

A large, faint, light blue watermark of the Java logo is centered in the background of the slide. It consists of three concentric, stylized circular shapes that form a 'J' and a 'C' intertwined.

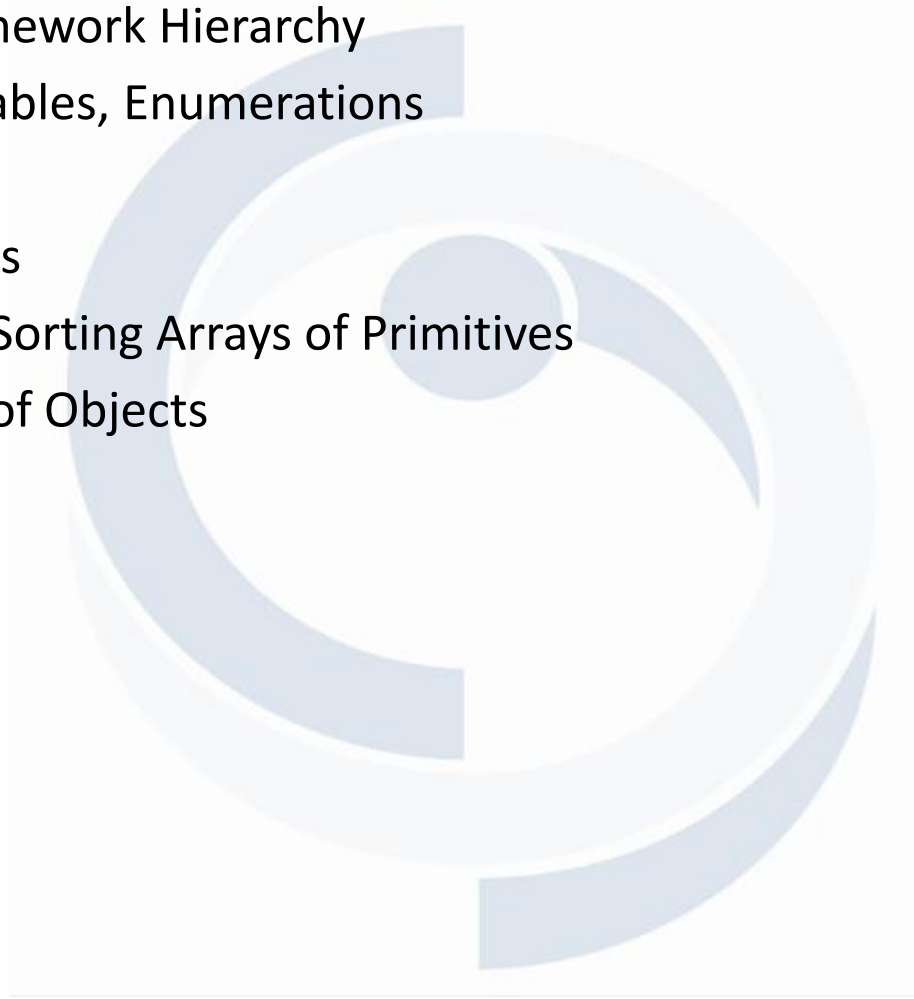
Core Java Training

Java Collections Framework



Collection API

- Collection interface
- Collection Framework Hierarchy
- Vectors, Hashtables, Enumerations
- Lists, Maps
- The Arrays Class
- Searching and Sorting Arrays of Primitives
- Sorting Arrays of Objects
- Iterators



Objectives

After completion of this module, you should be able to:

- Explain the limitations of array
- Understand Java framework hierarchy
- Interpret how to use iterator interface to traverse a collection
- Set interface, HashSet, and TreeSet
- List interface, ArrayList, and LinkedList
- Explain vector and stack
- Define map, hashmap and treemap
- Differentiate between collections and array classes



Limitations of array

- Arrays do not grow while applications demand otherwise
- Inadequate support for
 - inserting,
 - deleting,
 - sorting, and
 - searching operations



Collections

- A *collection* -- is an object that groups multiple elements into a single unit
- Collections are used to store, retrieve, manipulate, and communicate aggregate data
- Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers)
- A collection is an object that contains a group of objects within it. These objects are called the elements of the collection
- The elements of a collection are objects of same class
- Collections can grow to any size unlike arrays
- Java Collection Framework supports two types of collections
 - Collections
 - maps



Collection framework

Collections framework is a unified architecture for representing & manipulating collections

All collections frameworks contain the following:

- **Interfaces:**

Abstract data types that represent collections

Interfaces allow collections to be manipulated independently of the details of their representation and generally form a hierarchy

