COLLECTIONS FRAMEWORK

Mrs.S.Lakshmi Priya

Assistant Professor/CSE

SSNCE

JAVA.UTIL

- The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects.
- Some top level classes:
  - Calendar
  - Stack
  - Vector
  - LinkedList
  - HashMap
  - TreeSet etc.,

JAVA.UTIL – SOME INTERFACES

- Collection
- Comparator
- List
- Iterator
- ListIterator
- EventListener
- Map
- Set
- Queue etc.,

WHY COLLECTIONS?

- The Java Collections Framework standardizes the way in which groups of objects are handled by your programs.
- Prior to the Collections Framework, Java provided ad hoc classes such as **Dictionary**, **Vector**, **Stack**, and **Properties** to store and manipulate groups of objects.
- Why Collections?
  - Although these classes were quite useful, they lacked a central, unifying theme.
  - The way that you used **Vector** was different from the way that you used **Properties**, for example.
  - This ad hoc approach was not designed to be easily extended or adapted.

DESIGN GOALS

- Need of high performance
  - No need to code the“data engines” (Vector, Stack etc.,) manually

- High degree of interoperability between classes

- Extending/adapting collections should be easy

ALGORITHMS & ITERATOR

- ***Algorithms*** are another important part of the collection mechanism.
  - Algorithms operate on collections and are defined as static methods within the **Collections** class. Thus, they are available for all collections.

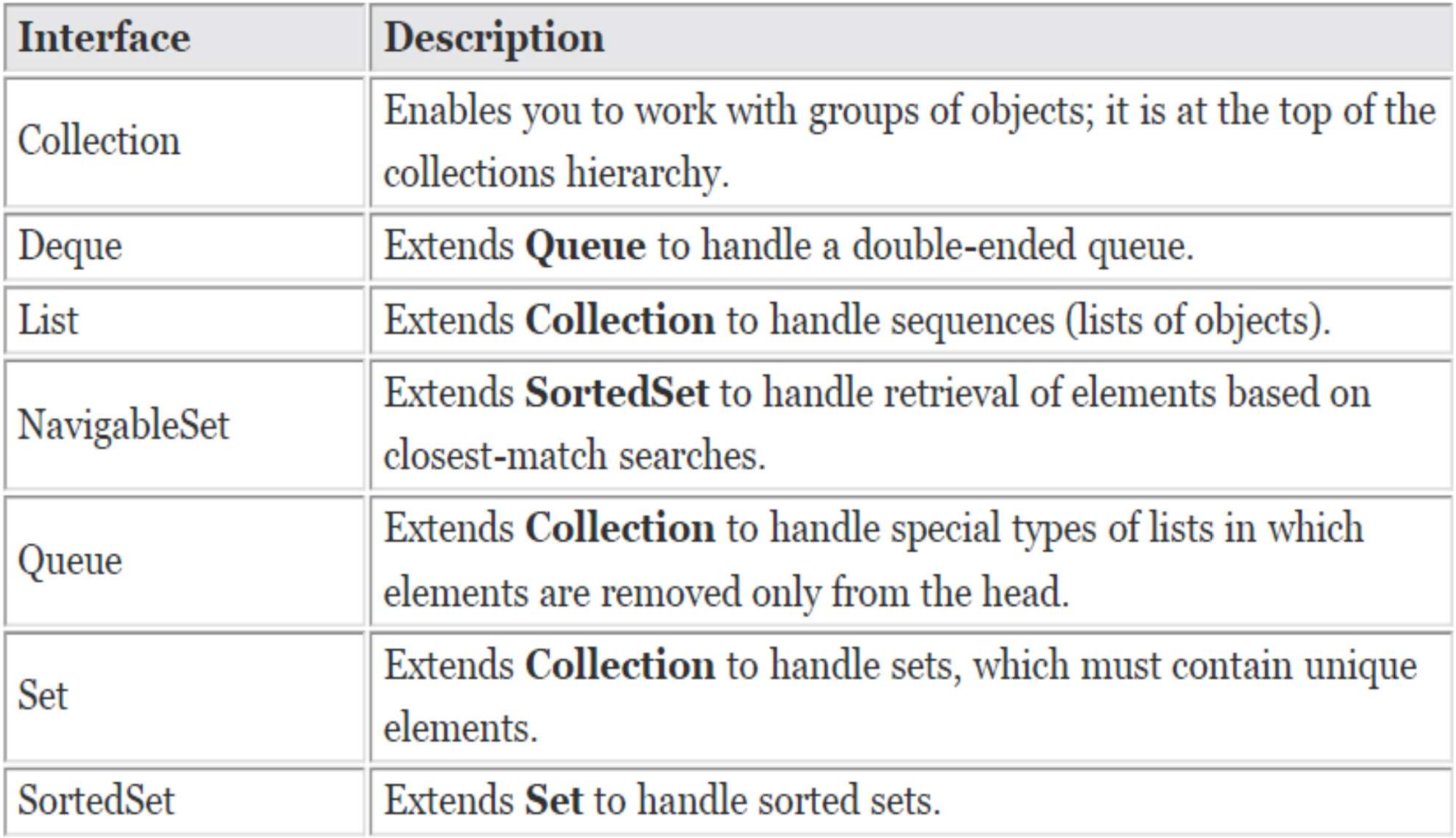
- **Iterator** interface.
  - An *iterator* offers a general-purpose, standardized way of accessing the elements within a collection, one at a time.
  - Because each collection provides an iterator, the elements of any collection class can be accessed through the methods defined by **Iterator**.

