Java Workshop

DDUC ACM STUDENT CHAPTER
DEEN DAYAL UPADHYAYAYA COLLEGE
UNIVERSITY OF DELHI

Exploring java.util

CONTAINS A LARGE COLLECTION OF CLASSES AND INTERFACES TO PROVIDE A BROAD RANGE OF FUNCTIONALITY

java.util

- Contains classes and interfaces to provide a broad range of functionality.
- Contains classes to generate random numbers, manage date and time, manipulate bits and tokenize strings.
- Contains one of Java's most powerful subsystem: ***Collections Framework***.
- Because **java.util** provides a wide array of functionality, it is quite large.

Important Classes and Packages in java.util

Classes				
AbstractColection	ArrayList	Dictionary	StringTokenizer	EnumMap
AbstractList	Arrays	Hashtable	Timer	EnumSet
AbstractMap	BitSet	LinkedList	TreeMap	Date
AbstractQueue	Scanner	Random	TreeSet	Calender
AbstractSet	Collections	Vector	PriorityQueue	Stack

Interfaces			
Collection	Enumeration	List	Queue
Comparactor	Map	ListIterator	Set
Deque	Iterator	SortedSet	SortedMap

Collections Overview

- The Java Collection Framework standardize the way in which groups of objects are handled.
- Collections were not part of original Java release, but were added by J2SE 1.2.
- The Collection Framework was designed to meet several goals:
  - It had to be high-performance.
  - It had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
  - Extending and/or adapting a collection had to be easy.

The Collection Interfaces

- Main interfaces of The Collection Framework are:

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

The Collection Interface

- The **Collection** interface is the foundation of the Collections Framework and 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 loop.
- **Collection** declares the core methods that all collections will have.

Exceptions Thrown in Collection Interface

- **UnsupportedOperationException**
  - If a collection cannot be modified and an attempt to modify it is made.
- **ClassCastException**
  - If an object is incompatible with another in an operation.
- **NullPointerException**
  - If an attempt is made to store a **null** object in a collection where **null** elements are not allowed.
- **IllegalArgumentException**
  - If an invalid argument is used.
- **IllegalStateException**
  - If an attempt is made to add an element to fixed-length collection that is full.

Methods Defined in Collection Interface

- boolean add(E *object*)
- boolean addAll(Collection<? extends E> *c*)
- void clear()
- boolean contains(Object *object*)
- boolean containsAll(Collection<?> *c*)
- boolean equals(Object *object*)
- boolean isEmpty()
- Iterator<E> iterator()
- boolean remove(Object *object*)
- boolean removeAll(Collection<?> *c*)
- boolean retainAll(Collection<?> *c*)
- int size()
- Object[] toArray()

The List Interface

- The **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements.
- Elements can be inserted or accessed by their position in list, using a zero-based index.
- A list may contain duplicate elements.
- **List** is a generic interface that has this declaration:
  - interface List<E>
    - Here, **E** specifies the type of objects that the list will hold.

Methods Defined in List Interface

- void add(int *index*, E *object*)
- boolean addAll(int *index*, Collection<? extends E> *c*)
- E get(int *index*)
- int indexOf(Object *object*)
- int lastIndexOf(Object *object*)
- ListIterator<E> listIterator()
- ListIterator<E> listIterator(int *index*)
- E remove(int *index*)
- E set(int *index*, E *object*)
- List<E> subList(int *start*, int *end*)

The Set Interface

- The **Set** interface defines a set. It extends **Collection** and declares the behavior of a collection that does not allow duplicate elements.
- It does not define any additional methods of its own.
- **Set** is a generic interface that has this declaration:
  - interface Set<E>
    - Here, **E** specifies the type of objects that the set will hold.

The SortedSet Interface

- The **SortedSet** interface extends **Set** and declares the behavior of a set sorted in ascending order.
- **SortedSet** is a generic interface that has this declaration:
  - interface SortedSet<E>
    - Here, **E** specifies the type of objects that the set will hold.
- **SortedeSet**'s some methods may throw **NoSuchElementException**.