- **Implementations:**

These are the concrete implementations of the collection interfaces

They are like reusable data structures

- **Algorithms:**

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

The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface

Algorithms are reusable functionality

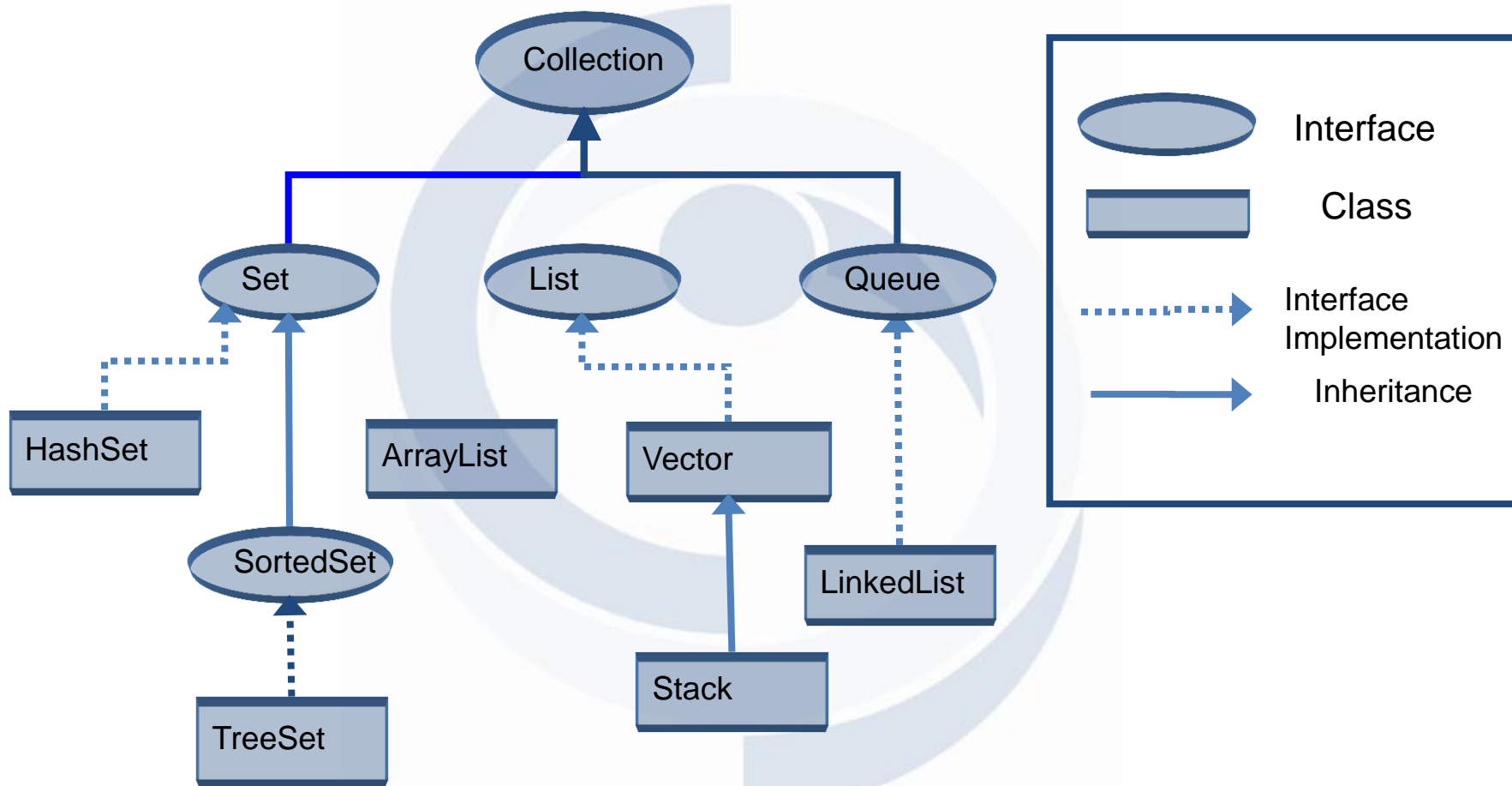


Collection framework - continued

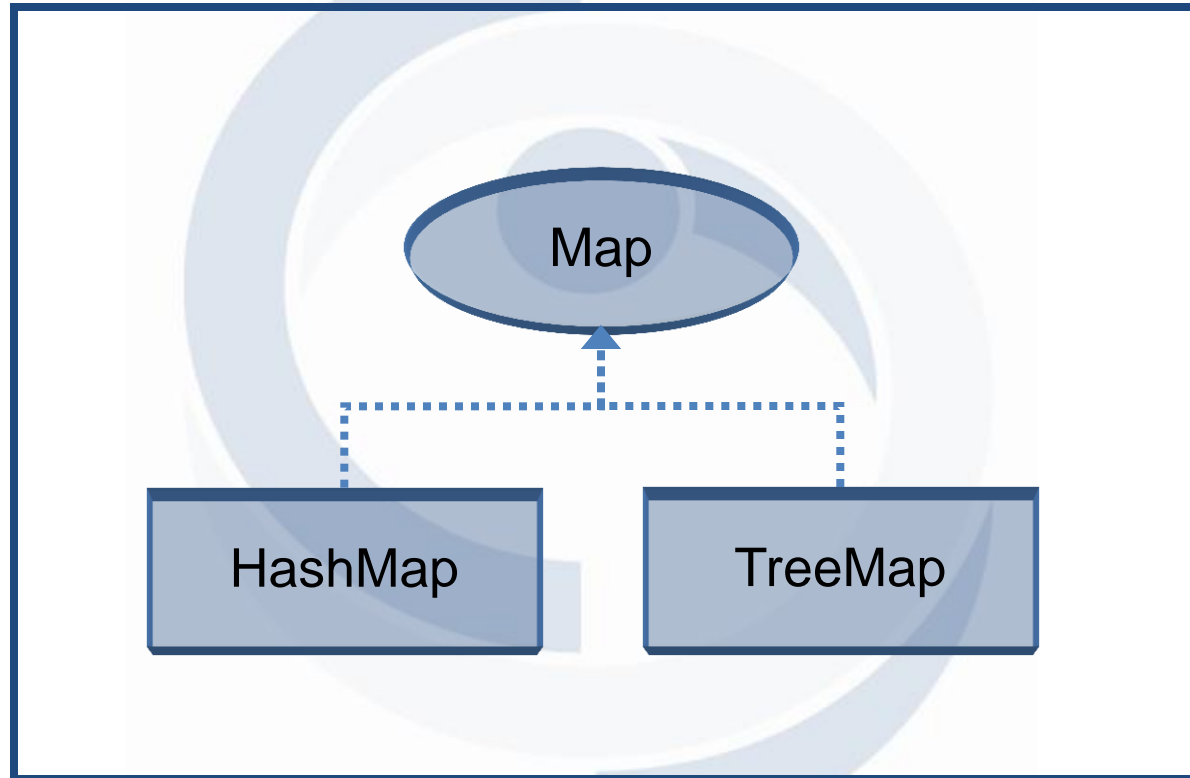
- Interfaces that characterize common collection types
- Abstract Classes which can be used as a starting point for custom collections and which are extended by the JDK implementation classes.
- Classes which provide implementations of the Interfaces.
- Algorithms that provide behaviors commonly required when using collections i.e. search, sort, iterate, etc.
- Another item in collections framework is the Iterator interface.
- An iterator gives you a general-purpose, standardized way of accessing the elements within a collection, one at a time.
- Apart from the Java Collections Framework, the best-known examples of collections frameworks are the C++ Standard Template Library (STL) and Smalltalk's collection hierarchy.



Collection framework hierarchy



Collection framework hierarchy - continued



Collection interface

- The Collection interface is the root interface for
 - storing a collection of objects, and
 - processing a collection of objects
- Present in java.util package



Collection interface methods

```
+add(element: Object): boolean  
+addAll(collection: Collection): boolean  
+clear(): void  
+contains(element: Object): boolean  
+containsAll(collection: Collection): boolean  
+equals(object: Object): boolean  
+hashCode(): int  
+isEmpty(): boolean  
+iterator(): Iterator  
+remove(element: Object): boolean  
+removeAll(collection: Collection): boolean  
+retainAll(collection: Collection): boolean  
+size(): int  
+toArray(): Object[]  
+toArray(array: Object[]): Object[]
```



Collection interface mostly used methods

- **boolean add (Object o)**
Ensures that this collection contains the specified element (optional operation).
- **boolean addAll(Collection c)**
Adds all of the elements in the specified collection to this collection (optional operation).
- **void clear()**
Removes all of the elements from this collection (optional operation).
- **boolean contains(Object o)**
Returns true if this collection contains the specified element.
- **Boolean containsAll(Collection c)**
Returns true if this collection contains all of the elements in the specified collection.
- **boolean equals(Object o)**
Compares the specified object with this collection for equality.