GENERICS & COLLECTIONS

- The entire Collections Framework was reengineered for generics.
- All collections are now generic, and many of the methods that operate on collections take generic type parameters.
- Generics added the one feature that collections had been missing: type safety.

- Prior to generics, all collections stored **Object** references, which meant that any collection could store any type of object.
  - Led to accidentally storing incompatible types in a collection.
  - Resulted in run-time type mismatch errors.
  - With generics, it is possible to explicitly state the type of data being stored, and run-time type mismatch errors can be avoided.

COLLECTIONS - INTERFACES



Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
Deque	Extends **Queue** to handle a double-ended queue.
List	Extends **Collection** to handle sequences (lists of objects).
NavigableSet	Extends **SortedSet** to handle retrieval of elements based on closest-match searches.
Queue	Extends **Collection** to handle special types of lists in which elements are removed only from the head.
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.

OTHER INTERFACES

- **Comparator** defines how two objects are compared;
- **Iterator**, **ListIterator**, and **Spliterator** enumerate the objects within a collection.
- By implementing **RandomAccess**, a list indicates that it supports efficient, random access to its elements.
- To provide the greatest flexibility in their use, the collection interfaces allow some methods to be optional. The optional methods enable you to modify the contents of a collection.
- Collections that support these methods are called *modifiable*.
- Collections that do not allow their contents to be changed are called *unmodifiable*.
  - If an attempt is made to use one of these methods on an unmodifiable collection, an **UnsupportedOperationException** is thrown.
- **All the built-in collections are modifiable.**

COLLECTION INTERFACE

- The **Collection** interface is the foundation upon which the Collections Framework is built because it must be implemented by any class that defines a collection.

- **Collection** is a generic interface that has this declaration:

  interface Collection<E>

- Here, **E** specifies the type of objects that the collection will hold.

- **Collection** extends the **Iterable** interface. This means that all collections can be cycled through by use of the for-each style **for** loop.

COLLECTIONS – METHODS & EXCEPTIONS

- **Collection** declares the core methods that all collections will have.
- Several of these methods can throw an
- **UnsupportedOperationException -** occurs if a collection cannot be modified.
- **ClassCastException** – occurs when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a collection.
- **NullPointerException -** occurs if an attempt is made to store a **null** object and **null** elements are not allowed in the collection.
- **IllegalArgumentException -** occurs if an invalid argument is used.
- **IllegalStateException -** occurs if an attempt is made to add an element to a fixed-length collection that is full.

