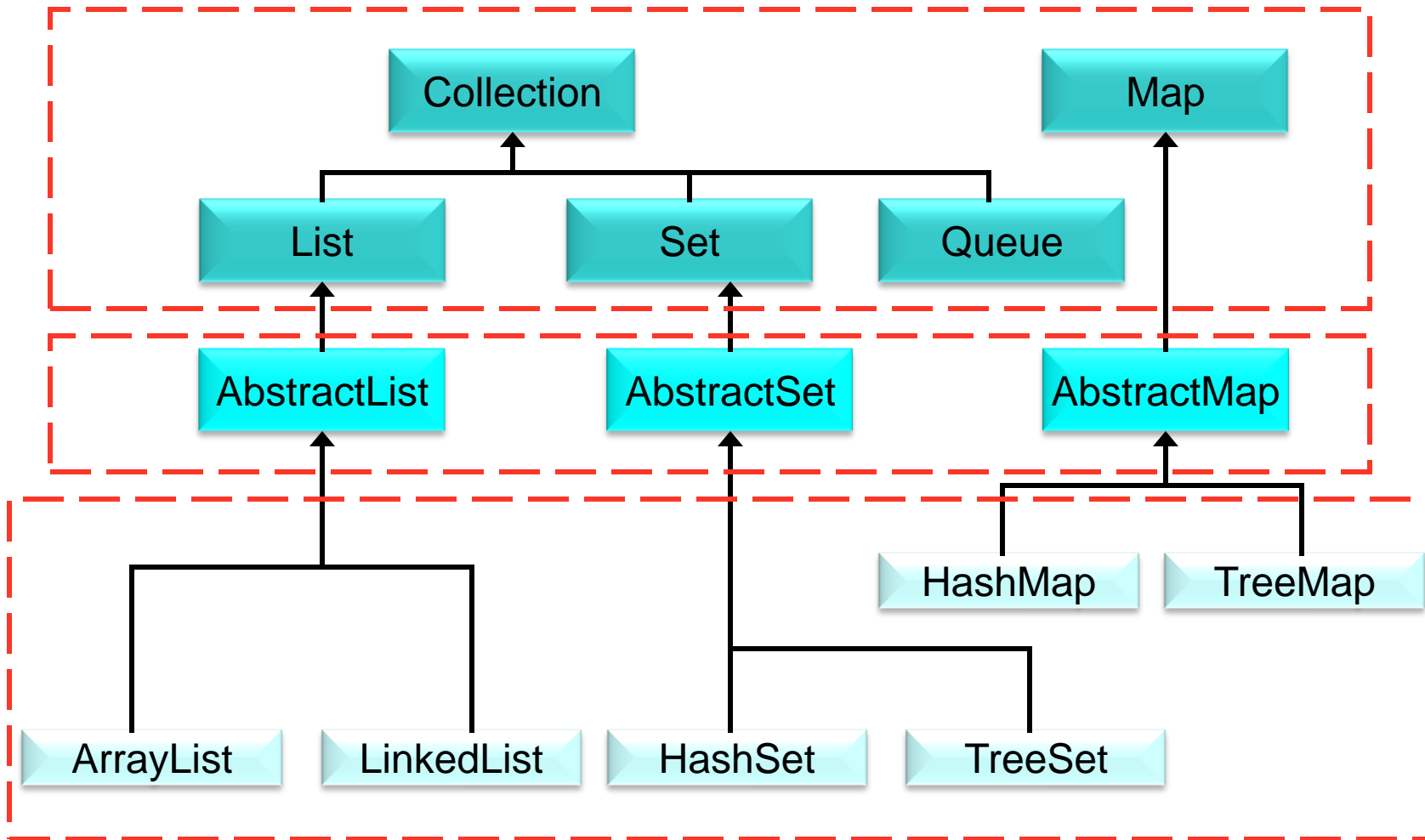
These are concrete implementations of the collection interfaces.
 - **Algorithms**

These are the methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces.