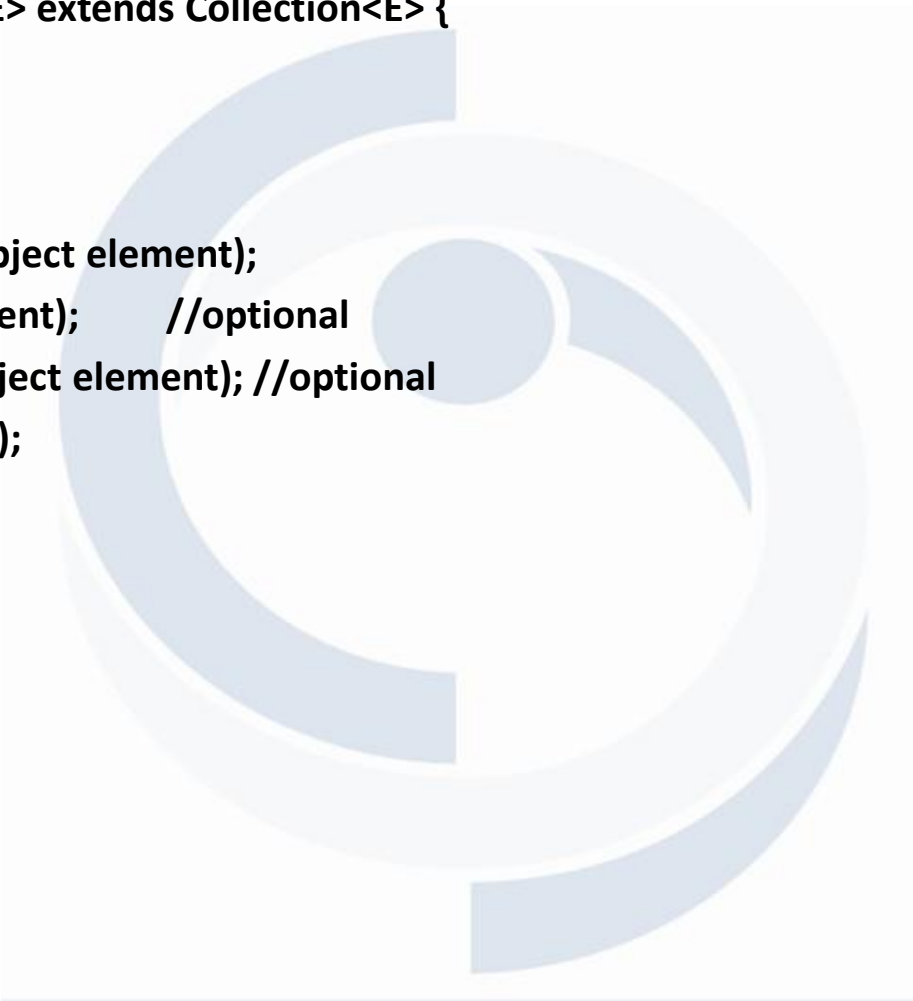
Set interface

- Mathematically a set is a collection of non-duplicate elements
- The Set interface is used to represent a collection which does not contain duplicate elements
- The Set interface extends the Collection interface.
- No new methods or constants
- Stipulates that an instance of Set contains no duplicate elements.
- The classes that implement Set must ensure that no duplicate elements can be added to the set.
- That is no two elements `e1` and `e2` can be in the set such that `e1.equals(e2)` is true.



Set interface - continued

```
public interface Set<E> extends Collection<E> {  
    // Basic operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);    //optional  
    boolean remove(Object element); //optional  
    Iterator<E> iterator();  
}
```



Set interface - continued

// Bulk operations

boolean containsAll(Collection<?> c);

boolean addAll(Collection<? extends E> c); //optional

boolean removeAll(Collection<?> c); //optional

boolean retainAll(Collection<?> c); //optional

void clear(); //optional

// Array Operations

Object[] toArray();

<T> T[] toArray(T[] a);

}



Set interface - continued

- The Java platform contains three general-purpose Set implementations: HashSet, LinkedHashSet and TreeSet.
- **HashSet**: This is an unsorted, unordered Set. This may be chosen when order of the elements are not important .
- **LinkedHashSet**: This is ordered. The elements are linked to one another (Double-Linked). This will maintain the list in the order in which they were inserted.
- **TreeSet**: This is a sorted set. The elements in this will be in ascending order in the natural order. You can also define a custom order by means of a Comparator passed as a parameter to the constructor.



Example: HashSet, LinkedHashSet, TreeSet

```
import java.util.*;

public class SetInterfaceEx{
    public static void main(String[] args) {
        int a[] = {5,2,9,4,1};

        Set <Integer> hs = new HashSet<Integer>();
        for(int i=0;i<a.length;i++) hs.add(new Integer(a[i]));
        System.out.println(hs.size() + " The HashSet is " + hs);

        Set <Integer> lhs = new LinkedHashSet<Integer>();
        for(int i=0;i<a.length;i++) lhs.add(new Integer(a[i]));
        System.out.println(hs.size() + " The LinkedHashSet is " + lhs);

        Set <Integer> ts = new TreeSet<Integer>();
        for(int i=0;i<a.length;i++) ts.add(new Integer(a[i]));
        System.out.println(hs.size() + " The TreeSet is " + ts);
    }
}
```



What to choose and when

- HashSet is a good choice for representing sets if you don't care about element ordering. But if ordering is important, then LinkedHashMap or TreeSet are better choices. However, LinkedHashMap or TreeSet come with an additional speed and space cost.
- Iteration over a LinkedHashMap is generally faster than iteration over a HashSet.
- Tree-based data structures get slower as the number of elements get larger
- HashSet and LinkedHashMap do not represent their elements in sorted order
- Because TreeSet keeps its elements sorted, it can offer other features such as the first and last methods, i.e. the lowest and highest elements in a set, respectively.



