Web Technology 17
Collection Framework & Exploring java.lang

Chittaranjan Pradhan
School of Computer Engineering,
KIIT University

Collection

Collection

- java.util package has an interface called *Collection*. It is a framework that provides an architecture to store and manipulate the group of objects
- Java Collection simply means a single unit of objects, i.e. a group
- This framework provides many interfaces(ex. Queue) and classes(ex. LinkedList)
- The Collection framework was designed to meet several goals:
  - The framework had to be high-performance
  - The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability
  - Extending or adapting a collection had to be easy

Collection...

Collection...

- *Algorithms* are another important part of the collection mechanism. Algorithms operate on collections and are defined as static methods within the **Collections** class. The algorithms provide a standard means of manipulating collections
- *Iterator* interface 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
- In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs

Collection Interfaces

Collection Interface

- It 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 as:
  interface Collection <E>
  E specifies the type of objects that the collection will hold
- Collection extends the Iterable interface, which declares the core methods that all collections will have
  - **add()** adds objects to a collection. **addAll()** adds the entire contents of one collection to another
  - **remove()** removes an object. **removeAll()** removes a group of objects
  - **clear()** is used to empty a collection
  - **contains()** determines whether a collection contains a specific object. **containsAll()** determines whether one collection contains all members of another
  - **isEmpty()** determines when a collection is empty
  - **size()** determines the number of elements currently held in a collection
  - **equals()** compares two collections
  - **iterator()** returns an iterator to a collection

Collection Interfaces...

List Interface

- List interface extends Collection
- It contains methods to insert & delete elements in index basis
- A list may contain duplicate elements
  interface List <E>
  here, E specifies the type of objects that the list will hold
- List defines its own methods in addition to the methods defined in Collection
  - **add(int, E)** and **addAll(int, Collection)** insert elements at the specified index
  - **get()** returns the object stored at the specified index
  - **set()** assigns a value to an element in the list
  - **indexOf()** and **lastIndexOf()** find the index of an object
  - **subList(int start, int end)** obtains a sublist of a list
  - **sort()** sorts a list

Collection Interfaces...

Set Interface

- Set interface defines a set. It extends Collection and specifies the behavior of a collection that doesn't allow duplicate elements
  interface Set <E>
  Here, E specifies the type of objects that the set will hold
- *add()* returns false if an attempt is made to add duplicate elements to a set
- It doesn't specify any additional methods of its own

Collection Interfaces...

Queue Interface

- Queue interface extends Collection and declares the behavior of a queue, which is a FIFO list
  interface Queue <E>
  Here, E specifies the type of objects that the queue will hold
- Elements can only be removed from the head of the queue
- There are two methods that obtain and remove elements: **poll()** and **remove()**. poll() returns null if the queue is empty; whereas remove throws an exception
- There are two methods that obtain but don't remove the element at the head of the queue: **element()** and **peek()**. element() throws an exception if the queue is empty; whereas peek() returns null
- **offer()** attempts to add an element to a queue

Collection Interfaces...

Deque Interface

- Deque interface extends Queue and declares the behavior of a double-ended queue. Deques can function as standard, first-in, first-out queues or as last-in, first-out stacks
  interface Deque <E>
  Here, E specifies the type of objects that the deque will hold
- Along with the methods of Queue, Deque adds the following:
  - **push()** and **pop()** methods enable a Deque to function as a stack
  - **descendingIterator()** returns an iterator that returns elements in reverse order

Collection Classes

Collection Classes

- AbstractCollection
- AbstractList
- AbstractQueue
- LinkedList
- ArrayList
- ArrayDeque
- AbstractSet
- PriorityQueue

ArrayList Class

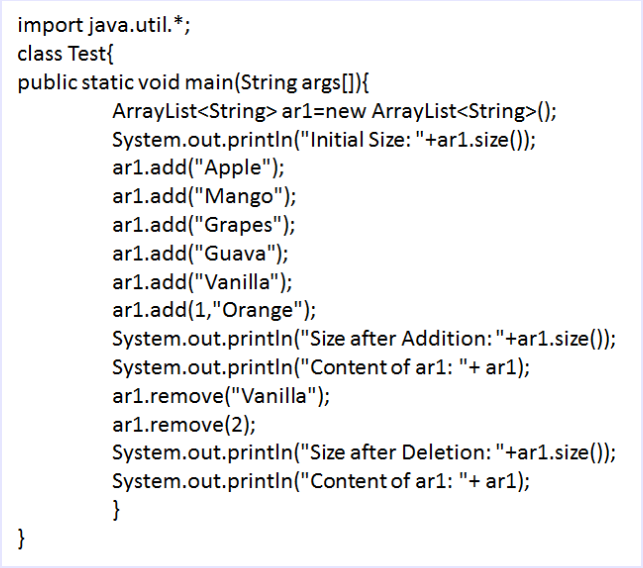
ArrayList Class

- ArrayList class extends AbstractList and implements the List interface
  class ArrayList <E>
  Here, E specifies the type of objects that the list will hold
- ArrayList supports dynamic array
- ArrayList class can contain duplicate elements
- It allows random access
- In ArrayList, manipulation is slow
- Constructors:
  ArrayList()
  ArrayList(Collection <? extends E> c)
  ArrayList(int capacity)

- **void ensureCapacity(int cap)**: increases the capacity
- **void trimToSize()**: reduces the size

ArrayList Class...