Benefits of Collection Framework

- **Reduces programming effort**
 - Provides useful data structures and algorithms.
 - Thereby, programmer concentrates on the business logic.
- **Increases program speed and quality:**
 - Provides high-performance, high-quality implementations of useful data structures and algorithms.
- **Allows interoperability among unrelated APIs:**
 - Facilitates interoperability among unrelated APIs.
 - Frees the programmer from writing adapter objects or conversion code to connect APIs.
- **Reduces effort to learn and to use new APIs:**
 - No need to learn each API from scratch
- **Reduces effort to design new APIs:**
 - Don't have to reinvent the wheel.
 - Instead, use the standard collection interfaces.
- **Fosters software reuse:**
 - New data structures that conform to the standard collection interfaces are by nature reusable.

Collections Framework



Collection Interface

- Collection Interface is the root interface in the collection hierarchy.
- It acts as an abstraction for different types of collections.
- It is foundation upon which the collections framework is built.
- The collection interface is typically used to pass collections around manipulate them where maximum generality is desired.
- Collection interface has no direct implementations, it has sub interfaces like Set and List.

Collection Interface

- public interface **Collection**
- **Some important Methods**
 - **int size()**
Returns the number of elements in this collection.
 - **boolean isEmpty()**
Returns true if the contains no elements.
 - **boolean contains(Object o)**
Returns true if the collection contains the specified element.
 - **Iterator iterator()**
Returns an iterator over the elements in this collection.

Collection Interface

- **boolean add(Object o)**
Ensures that this collection contains the specified element, returns true if changes made else returns false if duplicates are not allowed.
- **boolean remove(Object o)**
Removes a single instance of the specified element and returns true if the collection contained the element.
- **Object[] toArray()**
Returns an array containing all of the elements in this collection.
- **void clear()**
Removes all the elements from this collection.

List Interface

- List is an ordered collection (also called a sequence).
- The user of this interface has precise control over where in the list each element is inserted.
- The user can access elements by their integer index and search for elements in the list.
- Lists may contain duplicate elements.
- In addition to operations inherited from Collection, the List interface includes operations for Positional access , Search , Iteration ,Range View.

List Interface

- public interface **List** extends Collection

Methods in addition to Collection interface

- **void add(int index, Object element)**
Inserts the specified element at the specified position in this list.
- **Object get(int index)**
Returns the element at the specified position in the list.
- **Object set(int index , Object element)**
Replaces the element at the specified position in this list with the specified element.

List Interface

- **List subList(int fromIndex , int toIndex)**
Returns a view of the portion of this list between the specified
fromIndex, inclusive, and toIndex, exclusive.
- **int indexOf(Object o)**
Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain the element.
- **ListIterator listIterator()**
Returns a list iterator of the elements in the list.

The Set Interface

- A Set is a Collection that cannot contain duplicate elements.
- The Set interface contains *only* methods inherited from the Collection and adds the restriction that duplicate elements are not allowed.
- public interface **Set** extends Collection

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.
- The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings.

Map Interface

- public interface **Map**

Some important Methods

- **void clear()**
Removes all mappings from this map.
- **boolean containsKey(Object key)**
Returns true if this map contains a mapping for the specified key.
- **boolean containsValue(Object value)**
Returns true if this map maps one or more keys to the specified value.
- **Object get(Object key)**
Returns the value to which this map maps the specified key.
- **boolean isEmpty()**
Returns true if this map contains no key-value mappings.