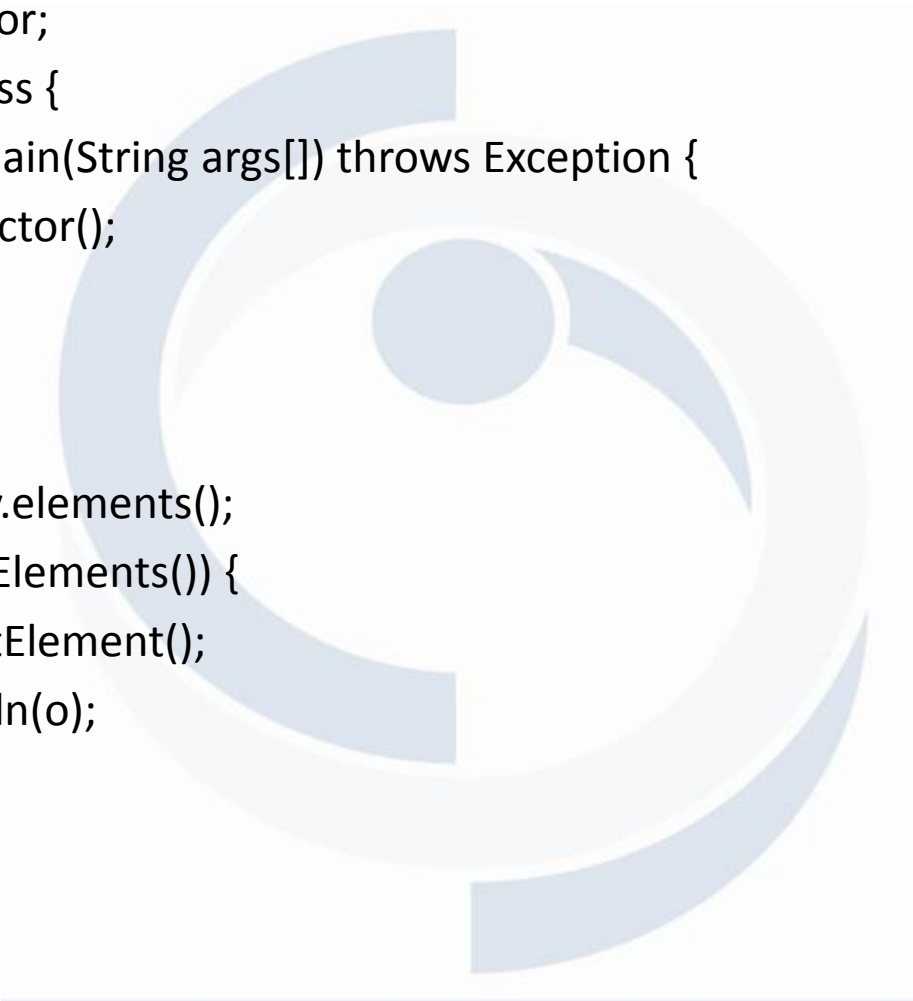
Enumeration interface

- The Enumeration interface defines a way to traverse all the members of a collection of objects.
- The `hasMoreElements()` method checks to see if there are more elements and returns a boolean.
- If there are more elements, `nextElement()` will return the next element as an `Object`.
- If there are no more elements when `nextElement()` is called, the runtime `NoSuchElementException` will be thrown.



Enumeration interface exmple

```
import java.util.Enumeration;
import java.util.Vector;
public class MainClass {
    public static void main(String args[]) throws Exception {
        Vector v = new Vector();
        v.add("a");
        v.add("b");
        v.add("c");
        Enumeration e = v.elements();
        while (e.hasMoreElements()) {
            Object o = e.nextElement();
            System.out.println(o);
        }
    }
}
```



Iterator interface


- Iterator is a special object to provide a way to access the elements of a collection sequentially.
- Iterator implements one of two interfaces: Iterator or ListIterator
- An Iterator is similar to the Enumeration interface, Iterators differ from enumerations in two ways:
 - Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
 - Method names have been improved.
 - `boolean hasNext()`
Returns true if the iteration has more elements.
 - `Object next()`
Returns the next element in the iteration.
 - `void remove()`
Removes from the underlying collection the last element returned by the iterator (optional operation).



Iterator interface example

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class MainClass {
    public static void main(String[] a) {
        Collection c = new ArrayList();
        c.add("1");
        c.add("2");
        c.add("3");
        Iterator i = c.iterator();
        while (i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```



The SortedSet interface and the TreeSet class

- SortedSet is a subinterface of Set, which guarantees that the elements in the set are sorted.
- TreeSet is a concrete class that implements the SortedSet interface. You can use an iterator to traverse the elements in the sorted order.
- The elements can be sorted in two ways.
 - One way is to use the Comparable interface. Objects can be compared using by the compareTo() method. This approach is referred to as a natural order.
 - The other way is to specify a comparator for the elements in the set if the class for the elements does not implement the Comparable interface, or you don't want to use the compareTo() method in the class that implements the Comparable interface. This approach is referred to as order by comparator



Using TreeSet to sort elements in a set

Using HashSet

```
Set hashSet = new HashSet();  
hashSet.add("Yellow");  
hashSet.add("White ");  
hashSet.add("Green");  
hashSet.add("Orange");  
System.out.println("An unsorted set of strings");  
System.out.println(hashSet + "\n");
```

Output: An unsorted set of strings

[Orange, Green, White, Yellow]

Using TreeSet

```
Set treeSet = new TreeSet(hashSet);  
System.out.println("Sorted set of strings");  
System.out.println(treeSet + "\n");
```

