CS205 Object Oriented Programming in Java

Module 4 - **Advanced features of Java**

(Part 4)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

☑ Java Library

- ❑**Collections framework**
  - ❑ List Interface
  - ❑Collections Class
  - ❑ArrayList Class

Prepared by Renetha J.B.

List Interface

- The **List** interface **extends** *Collection* interface.

- **List** declares the behavior of a collection that stores a sequence of elements.

  - In Java, the **List** interface is an **ordered collection** that allows us to **store and access elements sequentially.**

- Elements can be inserted or accessed by **their position** in the list, using zero-based index.

- A list may contain **duplicate elements**.

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

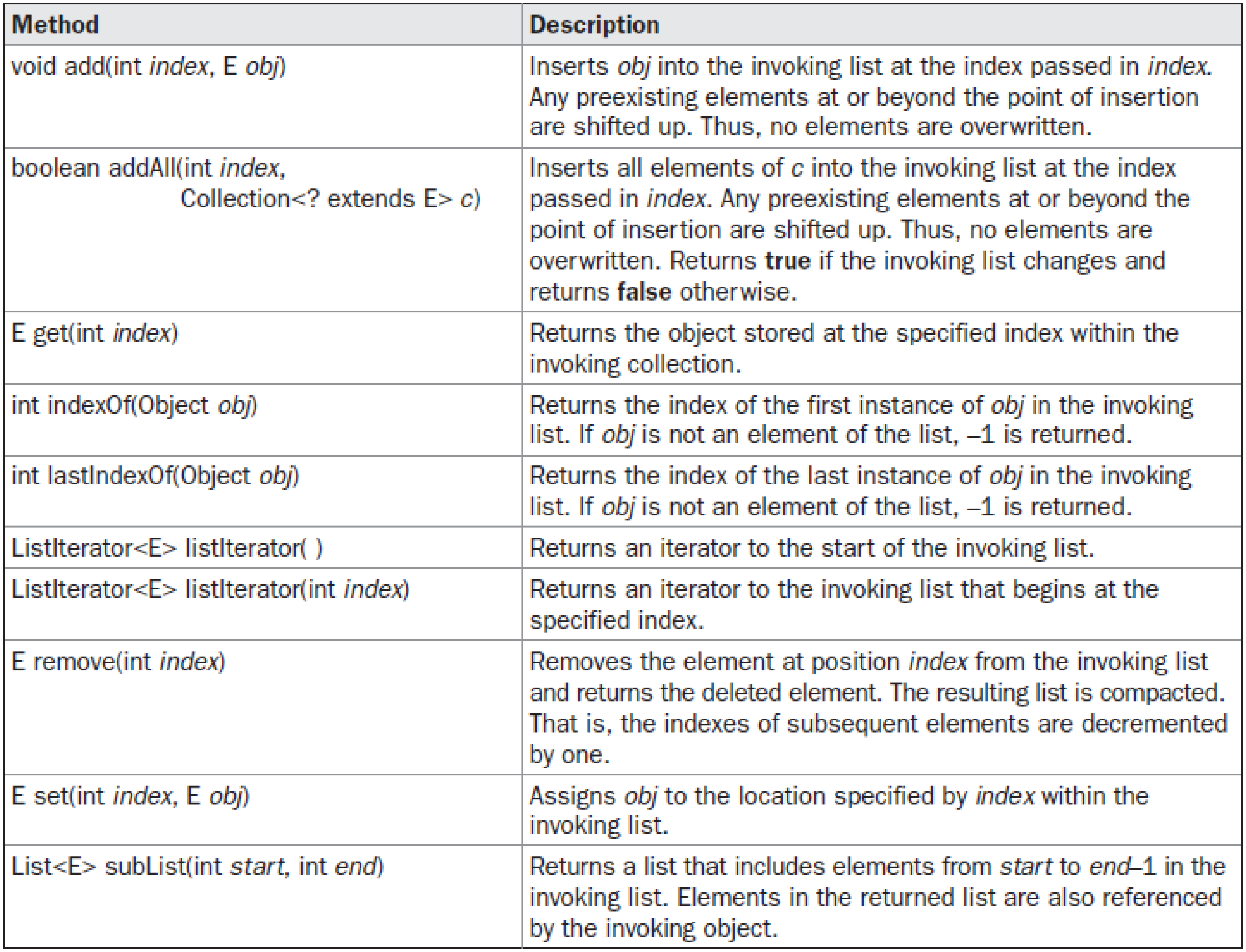
```
interface List<E>
```

List Interface(contd.)

- List supports methods defined by **Collection**,
- List defines its own methods also.
- Some methods throw exceptions.
- **Exceptions** that are thrown by List methods are:

UnsupportedOperationException	• if the list cannot be modified
ClassCastException	• when one object is incompatible with another
IndexOutOfBoundsException	• if an invalid index is used
NullPointerException	• thrown if an attempt is made to store a null object and null elements are not allowed in the list.
IllegalArgumentException	• if an invalid argument is used.