Map Interface

- **Set keySet()**
Returns a set view of the keys contained in this map.
- **Object put(Object key , Object value)**
Associates the specified value with the specified key in this map.
- **Object remove(Object key)**
Removes the mapping for this key from this map if it is present.
- **int size()**
Returns the number of key-value mappings in this map.
- **Collection values()**
Returns a collection view of the values contained in this map.

class ArrayList

- public class **ArrayList** implements List
- ArrayList is the resizable array implementation of the List interface.
- In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list.
- Each ArrayList instance has a capacity, the capacity is the size of the array used to store the elements in the list. As the elements are added to an ArrayList the capacity increases automatically.

class ArrayList

CONSTRUCTORS

- **ArrayList()**
Constructs an empty list with an initial capacity of ten.
- **ArrayList(int initialCapacity)**
Constructs an empty list with specified initial capacity.
- **ArrayList(Collection c)**
Constructs a list containing the elements of the specified collection.

class ArrayList

Some Useful Methods

- **boolean add(Object o)**
Appends the specified element to the end of this list.
- **void add(int index , Object element)**
Inserts the specified element at the specified position in this list.
- **void clear()**
Removes all of the elements from this list.
- **Object clone()**
Returns a shallow copy of this ArrayList instance.
- **boolean contains(Object element)**
Returns true if this list contains the specified element.

class ArrayList

- **void ensureCapacity(int minCapacity)**
Increases the capacity to hold the number of elements specified.
- **Object get(int index)**
Returns the element at the specified position in this list.
- **boolean isEmpty()**
Tests if this list has no elements.
- **Object remove(int index)**
Removes the element at the specified position in this list.
- **Object set(int index , Object element)**
Replaces the element at the specified position with the specified element.

Example ArrayList

```
import java.util.*;
class ArrayListDemo1 {
    public static void main(String[] args)
    {
        ArrayList list = new ArrayList();
        list.add("J2SE");
        list.add("J2ME");
        list.add("J2EE");
        System.out.println("Size of the arraylist : " +list.size());
        for(int i=0 ; i<list.size() ; ++i) {
            String str = (String) list.get(i);
            System.out.println(str);
        }
        System.out.println("Adding one more element to the arraylist");
        list.add("The Java Platform");
        System.out.println("Element 4 : " + list.get(3));
        System.out.println("Size of arraylist now : " +list.size());
    }
}
```

Example ArrayList

```
import java.util.*;
public class ArrayListDemo2 {
    public static void main(String args[ ]) {
        ArrayList al = new ArrayList ();
        System.out.println("Initial size of al: " + al.size());
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        al.add(1, "A2");
        System.out.println("Size after additions:"+al.size());
        System.out.println("Contents of al: " + al);
        al.remove("F");
        al.remove(2);
        System.out.println("Size after deletions: "+al.size());
        System.out.println("Contents of al: " + al);
    }
}
```

Generics

A typical program using collections in pre Java 1.5

```
List myList = new ArrayList();  
myList.add(new Integer(0));  
myList.add(new Integer(1));  
myList.add(new Integer(2));  
Integer i1 = (Integer) myList.get(1);  
Integer i2 = (Integer) myList.get(2);  
String str = (String) myList.get(3);
```



Casting
required

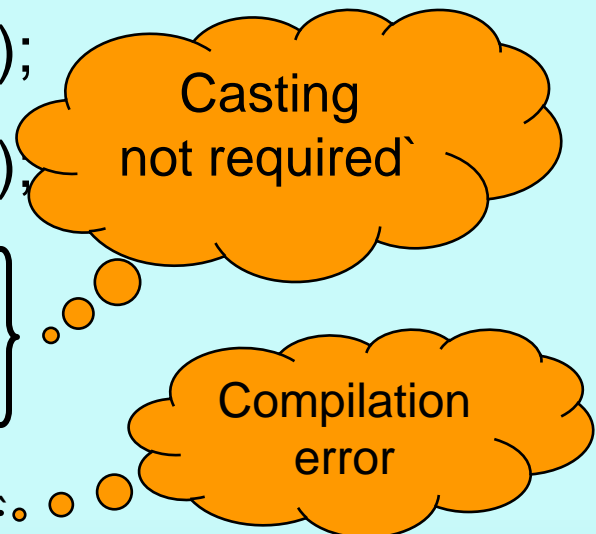
Generics

- Issues with the last program
 - Since a collection was looked at as an heterogeneous container an object of any type can be added.
 - However it is observed that homogeneous collections are more common place than heterogeneous collections.
 - In the example, the `ArrayList` is collection of `Integer` objects, in spite of this an explicit cast is essential when returning objects from the `ArrayList`.
 - In the last line of the example, an `Integer` object returned from the `ArrayList` is cast to `String`. This would not be an compile time error but definitely a run time error. So collections are not type safe.