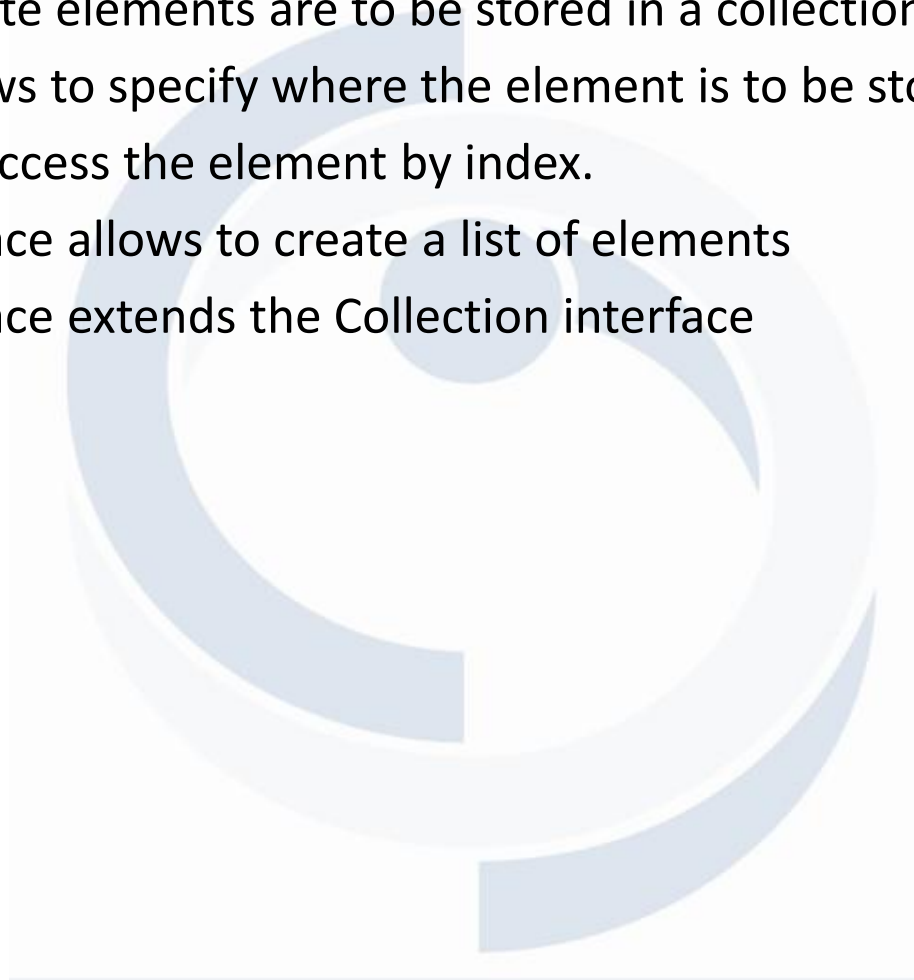
Output: A sorted set of strings

[Green, Orange, White, Yellow]



The list interface

- A list allows duplicate elements in a collection
- Where duplicate elements are to be stored in a collection, list can be used.
- A list also allows to specify where the element is to be stored.
- The user can access the element by index.
- The List interface allows to create a list of elements
- The List interface extends the Collection interface



Using ArrayList and LinkedList - continued

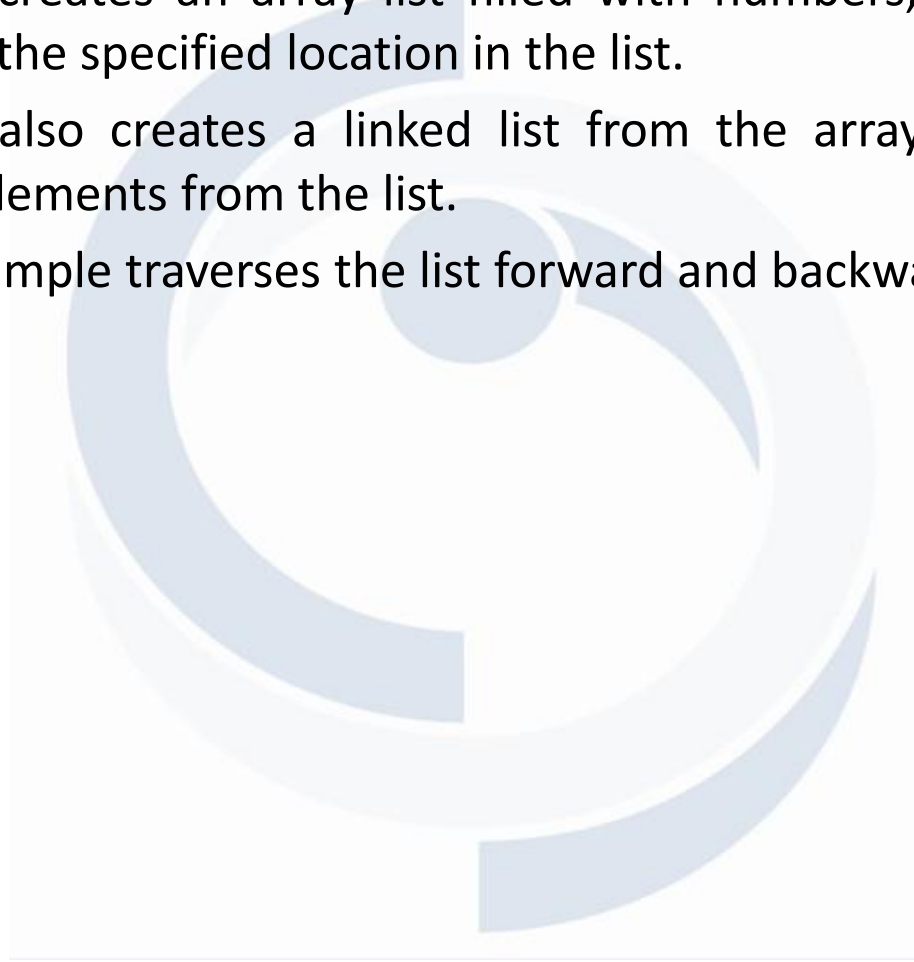
A List interface has the following methods

- +add(index: int, element: Object) : boolean*
- +addAll(index: int, collection: Collection) : boolean*
- +get(index: int) : Object*
- +indexOf(element: Object) : int*
- +lastIndexOf(element: Object) : int*
- +listIterator() : ListIterator*
- +listIterator(startIndex: int) : ListIterator*
- +remove(index: int) : int*
- +set(index: int, element: Object) : Object*
- +subList(fromIndex: int, toIndex: int) : List*



Using ArrayList and LinkedList

- ArrayList and LinkedList are two implementations of List interface.
- This example creates an array list filled with numbers, and inserts new elements into the specified location in the list.
- The example also creates a linked list from the array list, inserts and removes the elements from the list.
- Finally, the example traverses the list forward and backward.



Using ArrayList and LinkedList - continued

```
ArrayList arrayList = new ArrayList();  
arrayList.add(new Integer(1));  
arrayList.add(new Integer(2));  
arrayList.add(new Integer(3));  
arrayList.add(0, new Integer(10));  
arrayList.add(3, new Integer(20));  
System.out.println(arrayList);
```

Output:

[10,1,2,20,3]