Methods in List interface



Method	Description
void add(int *index*, E *obj*)	Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
boolean addAll(int *index*, Collection<? extends E> *c*)	Inserts all elements of *c* into the invoking list at the index passed in *index*. Any preexisting 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.
E get(int *index*)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object *obj*)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
int lastIndexOf(Object *obj*)	Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
ListIterator<E> listIterator( )	Returns an iterator to the start of the invoking list.
ListIterator<E> listIterator(int *index*)	Returns an iterator to the invoking list that begins at the specified index.
E remove(int *index*)	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.
E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within the invoking list.
List<E> subList(int *start*, int *end*)	Returns a list that includes elements from *start* to *end*–1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

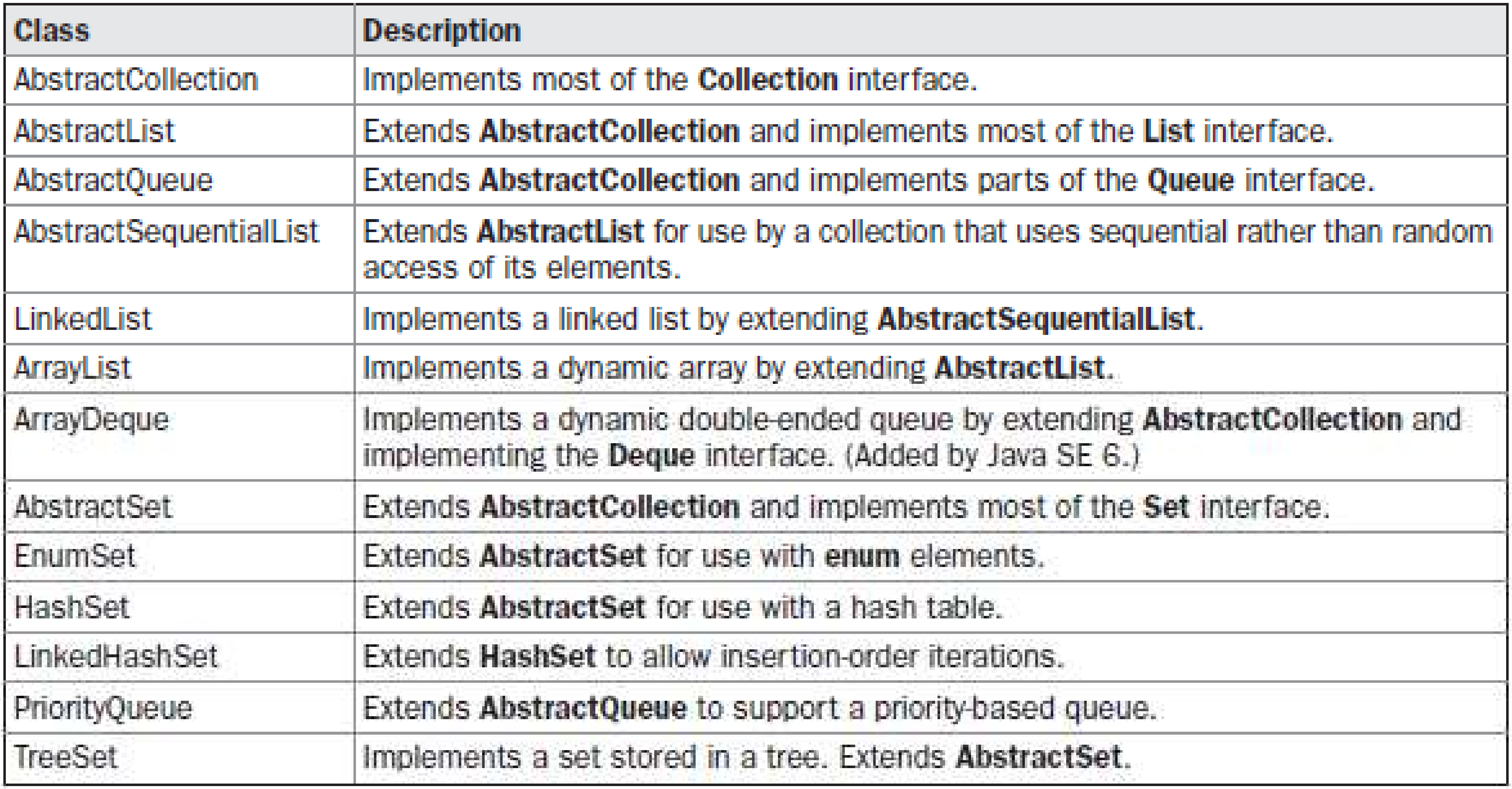
Methods in List interface (contd.)