ArrayList Class...

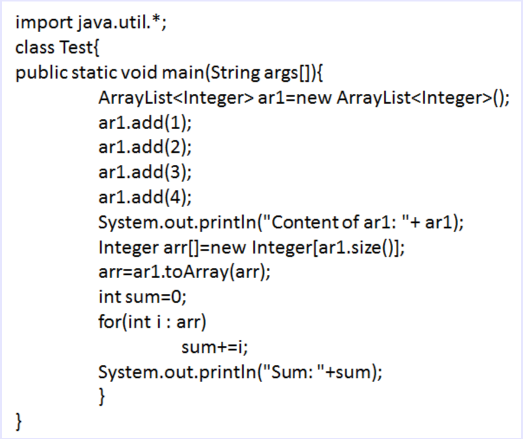


```
import java.util.*;
class Test{
public static void main(String args[]){
            ArrayList<String> ar1=new ArrayList<String>();
            System.out.println("Initial Size: "+ar1.size());
            ar1.add("Apple");
            ar1.add("Mango");
            ar1.add("Grapes");
            ar1.add("Guava");
            ar1.add("Vanilla");
            ar1.add(1,"Orange");
            System.out.println("Size after Addition: "+ar1.size());
            System.out.println("Content of ar1: "+ ar1);
            ar1.remove("Vanilla");
            ar1.remove(2);
            System.out.println("Size after Deletion: "+ar1.size());
            System.out.println("Content of ar1: "+ ar1);
            }
}
```

ArrayList Class...

Obtaining Array from ArrayList

- **toArray()** obtains the actual array from ArrayList
  object[] toArray(): returns an array of Object
  <T>T[] toArray(T array[]): returns an array of elements that have the same type as T



```
import java.util.*;
class Test{
public static void main(String args[]){
        ArrayList<Integer> ar1=new ArrayList<Integer>();
        ar1.add(1);
        ar1.add(2);
        ar1.add(3);
        ar1.add(4);
        System.out.println("Content of ar1: "+ ar1);
        Integer arr[]=new Integer[ar1.size()];
        arr=ar1.toArray(arr);
        int sum=0;
        for(int i : arr)
                sum+=i;
        System.out.println("Sum: "+sum);
        }
}
```

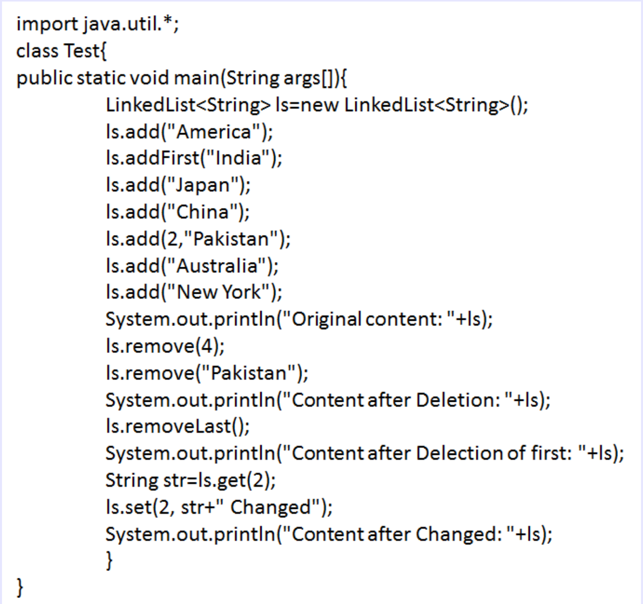
LinkedList Class

LinkedList Class

- LinkedList class extends AbstractList class and implements List, Deque interfaces
- It uses doubly linked list to store elements
  class LinkedList <E>
  Here, E specifies the type of objects that the list will hold
- Constructors:
  LinkedList()
  LinkedList(Collection <? extends E> c)
- LinkedList class can contain duplicate elements
- Here, manipulation is fast
- It can be used as list, stack or queue

LinkedList Class...