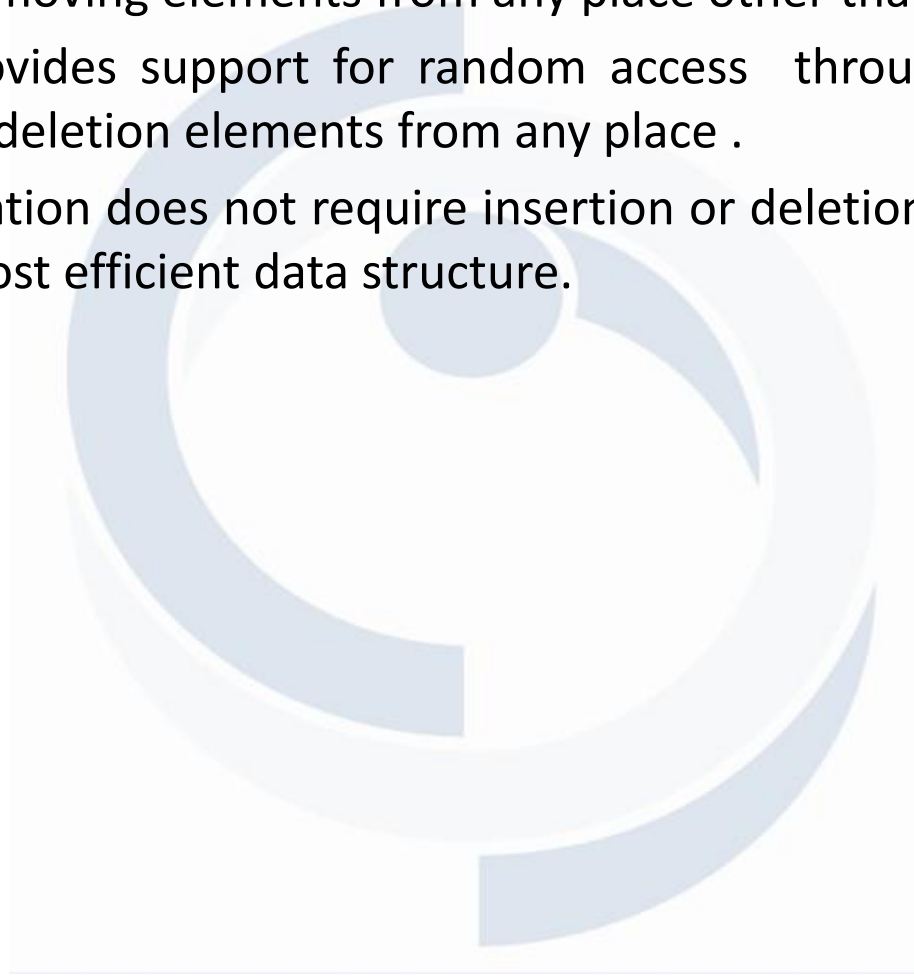
```
LinkedList linkedList = new LinkedList(arrayList);  
linkedList.add(1, "A"); //[10,"A", 1, 2, 20]  
linkedList.removeLast(); //[10,"A", 1, 2]  
linkedList.addFirst("B"); //[“B”, 10,“A”, 1, 2]  
System.out.println(linkedList);
```

Output:

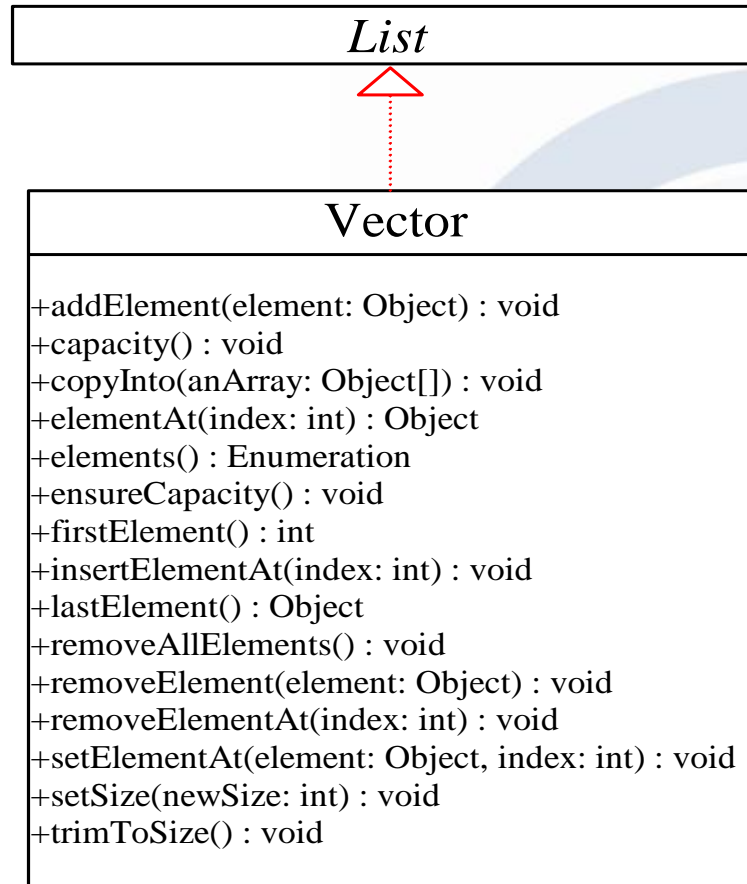
```
[“B”, 10, “A”, 1, 2, 20]
```

ArrayList vs. LinkedList

- ArrayList provides support random access through an index without inserting or removing elements from any place other than an end.
- LinkedList provides support for random access through an index with inserting and deletion elements from any place .
- If your application does not require insertion or deletion of elements, the Array is the most efficient data structure.

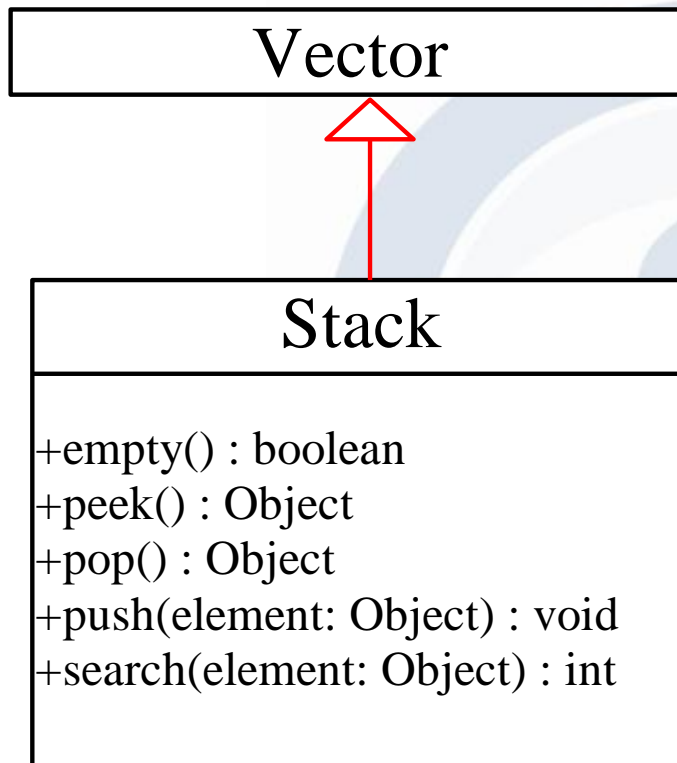


The Vector class



- The *Vector* class implements a growable array of objects.
- Like an array, it contains components that can be accessed using an integer index.
- However, the size of a *Vector* can grow or shrink as needed to accommodate adding and removing items after the *Vector* has been created.