Methods Defined in SortedSet Interface

- E first()
- E last()
- SortedSet<E> headSet(E *end*)
- SortedSet<E> subSet(E *start*, E *end*)
- SortedSet<E> tailSet(E *start*)

The NavigableSet Interface

- The **NavigableSet** interface extends **SortedSet**.
- It declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values.
- **NavigableSet** is a generic interface that has this declaration:
  - interface NavigableSet<E>
    - Here, **E** specifies the type of objects that the set will hold.

Methods Defined in NavigableSet Interface

- E ceiling(E *object*)
- E floor(E *object*)
- NavigableSet<E> headSet(E *upperBound*, boolean *include*)
- SortedSet<E> subSet(E *lowerBound*, boolean *includeLower*, E *upperBound*, boolean *includeUpper*)
- SortedSet<E> tailSet(E *lowerBound*, boolean *include*)
- E higher(E *object*)
- E lower(E *object*)
- E pollFirst()
- E pollLast()
- Iterator<E> descendingIterator()
- NavigableSet<E> descendingSet()

The Queue Interface

- The **Queue** interface extends **Collection**. It declares the behavior of a queue, which is often a first-in, first-out list. But sometimes the ordering may be based on some other criteria.
- **Queue** is a generic interface that has this declaration:
  - interface Queue<E>
    - Here, **E** specifies the type of objects that the set will hold.

Methods Defined in Queue Interface

- E element()
- E peek()
- E poll()
- E remove()
- boolean offer(E *object*)

The Deque Interface

- The **Deque** interface extends **Queue**. It declares the behavior of a double-ended queue.
- **Deque** is a generic interface that has this declaration:
  - interface Deque<E>
    - Here, **E** specifies the type of objects that the set will hold.

Methods Defined in Deque Interface

- void addFirst(E *object*)
- void addLast(E *object*)
- E getFirst()
- E getLast()
- boolean offerFirst(E *object*)
- boolean offerLast(E *object*)
- E peekFirst()
- E peekLast()
- E pollFirst()
- E pollLast()

Methods Defined in Deque Interface

- E pop()
- void push(E *object*)
- E removeFirst()
- E removeFirstOccurence(Object *object*)
- E removeLast()
- E removeLastOccurence(Object *object*)
- Iterator<E> descendingIterator()

The Enumeration Interface

- The **Enumeration** interface defines the methods by which you can *enumerate* the elements in a collection of objects.
- This legacy interface has been superseded by **Iterator**.
- Declaration of **Enumeration** interface is:
  - interface Enumeration<E>
    - Here, **E** specifies the type of elements being enumerated.
- **Enumeration** interface defines following two methods:
  - boolean hasMoreElements()
  - E nextElement()

Iterator Interface

- **Iterator** interface enables you to cycle through a collection, obtaining or removing elements.
- **Iterator** is a generic interface which is declared as:
  - interface Iterator<E>
    - Here, **E** specifies the type of objects being iterated.
- Methods declared in **Iterator** interface is:
  - boolean hasNext()
  - E next()
  - void remove()

ListIterator Interface

- **ListIterator** extends **Iterator** to allow bidirectional traversal of a list, and the modification of elements.
- **ListIterator** is a generic interface which is declared as:
  - interface ListIterator<E>
    - Here, **E** specifies the type of objects being iterated.
- Methods declared in **ListIterator** interface is:
  - boolean hasNext()
  - boolean hasPrevious()
  - E next()
  - E previous()
  - void add(E *object*)
  - void remove()
  - int nextIndex()
  - int previousIndex()

The Comparator Interface

- **TreeSet** store elements in sorted order. It is the comparator that defines precisely what "sorted order" means.
- By default, Java considers "natural ordering" but it can be changed by specifying a **Comparator**.
- **Comparator** is a generic interface with declaration:
  - interface Comparator<T>
    - Here, **T** specifies the type of objects being compared.
- **Comparator** interface defines two methods:
  - int compare(T *object1*, T *object2*)
  - boolean equals(Object *object*)

The Collection Classes

- Main classes of The Collection Framework 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.
AbstractSet	Extends **AbstractCollection** and implements most of the **Set** interface.
LinkedList	Implements a linked list by extending **AbstractSequentialList**.
ArrayList	Implements a dynamic array by extending **AbstractList**.
EnumSet	Extends **AbstractSet** for use with **enum** elements.
PriorityQueue	Extends **AbstractQueue** to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends **AbstractSet**.