Generics

Using Generics

```
List<Integer> myList = new ArrayList<Integer>();  
myList.add(new Integer(0));  
myList.add(new Integer(1));  
myList.add(new Integer(2));  
Integer i1 = myList.get(1);  
Integer i2 = myList.get(2);  
String str = myList.get(3);
```



Casting not required

Compilation error

See listing: **GenericsDemo.java**

Generics

- Why Generics?
 - Type safety : In the example the ArrayList is declared as a collection of Integers any attempt to add an object other than the Integer would give a compilation error.
 - No casting required: While retrieving objects from the collection explicit casting is not required and the type checking of the returned object and its reference type is done at compile time.
 - The net effect of this especially in a large program is improved readability and robustness.

Using Generics

- Excerpt from the definition of the List interface

```
public interface List<E> {  
    void add(E x);  
    E get(int index);  
    Iterator<E> iterator();  
}
```

The declarations within the angle brackets **<E>** are the formal type parameters of the List interface.

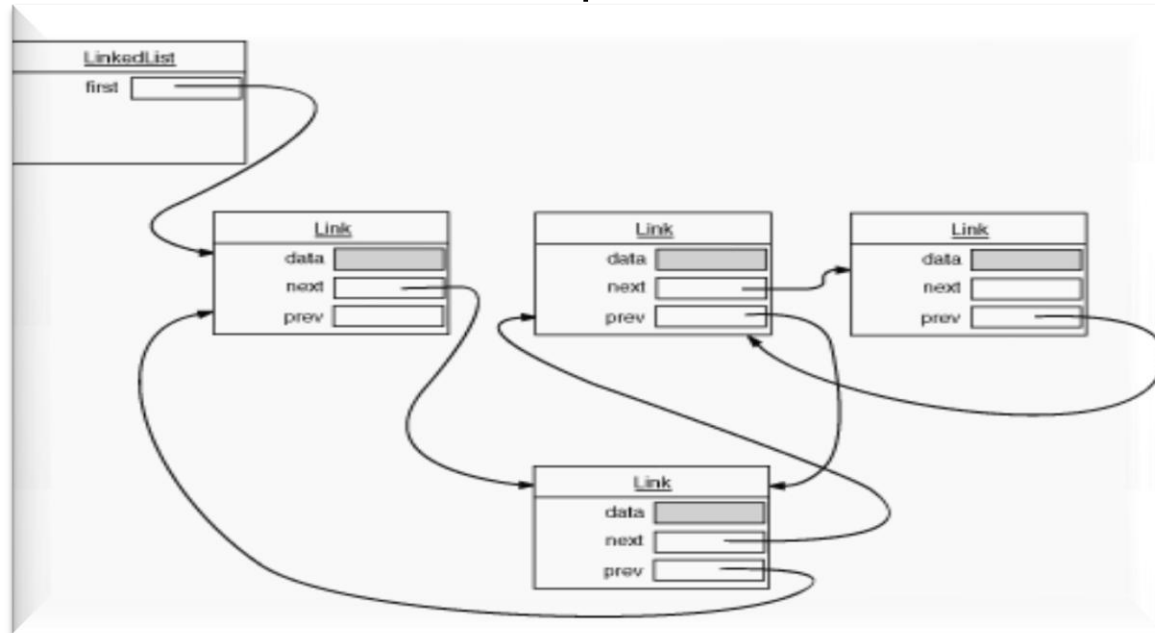
In the invocation :