- List has many methods:-
- **add(int, E)** and **addAll(int, Collection)**
  - These methods **insert elements at the specified index.**
- The meaning of **add(E)** and **addAll(Collection)** defined by Collection are changed by List. In **List** **they add elements to the end of the list.**
- To **obtain the object stored at a specific location**, call **get( )** with the index of the object.
- To **assign a value to an element in the list**, call **set( )**, specifying the index of the object to be changed.
- To **find the index of an object**, use **indexOf( )** or **lastIndexOf()**.
- A **sublist of a list can be obtained** by calling **subList()** **,** specifying the beginning and ending indexes of the sublist.

The Collection Classes

- The collection classes **implement collection interfaces**.
- Some of the collection classes provide **full implementations** that can be used as-is.
- Some of the collection classes are **abstract**, providing **skeletal implementations** that are used as starting points for creating concrete collections.
- Collection classes are **not synchronized.**
  - Two or more threads can access the methods of collection class at any time.

The standard collection classes are



Class	Description
AbstractCollection	Implements most of the **Collection** interface.
AbstractList	Extends **AbstractCollection** and implements most of the **List** interface.
AbstractQueue	Extends **AbstractCollection** and implements parts of the **Queue** interface.
AbstractSequentialList	Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending **AbstractSequentialList**.
ArrayList	Implements a dynamic array by extending **AbstractList**.
ArrayDeque	Implements a dynamic double-ended queue by extending **AbstractCollection** and implementing the **Deque** interface. (Added by Java SE 6.)
AbstractSet	Extends **AbstractCollection** and implements most of the **Set** interface.
EnumSet	Extends **AbstractSet** for use with **enum** elements.
HashSet	Extends **AbstractSet** for use with a hash table.
LinkedHashSet	Extends **HashSet** to allow insertion-order iterations.
PriorityQueue	Extends **AbstractQueue** to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends **AbstractSet**.

ArrayList Class

- The **ArrayList class** **extends AbstractList** and implements the **List** interface.

- ArrayList is a generic class that has declaration:

```
class ArrayList<E>
```

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

- **ArrayList** supports **dynamic arrays** that can grow as needed.
  - This is needed because in some cases we may not know how large an array we need precisely until run time.

ArrayList Class(contd.)

- An **ArrayList** is a **variable-length array** of object references.
  - So **ArrayList** can dynamically increase or decrease in size.
- 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**.

ArrayList Class(contd.)

- **ArrayList** has following **constructors**:

ArrayList( )

- ➢ This constructor builds an **empty** array list.

ArrayList(Collection<? extends E> *c)*

- – This constructor builds an array list that is **initialized with the elements of the collection c**.

ArrayList(int *capacity)*

- – This constructor builds an array list that has the specified initial capacity.
- – The capacity is the **size** of the underlying array that is used to store the elements.
- – The capacity **grows automatically** as elements are added to an array list.

ArrayList Class(contd.)

```
import java.util.*;
class ArrayListDemo {
public static void main(String args[]) {
ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size=" +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 now=" +al.size());
System.out.println("Contents : " + al);
al.remove("F");
al.remove(2);
System.out.println("Size=" +al.size());
System.out.println("Contents=" +al);
}
}
```

```
Initial size=0
Size now=7
Contents : [C, A2, A, E, B, D, F]
Size=5
Contents=[C, A2, E, B, D]
```

ArrayList Class(contd.)

- The contents of a collection are displayed using the default conversion provided by **toString( )**, which was inherited from **AbstractCollection**.

- We can increase the capacity of an **ArrayList** object manually by calling **ensureCapacity( ).**

```
void ensureCapacity(int cap)
```

- If we want to **reduce** the size of the array that of**ArrayL ist** object so that it is precisely as large as the number of items that it is currently holding, call **trimToSize( )**:

```
void trimToSize( )
```

Obtaining an Array from an ArrayList

- To **convert a collection into an array**, **toArray( ),** which is defined by **Collection** can be called.
  - This is needed
    - To obtain **faster processing times** for certain operations
    - To **pass an array to a method** that is not overloaded to accept a collection
    - To **integrate collection-based code with legacy code** that does not understand collections
- Two versions of **toArray( )** are:

Object[ ] toArray( )

<T> T[ ] toArray(T *array[ ])*

Prepared by Renetha J.B.

ArrayList Class(contd.)

```
import java.util.*;
class ArrayListToArray {
public static void main(String args[]) {
ArrayList<Integer> al = new ArrayList<Integer>();

al.add(1);
al.add(2);
al.add(3);
al.add(4);
```

```
System.out.println("Contents of al: " + al);
Integer arr[] = new Integer[al.size()];
arr = al.toArray(arr);
int sum = 0;
for(int i : arr) sum += i;
System.out.println("Sum is: " + sum);
}
}
```

```
Contents of al: [1, 2, 3, 4]
Sum is: 10
```

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.