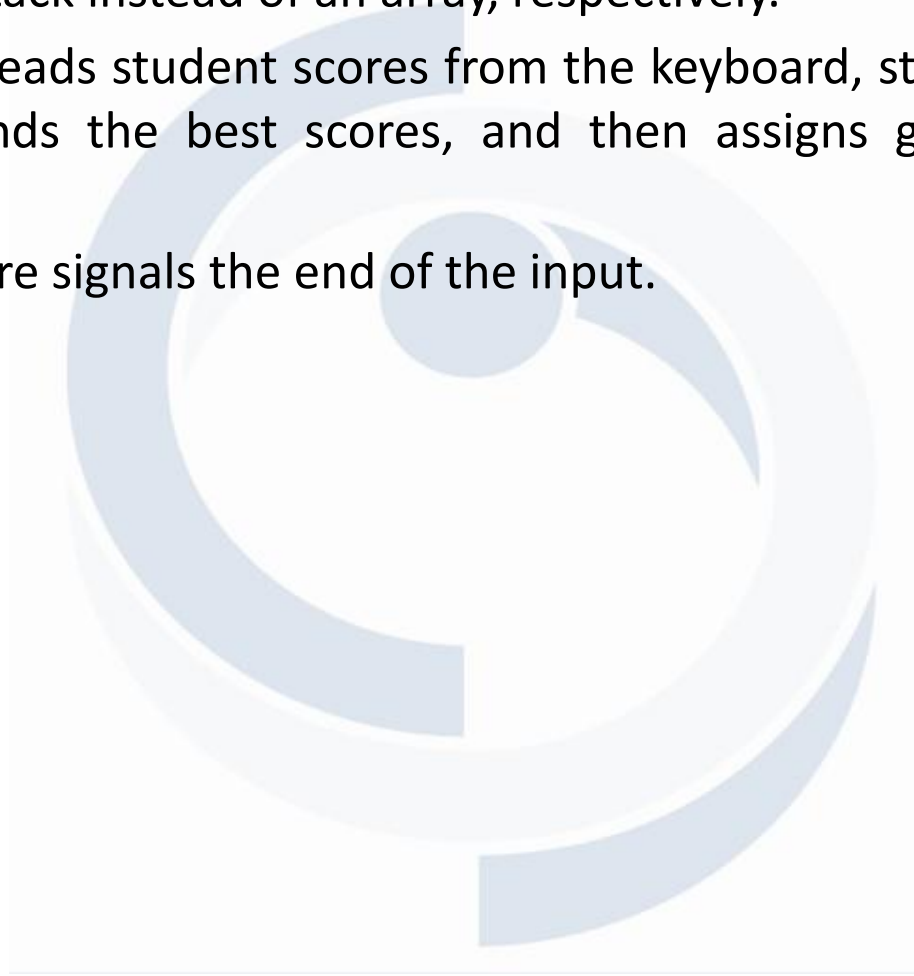
The Stack class



- The **Stack** class represents a last-in-first-out (LIFO) stack of objects. The elements are accessed only from the top of the stack. You can retrieve, insert, or remove an element from the top of the stack.

Using vector and stack

- This example presents two programs to rewrite Example 5.2, using a vector and a stack instead of an array, respectively.
- The program reads student scores from the keyboard, stores the scores in the vector, finds the best scores, and then assigns grades for all the students.
- A negative score signals the end of the input.



Map interface

- A Map is a storage that maps keys to values. There cannot be duplicate keys in a Map and each key maps to at most one value.
- The Map interface is not an extension of Collection interface. Instead the interface starts of it's own interface hierarchy.
- for maintaining key-value associations. The interface describes a mapping from keys to values, without duplicate keys, by definition.
- The Map interface maps keys to the elements. The keys are like indexes. In List, the indexes are integer. In Map, the keys can be any objects.



Map interface - continued

Removes all mappings from this map (optional operation)

- `void clear()`
- `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

- `Set<Map.Entry<K,V>> entrySet()`

Returns a set view of the mappings contained in this map

- `boolean equals(Object o)`

Compares the specified object with this map for equality

- `V get(Object key)`

Returns the value to which this map maps the specified key

- `int hashCode()`

Returns the hash code value for this map

- `boolean isEmpty()`



Map interface - continued

Returns true if this map contains no key-value mappings

- `Set<K> keySet()`

Returns a set view of the keys contained in this map

- `V put(K key, V value)`

Associates the specified value with the specified key in this map (optional operation)

- `void putAll(Map<? extends K,? extends V> t)`

Copies all of the mappings from the specified map to this map (optional operation)

- `V remove(Object key)`

Removes the mapping for this key from this map if it is present (optional operation).

- `int size()`

Returns the number of key-value mappings in this map

- `Collection<V> values()`

Returns a collection view of the values contained in this map



HashMap and TreeMap

- The HashMap and TreeMap classes are two concrete implementations of the Map interface. Both will require key & value. Keys must be unique.
- The HashMap class is efficient for locating a value, inserting a mapping, and deleting a mapping.
- The TreeMap class, implementing SortedMap, is efficient for traversing the keys in a sorted order.
- HashMap allows null as both keys and values
- TreeMap is slower than HashMap
- TreeMap allows us to specify an optional Comparator object during its creation. This comparator decides the order by which the keys need to be sorted.



Example: Map interface

```
import java.util.*;

class MapEx{
    public static void main(String args[]) {
        Map<String,String> map = new HashMap<String,String>();
        map.put ("1", "one");
        map.put ("2", "two");
        System.out.println(map.size() ); //print 2
        System.out.println(map.get("1") ); //print "one"
        Set keys = map.keySet();    // print the keys
        for (Object object : keys) {    System.out.println(object);    }
    }
}
```