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

all occurrences of the formal type parameter (E) is replaced by actual type parameter (Integer).

class LinkedList

- public class **LinkedList<E>** implements List<E>



- LinkedList is the linked list implementation of the List interface.
- In addition to implementing the List interface, the LinkedList class provides uniformly named methods to get, remove and insert an element at the beginning and end of the list.
- LinkedList also implements the Queue interface, providing first-in-first out queue operations.

class LinkedList

CONSTRUCTORS

- **LinkedList()**
Constructs an empty list.
- **LinkedList(Collection c)**
Constructs a list containing the elements of the specified collection, in the order returned by the collection's iterator.

class LinkedList

Some Useful Methods

- **void addFirst(E o)**
Inserts the given element at the beginning of the list.
- **void addLast(E o)**
Appends the given element to the end of the list.
- **E getFirst()**
Returns the first element in this list.
- **E getLast()**
Returns the last element in this list.
- **E peek()**
Retrieves, but does not remove first element of the list.

class LinkedList

- **E poll()**
Retrieves and removes first element of the list.
- **E remove(int index)**
Removes the element at the specified position in this list.
- **E removeFirst() & E removeLast()**
Removes and returns the first and last elements.
- **E set(int index , E element)**
Replaces the element at the specified position with the specified element.

Example LinkedList

```
import java.util.*;
public class LinkedListDemo
{
    public static void main(String args[ ]) {
        LinkedList<String> list = new LinkedList<String>();
        list.add("Boolean");
        list.add("FileInputStream");
        list.addLast("System");
        list.addFirst("ArrayList");
        System.out.println ("Original contents of list: " + list);
        list.add(1, "Array");
        System.out.println("Contents of list: " + list);
        list.remove("FileInputStream");
        list.remove(2);
        System.out.println ("Contents after deletion: " + list);
        list.removeFirst();
        list.removeLast();
        System.out.println ("After deleting first & last:" + list);
    }
}
```

Looping over Collections

- public interface **Iterator<E>**
- An Iterator is a light weight object whose job is to move through a sequence of objects and select each object in that sequence.
- The Iterator can traverse through the collection without the knowledge of the underlying structure.
- By calling the method `iterator()`, the container returns an Iterator.

Iterator Interface

Some Useful Methods

- **boolean hasNext()**
Returns true if the iteration has more elements.
- **E next()**
Returns the next element in the iteration.
- **void remove()**
Removes from the underlying collection the last element returned by the iterator.

Example Iterator

```
import java.util.*;
public class IteratorDemo{
    public static void main(String args[]){
        List<String> list = new ArrayList<String>();
        list.add("Collection");
        list.add("List");
        list.add("ArrayList");
        list.add ("LinkedList");
        System.out.println("Contents of the List");
        for(Iterator<String> I = list.iterator() ; i.hasNext() ; ){
            System.out.println(i.next());
        }
    }
}
```


for each loop

- Classical way of iterating through a collection:

```
List<Integer> list = new ArrayList<Integer>();  
list.add(1);  
list.add(2);.....  
for (Iterator<Integer> i = list.iterator() ; i.hasNext(); )  
{  
    System.out.println(i.next());  
}
```

In this example, we retrieve an iterator of the collection and iterate through the collection using the for loop.

for each loop

- Iterating the collection using the for-each loop:

```
List<Integer> list = new ArrayList<Integer>();  
list.add(1);  
list.add(2);.....  
for (Integer i : list)  
{  
    System.out.println(i);  
}
```

The for-each loop does not add any functionality, it's a convenient way of iterating through collections.

for each loop

- The for-each loop allows you to automatically iterate through the elements of an array or collection.
- Instead of manually advancing a looping variable, it is managed by the for-each construct.
- The signature of the for-each loop is:

**for(declaration :expression)
statement**

- The expression must be either an array or an object that implements the `java.lang.Iterable` interface.
- The type of array or `Iterable` elements must be assignment compatible with the type of variable declared in declaration.