The ArrayList Class

- **ArrayList** class extends **AbstractClass** and implements the **List** interface.
- **ArrayList** is a generic class that has the 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.
- Array lists are created with an initial size. When the size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.

The ArrayList Class

- **ArrayList** has the constructors as follows:
  - ArrayList()
  - ArrayList(Collection<? extends E> *c*)
  - ArrayList(int *capacity*)
- Some methods of **ArrayList** are:
  - void add(E *object*)
  - void add(int *index*, E *object*)
  - void remove(E *object*)
  - void remove(int *index*)
  - int size()
  - void ensureCapacity(int *capacity*)
  - void trimToSize()
  - object[] toArray()

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

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

    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);
  }
}
```

The output from this program is shown here:

```
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]
```

The LinkedList Class

- **LinkedList** extends **AbstractSequentialList** and implements the **List**, **Deque**, and **Queue** interface.
- It provides a linked-list data structure.
- **LinkedList** is a generic class with its declaration as:
  - class LinkedList<E>
    - Here, **E** specifies the type of objects that the list will hold.
- **LinkedList** has its constructors as follows:
  - LinkedList()
  - LinkedList(Collection<? extends E> *c*)

The TreeSet Class

- **TreeSet** extends **AbstractSet** and implements the **NavigableSet** interface. It creates a collection that uses tree for storage.
- Objects are stored in sorted, ascending order.
- Access and retrieval times are quite fast for **TreeSet**.
- **TreeSet** is excellent choice when storing large amounts of sorted information that must be found quickly.

The TreeSet Class

- **TreeSet** is a generic class with its declaration as:
  - class TreeSet<E>
    - Here, **E** specifies the type of object that the set will hold.
- **TreeSet** has following constructors:
  - TreeSet()
  - TreeSet(Collection<? extends E> *c*)
  - TreeSet(Comparator<? super E> *comp*)
  - TreeSet(SortedSet <E> *ss*)

The PriorityQueue Class

- **PriorityQueue** extends **AbstractQueue** and implements the **Queue** interface. It creates a queue that is prioritized based on the queue's comparator.
- **PriorityQueue**s are dynamic, growing as necessary.
- **PriorityQueue** is a generic class with its declaration as:
  - class PriorityQueue<E>
    - Here, **E** specifies the type of objects that the queue will hold.
- **PriorityQueue** has its constructors as follows:
  - PriorityQueue()
  - PriorityQueue(int *capacity*)
  - PriorityQueue(int *capacity*, Comparactor<? super E> *comp*)
  - PriorityQueue(Collection<? extends E> *c*)
  - PriorityQueue(PriorityQueue<? extends E> *c*)
  - PriorityQueue(SortedSet<? extends E> *c*)

The EnumSet Class

- **EnumSet** extends **AbstractSet** and implements **Set** interface. It is specially for use with keys of an **enum** type.
- **EnumSet** is a generic class with its declaration as:
  - class EnumSet<E extends Enum<E>>
    - Here, **E** specifies the type of objects that the queue will hold.
    - **E** must extend **Enum<E>**, which enforces the requirement that the elements must be of the specified **enum** type.
- **EnumSet** defines no constructor but it has static methods which create objects.

The Collection Algorithms

- The Collections Framework defines several algorithms that can be applied to collections.
- These algorithms are defined as static methods within the **Collections** class.
- Few of these are:
  - static <T> boolean addAll(Collection<? super T> *c*, T... *elements*)
  - static <T> Queue<T> asLifoQueue(Deque<T> *c*)
  - static <T> int binarySearch(List<? extends T> *list*, T *value*, Comparator<? super T> *c*)
  - static <T> void copy(List<? super T> *list1*, List<? extends T> *list2*)
  - static <T> boolean disjoint(Collection<?> *a*, Collection<?> *b*)