COLLECTION INTERFACE - METHODS

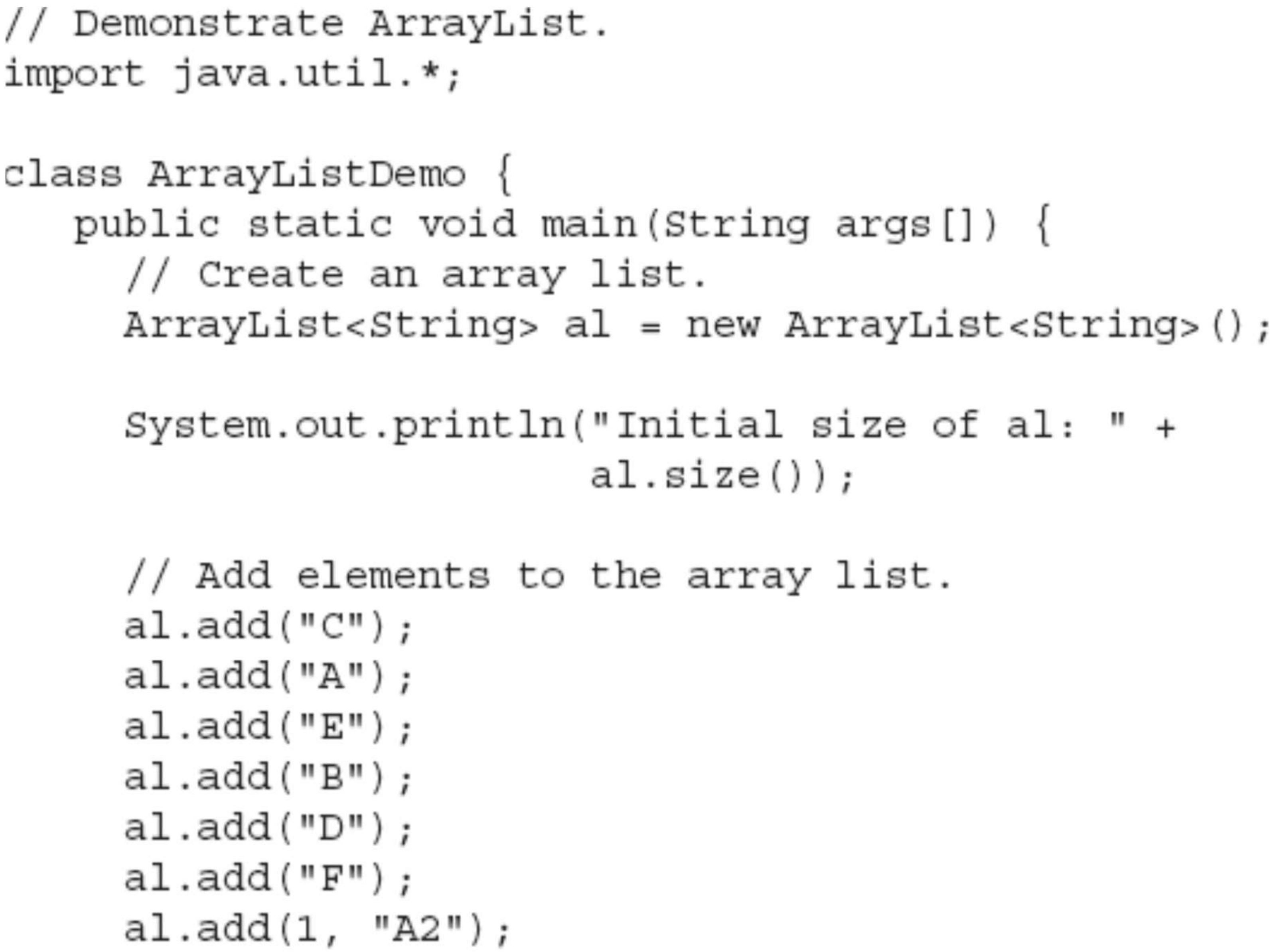
No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.

COLLECTION INTERFACE - METHODS

10	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
12	public Object[] toArray()	It converts collection into array.
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public boolean isEmpty()	It checks if collection is empty.
15	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
17	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
18	public boolean equals(Object element)	It matches two collections.

SIMPLE EXAMPLE - ARRAYLIST

- **ArrayList** supports dynamic arrays that can grow as needed
- **ArrayList** is a variable-length array of object references.
- An **ArrayList** can dynamically increase or decrease insize.
- Array lists are created with an initial size
- When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.



```
// Demonstrate ArrayList.
import java.util.*;

class ArrayListDemo {
   public static void main(String args[]) {
     // Create an array list.
     ArrayList<String> al = new ArrayList<String>();

     System.out.println("Initial size of al: " +
                        al.size());

     // Add elements to the array list.
     al.add("C");
     al.add("A");
     al.add("E");
     al.add("B");
     al.add("D");
     al.add("F");
     al.add(1, "A2");
```

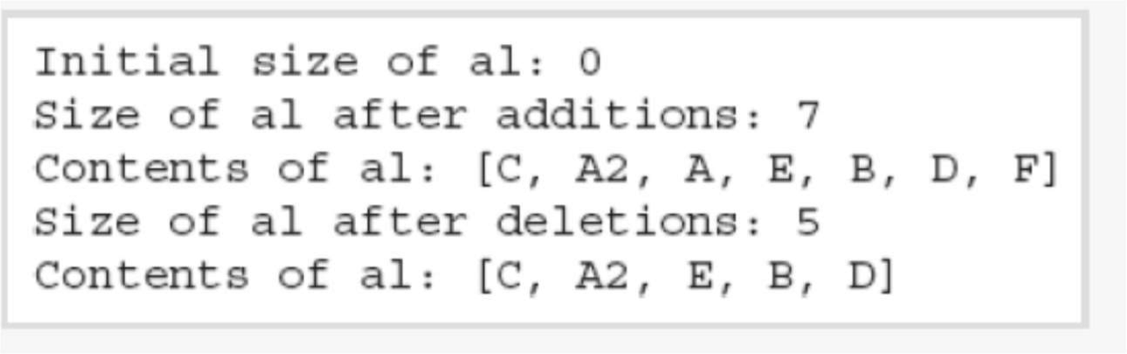
```
    System.out.println("Size of al after additions: " +
                       al.size());

    // Display the array list.
    System.out.println("Contents of al: " + al);

    // Remove elements from the array list.
    al.remove("F");
    al.remove(2);

    System.out.println("Size of al after deletions: " +
                       al.size());

    System.out.println("Contents of al: " + al);
  }
}
```



```
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```