Example for-each

```
import java.util.*;
public class ForEachDemo{
    public static void main(String args[]){
        List<String> list = new ArrayList<String>();
        list.add("Collection");
        list.add("List");
        list.add("ArrayList");
        list.add("LinkedList");
        System.out.println("Contents of the List");
        for(String s : list ){
            System.out.println(s);
        }
    }
}
```

Exercise

- See Exercise1.doc

Generics and Subtyping

```
List<Integer> intList = new ArrayList<Integer>();  
List<Object> objList = intList;  
objList.add("Hello World!");  
Integer I = intList.get(0);
```

- In the above code snippet we are trying to cast a List of String to a List of Objects.
- Even though String is a subtype of Object, the casting cannot be done based on the parameter type. Line 2 would throw a compilation error.
- Here List<String> is not a subtype of List<Object> so the compilation error.

Wildcards

```
void printCollection(List<Object> list){  
    for(Object o : list){  
        System.out.println(o);  
    }  
}
```

As demonstrated earlier `List<Object>` is not a super type of all lists.

Hence in the above code snippet the method `printCollection()` takes only `List<Object>` as a parameter.

So what is the super type of all kinds of lists?

Wildcards

- The last code snippet can be rewritten as

```
void printCollection(List<?> list){  
    for(Object o : list){  
        System.out.println(o);  
    }  
}
```

Here the List is of unknown type so its elements type can be anything.

The List in the above method is called **wildcard type**.

We can now call this method with a list of any type and read elements from the list.

We cannot add objects to the list since the add() method takes a argument of type E, the element type of the list, but the actual type parameter is ? Which stands for unknown type.

Wildcards

```
void printNumbers(List<? extends Number > list){  
    for(Object o : list){  
        System.out.println(o);  
    }  
}
```

- The printNumbers() method accepts a List which is a subtype of class Number.
- The formal type parameters of the List is **<? extends Number>** which means the argument type can be a Number or its subtype only.

Example

```
import java.util.*;

class WildcardDemo {
    public static void printNumbers(List<? extends Number> list)
    {
        for(Object o : list)
            System.out.println(o);
    }
    public static void main(String[] args)
    {
        List<Integer> list = new ArrayList<Integer>();
        list.add(0);
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        printNumbers(list);
    }
}
```

class HashSet

- public class **HashSet<E>** extends AbstractSet<E>
- The HashSet implements the Set interface, backed by a HashMap instance.
- A HashSet does not hold more than one instance of each object value.

CONSTRUCTORS

- **HashSet()**
- **HashSet(Collection c)**
Constructs a new set containing the elements in the specified collection.
- **HashSet(int initialCapacity)**
Constructs a new set, the backing HashMap instance has the specified initial capacity.

class HashSet

Some Useful Methods

- **boolean add(E o)**
Adds the specified element to this set if it is not already present.
- **void clear()**
Removes all of the elements from this list.
- **Object clone()**
Returns a shallow copy of this HashSet instance, the elements themselves are not cloned.
- **boolean contains(Object o)**
Returns true if this set contains the specified element.

class HashSet

- **boolean isEmpty()**
Returns true if this set contains no elements.
- **Iterator<E> iterator()**
Returns an iterator over the elements in this set.
- **boolean remove(Object o)**
Removes the specified element from this set if it is present.
- **int size()**
Returns the number of elements in the set.