The Collection Algorithms

- static <T> void fill(List<? super T> *list*, T *object*)
- static <T> int frequency(Collection<? > *c*, Object *elements*)
- static <T> boolean replaceAll(List<T> *list*, T *old*, T *new*)
- static void reverse(List<T> *list*)
- static void rotate(List<T> *list*, int *n*)
- static void shuffle(List<T> *list*)
- static <T> void sort(List<T> *list*, Comparator<? super T> *comp*)
- static void swap(List<? > *list*, int *index1*, int *index2*)

The Arrays Class

- The **Arrays** class provides various methods that are useful when working with arrays.
- These methods help bridge gap between collections and arrays.
- **Arrays** class has following methods:
  - static <T> List asList(T... *array*)
  - static int binarySearch(Object *array[]*, Object *value*)
  - static <T> int binarySearch(T *array[]*, T *value*, Comparator<? super T> *c*)
  - static <T> T[] copyOf(T[] *source*, int *len*)
  - static <T> T[] copyOfRange(T[] *source*, int *start*, int *end*)
  - static boolean equals(Object *array1[]*, Object *array2[]*)

The Arrays Class

- static boolean deepEquals(Object *array1[]*, Object *array2[]*)
- static void fill(Object *array[]*, Object *value*)
- static void fill(Object *array[]*, int *start*, int *end*, Object *value*)
- static void sort(Object *array[]*)
- static <T> void sort(T *array[]*, Comparator<? super T> *c*)
- static void sort(Object *array[]*, int *start*, int *end*)
- static <T> void sort(T *array[]*, int *start*, int *end*, Comparator<? super T> *c*)

The Vector Class

- The **Vector** class implements a dynamic array and is similar to **ArrayList**, but is synchronized.
- **Vector** was reengineered to extend **AbstractList** and to implement the **List** and **Iterable** interface.
- Now **Vector** is fully compatible with collections, and have its contents iterated by for-each loop.
- Declaration of **Vector** class is:
  - class Vector<E>
    - Here, **E** specifies the type of elements that will be stored.
- Constructors of **Vector** class are:
  - Vector() : initial size 10
  - Vector(int *size*)
  - Vector(int *size*, int *increment*)
  - Vector(Collection<? extends E> *c*)

The Vector Class

- **Vector** class defines these protected data members:
  - int capacityIncrement()
  - int elementCount()
  - Object[] elementData;
- In addition to methods defined by **List**, **Vector** defines several other methods:
  - void addElement(E *element*)
  - int capacity()
  - void ensureCapacity(int *size*)
  - boolean contains(Object *element*)
  - void copyInto(Object *array[]*)

The Vector Class

- E firstElement()
- E lastElement()
- int indexOf(Object *element*)
- int indexOf(Object *element*, int *start*)
- boolean isEmpty()
- int lastIndexOf(Object *element*)
- int lastIndexOf(Object *element*, int *start*)
- void removeAllElements()
- E elementAt(int *index*)
- void setElementAt(E *element*, int *index*)
- int size()
- void setSize(int *size*)
- void trimToSize()

The Stack Class

- The **Stack** class is a subclass of **Vector** that implements a standard last-in, first-out stack.
- **Stack** only defines the default constructor, which creates an empty stack.
- Declaration of **Stack** class is:
  - class Stack<E>
    - Here, **E** specifies the type of elements stored in stack.
- **Stack** includes all the methods defined by **Vector** and adds several of its own:
  - boolean empty()
  - E peek()
  - E pop()
  - E push(E *element*)
  - int search(Object *element*)

The Dictionary Class

- **Dictionary** is an abstract class that represents a key/value storage repository and operates much like a **Map**.
- Given a key and value, the value can be stored in a **Dictionary** object. After storing this value, it can be retrieved by using its key.
- **Dictionary** is now fully superseded by **Map**.
- Declaration of **Dictionary** class is:
  - class Dictionary<K,V>
    - Here, **K** specifies the type of keys and **V** specifies the type of values.

The Dictionary Class

- **Dictionary** defines following methods:
  - V get(Object *key*)
  - V put(K *key*, V *value*)
  - boolean isEmpty()
  - Enumeration<V> elements()
  - Enumeration<K> keys()
  - V remove(Object *key*)
  - int size()

The StringTokenizer Class

- Processing of text often consists of parsing a formatted input string.
- ***Parsing*** is division of text into a set of discrete parts, or ***tokens***.
- **StringTokenizer** class provides the first step in this parsing process, often called the ***lexer*** (lexical analyzer) or ***scanner***.
- **StringTokenizer** implements the **Enumeration** interface. Therefore, given an input string, its individual tokens can be enumerated.

The StringTokenizer Class

- To use **StringTokenizer** an input string and a string that contains delimiters is specified.
- ***Delimiters*** are characters that separate tokens. Each character in the delimiters string is considered a valid delimiter.
- The default set of delimiters consists of the whitespace characters: space, tab, newline, and carriage return.
- **StringTokenizer** defines following constructors:
  - StringTokenizer(String *str*)
  - StringTokenizer(String *str*)
  - StringTokenizer(String *str*, String *delimiters*, boolean *delimAsToken*)

The StringTokenizer Class

- **StringTokenizer** defines following methods:
  - int countTokens()
  - boolean hasMoreElements()
  - boolean hasMoreTokens()
  - Object nextElement()
  - String nextToken()
  - String nextToken(String *delimiters*)

The BitSet Class

- A **BitSet** class creates a special type of array that holds bit values.
- This array can increase in size as needed which makes it similar to a vector of bits.
- **BitSet** defines following constructors:
  - Bitset()
  - BitSet(int *size*)
    - First constructor creates a default object.
    - Second constructor allows you to specify its initial size (i.e. number of bits it can hold).
    - All bits are initialized to zero.

The BitSet Class

- **BitSet** defines following methods:
  - int cardinality()
  - void and(BitSet *bitSet*)
  - void andNot(BitSet *bitSet*)
  - void clear()
  - void clear(int *index*)
  - void clear(int *startIndex*, int *endIndex*)
  - void flip(int *index*)
  - void flip(int *startIndex*, int *endIndex*)
  - boolean get(int *index*)
  - BitSet get(int *startIndex*, int *endIndex*)
  - boolean intersects(BitSet *bitSet*)

The BitSet Class

- int length()
- int nextClearBit(int *startIndex*)
- int nextSetBit(int *startIndex*)
- void or(BitSet *bitSet*)
- int previousClearBit(int *startIndex*)
- int previousSetBit(int *startIndex*)
- void set(int *index*)
- void set(int *startIndex*, boolean *v*)
- void set(int *startIndex*, int *endIndex*)
- void set(int *startIndex*, int *endIndex*, boolean *v*)
- int size()
- void xor(BitSet *bitSet*)

The Date Class

- A **Date** class encapsulates the current date and time.
- **Date** defines following constructors:
  - Date()
    - Initializes the object with the current date and time..
  - Date(long *milliSeconds*)
    - Accepts the argument which is equal to the number of milliseconds that have elapsed since midnight, January1, 1970.

The Date Class

- **Date** class defines following methods:
  - boolean after(Date *date*)
  - boolean before(Date *date*)
  - int compareTo(Date *date*)
  - boolean equals(Object *date*)
  - long getTime()
  - void setTime(long *time*)
  - String toString()

The Random Class

- The **Random** class is a generator of pseudorandom numbers.
- These numbers are called pseudorandom numbers because they are simply uniformly distributed sequences.
- **Random** defines following constructors:
  - Random()
    - Creates a number generator that uses a reasonably unique seed.
  - Random(long *seed*)
    - Allows you to specify a seed value manually.

The Random Class

- **Random** class defines following methods:
  - boolean nextBoolean()
  - void nextBytes(byte *vals[]*)
  - double nextDouble()
  - float nextFloat()
  - double nextGaussian()
  - int nextInt()
  - long nextLong()
  - Void setSeed(long *newSeed*)