The array class

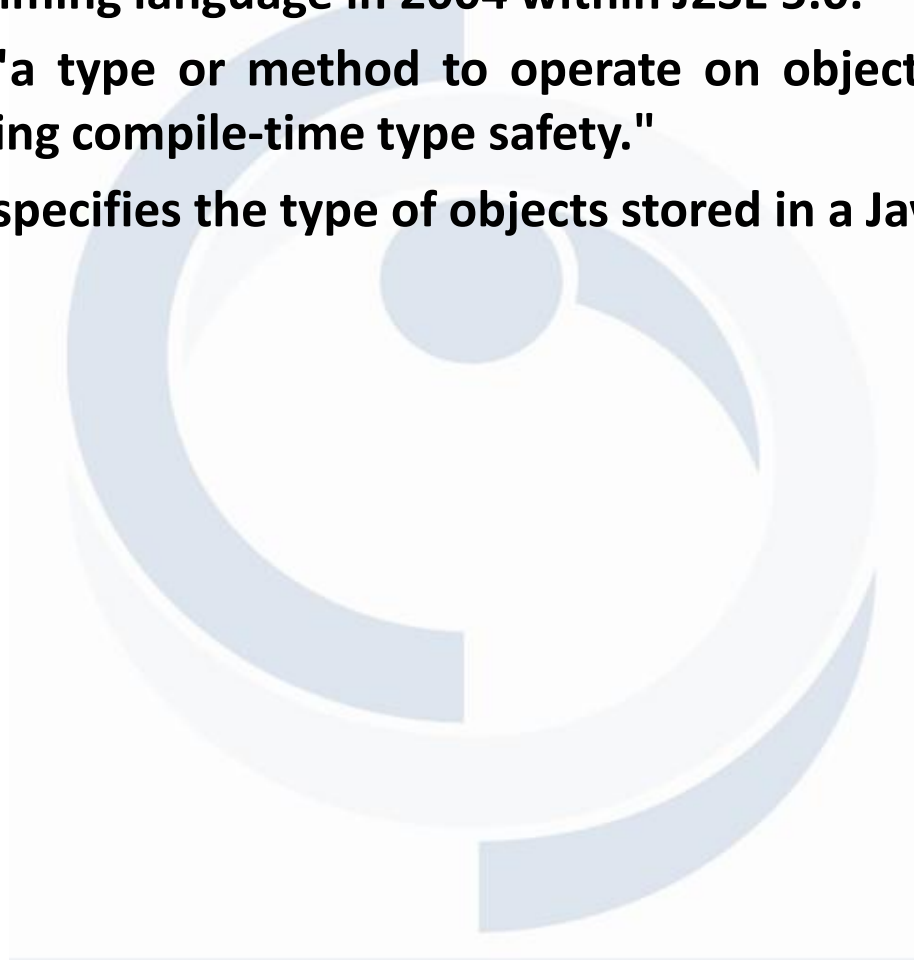
- The Collections class contains various static methods for operating on collections and maps, for creating synchronized collection classes, and for creating read-only collection classes.

Arrays
<u>+asList(a: Object[]) : List</u> <u>+binarySearch(a: byte[],key: byte) : int</u> <u>+binarySearch(a: char[], key: char) : int</u> <u>+binarySearch(a: double[], key: double) : int</u> <u>+binarySearch(a: float[] key: float) : int</u> <u>+binarySearch(a: int[], key: int) : int</u> <u>+binarySearch(a: long[], key: long) : int</u> <u>+binarySearch(a: Object[], key: Object) : int</u> <u>+binarySearch(a: Object[], key: Object, c: Comparator) : int</u> <u>+binarySearch(a: short[], key: short) : int</u> <u>+equals(a: boolean[], a2: boolean[]) : boolean</u> <u>+equals(a: byte[], a2: byte[]) : boolean</u> <u>+equals(a: char[], a2: char[]) : boolean</u> <u>+equals(a: double[], a2: double[]) : boolean</u> <u>+equals(a: float[], a2: float[]) : boolean</u> <u>+equals(a: int[], a2: int[]) : boolean</u> <u>+equals(a: long[], a2: long[]) : boolean</u> <u>+equals(a: Object[], a2: Object[]) : boolean</u> <u>+equals(a: short[], a2: short[]) : boolean</u> <u>+fill(a: boolean[], val: boolean) : void</u> <u>+fill(a: boolean[], fromIndex: int, toIndex: int, val: boolean) : void</u> Overloaded fill method for char, byte, short, int, long, float, double, and Object. <u>+sort(a: byte[]) : void</u> <u>+sort(a: byte[], fromIndex: int, toIndex: int) : void</u> Overloaded sort method for char, short, int, long, float, double, and Object.



Generics

- **Generics are a facility of generic programming that were added to the Java programming language in 2004 within J2SE 5.0.**
- **They allow "a type or method to operate on objects of various types while providing compile-time type safety."**
- **This feature specifies the type of objects stored in a Java Collection.**



Advantage of Generics

- There are mainly 3 advantages of generics. They are as follows:
 - 1) Type-safety : We can hold only a single type of objects in generics.
It doesn't allow to store other objects.
 - 2) Type casting is not required: There is no need to typecast the object.
- Before Generics, we need to type cast.

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0); //typecasting
```



Advantages of Generics

- After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
String s = list.get(0);
```

- 3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
list.add(32);//Compile Time Error
```



Generic Classes

- A class that can refer to any type is known as generic class. Here, we are using T type parameter to create the generic class of specific type.
- Let's see the simple example to create and use the generic class.

```
class MyGen<T>{  
    T obj;  
    void add(T obj){this.obj=obj;}  
    T get(){return obj;}  
}
```

- The T type indicates that it can refer to any type (like String, Integer, Employee etc.). The type you specify for the class, will be used to store and retrieve the data.



Using Generic classes

```
class Test{  
  
    public static void main(String args[]){  
  
        MyGen<Integer> m=new MyGen<Integer>();  
  
        m.add(2);  
  
        //m.add("vivek");//Compile time error  
  
        System.out.println(m.get());  
  
    }  
}
```



Generic Methods

Like generic class, we can create generic method that can accept any type of argument.

```
public class TestGenerics4{
    public static < E > void printArray(E[] elements) {
        for ( E element : elements){
            System.out.println(element );
        }
        System.out.println();
    }
    public static void main( String args[] ) {
        Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };
        System.out.println( "Printing Integer Array" );
        printArray( intArray );
        System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}
```



Upper Bound Wildcard Generics

- The ? (question mark) symbol represents wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number e.g. Integer, Float, double etc. Now we can call the method of Number class through any child class object.

```
abstract class Shape{ abstract void draw(); }

class Rectangle extends Shape{
    void draw(){
        System.out.println("drawing rectangle");
    } }

class Circle extends Shape{
    void draw() {
        System.out.println("drawing circle");
    }
}
```



Wildcard Generics

```
class GenericTest {  
    //creating a method that accepts only child class of Shape  
    public static void drawShapes(List<? extends Shape> lists){  
        for(Shape s:lists){ s.draw();//calling method of Shape class by child class  
                                instance  
        }  
    }  
  
    public static void main(String args[]){  
        List<Rectangle> list1=new ArrayList<Rectangle>();  
        list1.add(new Rectangle());  
        List<Circle> list2=new ArrayList<Circle>();  
        list2.add(new Circle());  
        list2.add(new Circle());  
        drawShapes(list1);  
        drawShapes(list2);  
    } }  
}
```



Lower Bound Wildcard Generics

- A *lower bounded* wildcard restricts the unknown type to be a specific type or a *super type* of that type.
- A lower bounded wildcard is expressed using the wildcard character ('?'), following by the super keyword, followed by its *lower bound*:
- `<? super A>`. It will accept A and subtype of A

```
public static void addNumbers(List<? super Integer> list)
{
    for (int i = 1; i <= 10; i++) { list.add(i);
    }
```

It will accept Integer, Number, and Object