Example

```
import java.util.*;

class HashSetDemo
{
    public static void main(String[] args)
    {
        Set<String> set = new HashSet<String>();
        for(String str : args)
        {
            if(!set.add(str))
                System.out.println("Duplicate detected :"+str);
        }
        System.out.println(set.size()+" distinct words :"+ set);
    }
}
```

SortedSet Interface

- A **SortedSet** is a Set that maintains its elements in ascending order.
- The elements are sorted according to the elements natural ordering or according to a Comparator provided at the SortedSet creation time.
- All elements inserted into a SortedSet must implement the Comparable interface or be accepted by the specified Comparator.
- In addition to the normal Set operations, the SortedSet interface provides operations for the following:
 - Range view – allows arbitrary range operations.
 - Endpoints – returns the first or last element.
 - Comparator access – returns the Comparator, if any.

SortedSet Interface

public interface **SortedSet<E>**

Some important Methods

- **Comparator<? Super E> comparator()**
Returns the comparator associated with this sorted set, or null if it uses its elements natural ordering.
- **E first()**
Returns the first(lowest) element currently in the set.
- **SortedSet<E> headSet(E toElement)**
Returns a view of the portion of this sorted set whose elements are strictly less than toElement.
- **E last()**
Returns the last(highest) element currently in the set.

SortedSet Interface

- **SortedSet<E> subSet(E fromElement , E toElement)**

Returns a view of the position of this sorted set whose elements range from fromElement, inclusive, to toElement, exclusive.

- **SortedSet<E> tailSet(E fromElement)**

Returns a view of the portion of this sorted set whose elements are greater than or equal to fromElement.

Example

```
import java.util.*;
class Employee implements Comparable{
    private int age;
    public Employee(int age) { this.age = age; }
    public int getAge() { return age;}
    public int compareTo(Employee emp) {
        Integer i1 = this.age;
        Integer i2 = emp.getAge();
        return i1.compareTo(i2);
    }
}
class MyComparator implements Comparator<Employee> {
    public int compare(Employee e1 , Employee e2) {
        Integer i1 = e1.getAge();
        Integer i2 = e2.getAge();
        return i1.compareTo(i2);
    }
}
```

Example - contd

```
public class TreeSetDemo
{
    public static void main(String[] args)
    {
        MyComparator comp = new MyComparator();
        TreeSet<Employee> set = new TreeSet<Employee>(comp);
        set.add(new Employee(41));
        set.add(new Employee (23));
        set.add(new Employee (31));
        set.add(new Employee (49));
        set.add(new Employee (51));
        for(Employee emp : set)
            System.out.println(emp);
    }
}
```

Exercise

- See Exercise2.doc

class HashMap

- public class **HashMap<K,V>** extends AbstractMap<K,V>
- The HashMap is the hash table based implementation of the Map interface.
- The implementation provides all of the optional map operations, and permits null values and the null key.

CONSTRUCTORS

- **HashMap()**

Constructs an empty HashMap with default initial capacity of 16.

- **HashMap(Map<K,V> m)**

Constructs an empty HashMap with the same mappings as the specified Map.

- **HashMap(int initialCapacity)**

Constructs an empty HashMap with the specified initial capacity.

class HashMap

Some Useful Methods

- **void clear()**
Removes all mappings from this map.
- **Object clone()**
Returns a shallow copy of this HashMap instance, the keys and values themselves are not cloned.
- **boolean containsKey(Object o)**
Returns true if this map contains a mapping for the specified key.
- **boolean containsValue(Object o)**
Returns true if this map maps one or more keys to the specified value.

class HashMap

- **boolean isEmpty()**
Returns true if this map contains no key-value mappings.
- **V get(Object key)**
Returns the value to which the specified key is mapped.
- **V put(K key , V value)**
Associates the specified value with specified key in this map.
- **boolean remove(Object o)**
Removes the specified element from this set if it is present.
- **int size()**
Returns the number of elements in the set.