LinkedList Class...



```
import java.util.*;
class Test{
public static void main(String args[]){
            LinkedList<String> ls=new LinkedList<String>();
            ls.add("America");
            ls.addFirst("India");
            ls.add("Japan");
            ls.add("China");
            ls.add(2,"Pakistan");
            ls.add("Australia");
            ls.add("New York");
            System.out.println("Original content: "+ls);
            ls.remove(4);
            ls.remove("Pakistan");
            System.out.println("Content after Deletion: "+ls);
            ls.removeLast();
            System.out.println("Content after Delection of first: "+ls);
            String str=ls.get(2);
            ls.set(2, str+" Changed");
            System.out.println("Content after Changed: "+ls);
            }
}
```

Use of Iterator

Use of Iterator

- Each of the collection classes provides an iterator() method that returns an iterator to the start of the collection, By using the iterator object, you can access each element in the collection, one element at a time
- In general, to use an iterator to cycle through the contents of a collection, follow these steps:
    - Obtain an iterator to the start of the collection using *iterator()*
    - Set up a loop that makes a call to *hasNext()*. Continue until hasNext() returns true
    - Obtain each element by calling *next()* within the loop
- For collections implementing List, you can also obtain iterator using *listIterator()*. It accesses the collection in either forward or backward direction and allows in modifying an element

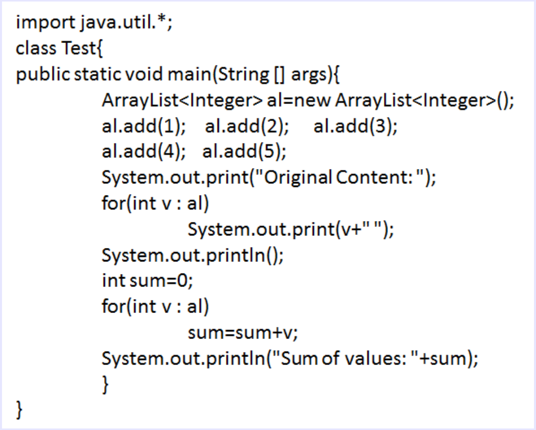
Use of Iterator...

```
import java.util.*;
class Test{
public static void main(String [] args){
          ArrayList<String> al=new ArrayList<String>();
          al.add("C");  al.add("A");  al.add("E");
          al.add("B");  al.add("D");  al.add("F");
          String str;
          System.out.print("Original Content: ");
          Iterator<String> itr=al.iterator();
          while(itr.hasNext()){
                    str=itr.next();
                    System.out.print(str+" ");
                    }
          System.out.println();
          ListIterator<String> litr=al.listIterator();
          while(litr.hasNext()){
                    str=litr.next();
                    litr.set(str+"$");
                    }
          System.out.print("Modified Content: ");
          itr=al.iterator();
          while(itr.hasNext()){
                    str=itr.next();
                    System.out.print(str+" ");
                    }
          System.out.println();
          System.out.print("Modified list backward :");
          while(litr.hasPrevious()){
                    str=litr.previous();
                    System.out.print(str+" ");
                    }
          }
}
```

For-Each Alternative to Iterator

For-Each Alternative to Iterator

- If you won’t be modifying the content of a collection or obtaining elements in reverse order, then for-each version of for loop is preferable over cycling through a collection than is using an iterator



```
import java.util.*;
class Test{
public static void main(String [] args){
        ArrayList<Integer> al=new ArrayList<Integer>();
        al.add(1);   al.add(2);    al.add(3);
        al.add(4);   al.add(5);
        System.out.print("Original Content: ");
        for(int v : al)
                System.out.print(v+" ");
        System.out.println();
        int sum=0;
        for(int v : al)
                sum=sum+v;
        System.out.println("Sum of values: "+sum);
        }
}
```

Primitive Type Wrappers

Primitive Type Wrappers

java.lang is automatically imported into all programs

- Java uses primitive types for performance reasons. These data types are not part of object hierarchy. They are passed by value to methods and can't be directly passed by reference. Also, there is no way for two methods to refer to the same instance of an int
- To address this need, java provides classes that correspond to each of the primitive types. These classes encapsulate or wrap the primitive types within a class
- **Double** and **Float** are wrappers for floating point values of type double and float, respectively
- **Byte**, **Short**, **Integer** and **Long** classes are wrappers for byte, short, int and long types, respectively
- **Character** is a wrapper around a char type
- **Boolean** is a wrapper around boolean values

Memory Management

Memory Management

- Java provides automatic garbage collection. Sometimes you need to know how large the object heap and how much of it is left
- To obtain these values, use *totalMemory()* and *freeMemory()*
- Java garbage collector runs periodically to recycle unused objects. **gc()** can be invoked using the Runtime instance to request the garbage collection

System

System

- **System** class holds a collection of static methods and variables
- The standard input, output and error output are stored in the *in*, *out* and *err* variables
- It provides a utility method for quickly copying a portion of an array
- It provides a means of loading files and libraries

Object

Object

- **Object** is a superclass of all other classes
- It is beneficial if you want to refer any object whose type you don't know
- It provides some common behaviors to all objects

finalize()

- **finalize()** is a protected and non-static method of Object class and will be available in all objects you create
- It is called by the garbage collector on an object when garbage collector determines that there are no more references to the object
- A subclass overrides the finalize() to keep those operations you want to perform before an object is destroyed. It is not automatically chained like constructors
- You can call finalize() method explicitly on an object before it is abandoned

Object...

toString()

- **toString()** method returns the string representation of the object
- If you print any object, java compiler internally invokes toString() on the object
- By overriding toString() method of the Object class, we can return values of the object
- It is also widely used for logging, debugging and for passing informative error messages to *Exception* constructors and assertions