Example

```
import java.util.*;

class HashMapDemo
{
    public static void main(String[] args)
    {
        Map<String,Integer> map =
                                new HashMap<String,Integer>();
        for(String str : args)
        {
            Integer i = str.length();
            map.put(str , i);
        }
        System.out.println(map.size()+" distinct words");
        System.out.println(map);
    }
}
```


Collections Class

- Collections is a utility class part of the Java Collections Framework.
- This class consists exclusively of static methods that operate on or return collections.
- All methods in this class throw `NullPointerException` if the collections or class objects provided to them are null.
- `public class Collections`

Collections Class

Some Useful Methods

- **static int binarySearch(Object[] o, Object a)**
Searches the specified array for the specified value, this method is overloaded to work with different data types.
- **static boolean equals(Object[] o, Object a)**
Returns true if two specified arrays equal to one another, this method is overloaded to work with different data types.
- **static void fill(Object[] a , Object val)**
Assigns the specified Object reference to each element of the specified array of objects and is overloaded for different data types.
- **static void sort(Object[] a)**
Sorts the specified array of objects into ascending order, according to natural ordering of its elements and is overloaded for different data types.

Example Collections

```
import java.util.*;
class CollectionsDemo {
    public static void main(String[] args) {
        ArrayList<Double> list = new ArrayList<Double>();
        list.add(4.56);
        list.add(3.22);
        list.add(14.77);
        list.add(14.79);
        list.add(20.0);
        list.add(8.55);
        for (double d : list)
            System.out.println(d);
        Collections.sort(list);
        System.out.println("Sorted list of double values");
    }
}
```

Example contd...

```
for (double d : list)
    System.out.println(d);
int pos = Collections.binarySearch(list,20);
System.out.println("20 is found at position " +
    pos + " in the list");
ArrayList<Double> a = new ArrayList<Double>();
for(int i=0;i<6;i++)
    a.add(0);
Collections.copy(a,list);
Collections.reverse(a);
System.out.println("Elements of copied ArrayList
    'a' in reverse order ");
for (double d : a)
    System.out.println(d);
}
}
```

Arrays Class

- Arrays is a utility class part of the Java Collections Framework.
- This class has various methods for manipulating arrays such as sorting and searching.
- All methods in this class throw `NullPointerException` if the specified array reference is null.
- `public class Arrays`

Arrays Class

Some Useful Methods

- **static int binarySearch(Object[] o, Object a)**
Searches the specified array for the specified value, this method is overloaded to work with different data types.
- **static boolean equals(Object[] o, Object a)**
Returns true if two specified arrays equal to one another, this method is overloaded to work with different data types.
- **static void fill(Object[] a , Object val)**
Assigns the specified Object reference to each element of the specified array of objects and is overloaded for different data types.
- **static void sort(Object[] a)**
Sorts the specified array of objects into ascending order, according to natural ordering of its elements and is overloaded for different data types.

Example Arrays

```
import java.util.*;
class ArraysTest {
    public static void main(String[] args) {
        int[ ] intArray = new int[5];
        int[ ] intArray2 = new int[5];
        Double[ ] wrapperArray = new Double[5];
        Double dbl = new Double(7.65);
        java.util.Arrays.fill(intArray,6);
        java.util.Arrays.fill(wrapperArray,dbl);
        for(int i=0 ; i<intArray2.length ; i++)
            intArray2[i] = new Random().nextInt(20);
        for(int i=0;i<intArray.length;i++)
            System.out.println(intArray[i]);
        for(int j=0;j<wrapperArray.length;j++)
            System.out.println(wrapperArray[j]);
    }
}
```

Example contd...

```
for(int k=0;k<intArray2.length;k++)
    System.out.println(intArray2[k]);
Arrays.sort(intArray2);
System.out.println("Sorted intArray2");
for(int k=0;k<intArray2.length;k++)
    System.out.println(intArray2[k]);
int pos = Arrays.binarySearch(intArray2,12);
if(pos > 0)
    System.out.println("Number 12 appears at
                        location " + pos);
else
    System.out.println("Number 12 does not
                        appear in the array");
}
}
```


Question time



Please try to limit the questions to the topics discussed during the session.

Participants are encouraged to discuss other issues during the breaks.

Thank you.