Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Java: Collections framework/5b76d2714227cf4f24a32baa

TUTORIAL

Java: Collections framework

Topics

Although we can use an array to store a group of elements of the same type (either primitives or objects). The array, however, does not support so-called dynamic allocation - it has a fixed length which cannot be changed once allocated. Furthermore, array is a simple linear structure. Many applications may require more complex data structure such as linked list, stack, hash table, sets, or trees. In Java, dynamically allocated data structures (such as ArrayList, LinkedList, Vector, Stack, HashSet, HashMap, Hashtable) are supported in a unified architecture called the Collection Framework, which mandates the common behaviors of all the classes. A collection, as its name implied, is simply an object that holds a collection (or a group, a container) of objects. Each item in a collection is called an element. A framework, by definition, is a set of interfaces that force you to adopt some design practices. A well-designed framework can improve your productivity and provide ease of maintenance.

The collection framework provides a unified interface to store, retrieve and manipulate the elements of a collection, regardless of the underlying and actual implementation. This allows the programmers to program at the interfaces, instead of the actual implementation.

The Java Collection Framework package (java.util) contains:

- A set of interfaces,
- Implementation classes, and
- Algorithms (such as sorting and searching).

Similar Collection Framework is the C++ Standard Template Library (STL).

- Prior to JDK 1.2, Java's data structures consist of array, Vector, and Hashtable that were designed in a non-unified way with inconsistent public interfaces. JDK 1.2 introduced the unified collection framework, and retrofits the legacy classes (Vector and Hashtable) to conform to this unified collection framework.
- JDK 1.5 introduced Generics (which supports passing of types), and many related features (such as auto-boxing and unboxing, enhance for-loop). The collection framework is retrofitted to support generics and takes full advantages of these new features.

Let us write a simple variable list using generics in java collections framework: -

Java

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

public class Main
{
  public static void main(String[] args)
  {
    List<String> lst = new ArrayList<String>();  // Inform compiler about the type
    lst.add("Code");         // compiler checks if argument's type is String
    lst.add("Quotient");
    lst.add("Gopal");
    System.out.println(lst);

    Iterator<String> iter = lst.iterator();   // Iterator of Strings
    while (iter.hasNext()) {
      String str = iter.next();  // compiler inserts downcast operator
      System.out.println(str);
    }

    // Enhanced for-loop (JDK 1.5)
    for (String str : lst)
```

```
        System.out.println(str);
    }
}
```

The Collection Interfaces

The hierarchy of the interfaces (and the commonly-used implementation classes) in the *Collection Framework* is as shown below:

```
Interfaces
                Classes
Collection<E>
        List<E>
{ArrayList, LinkedList, Stack, Vector)
        Set<E>
                SortedSet<E>
                        NavigableSet<E>
(HashSet, LinkedHashSet, TreeSet)
        Queue<E>
                Deque<E>
(PriorityQueue, ArrayDeque, LinkedList)
Map<K, V>
        SortedMap<K, V>
                NavigableMap<K, V>
(HashMap, HashLinkedMap, HashTable, TreeMap)
```

Above the Collection<E> interface there is a Iterable<E> interface to provide the iterator for all collections as defined below.

Iterable<E> Interface

The Iterable<E> interface, which takes a generic type E and read as Iterable of element of type E, declares one abstract method called iterator() to retrieve the Iterator<E> object associated with all the collections. This Iterator object can then be used to transverse through all the elements of the associated collection.

```
Iterator<E> iterator();   // Returns the associated Iterator
instance
                          // that can be used to transverse thru
all the elements of the collection
```

All implementations of the collection (e.g., ArrayList, LinkedList, Vector) must implement this method, which returns an object that implements Iterator interface.

Iterator<E> Interface

The Iterator<E> interface, declares the following three abstract methods:

```
boolean hasNext()  // Returns true if it has more elements
E next()           // Returns the next element of generic type E
void remove()      // Removes the last element returned by the
iterator
```

As seen in the introductory example, you can use a while-loop to iterate through the elements with the `Iterator` as follows:

```
List<String> lst = new ArrayList<String>();
lst.add("Code");
lst.add("Quotient");
lst.add("Gopal");

// Retrieve the Iterator associated with this List via the
iterator() method
Iterator<String> iter = lst.iterator();
// Transverse thru this List via the Iterator
while (iter.hasNext()) {
   // Retrieve each element and process
   String str = iter.next();
   System.out.println(str);
}
```

Collection<E> Interface

The Collection<E>, which takes a generic type E and read as Collection of element of type E, is the root interface of the Collection Framework. It defines the common behaviors expected of all classes, such as how to add or remove an element, via the following abstract methods:

```
// Basic Operations
int size()                        // Returns the number of
elements of this Collection
void clear()                      // Removes all the elements of
```

```
this Collection
boolean isEmpty()                 // Returns true if there is no
element in this Collection
boolean add(E element)            // Ensures that this Collection
contains the given element
boolean remove(Object element)    // Removes the given element,
if present
boolean contains(Object element)  // Returns true if this
Collection contains the given element

// Bulk Operations with another Collection
boolean containsAll(Collection<?> c)       // Collection of any
"unknown" object
boolean addAll(Collection<? extends E> c)  // Collection of E or
its sub-types
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)

// Comparison - Objects that are equal shall have the same
hashCode
boolean equals(Object o)
int hashCode()

// Array Operations
Object[] toArray()       // Convert to an Object array
<T> T[] toArray(T[] a)   // Convert to an array of the given type
T
```

List<E>, Set<E> & Queue<E> interfaces

In practice, we typically program on one of the sub-interfaces of the Collection interface: List<E>, Set<E>, or Queue<E>, which provide further specifications.

- List<E>: models a resizable linear array, which allows indexed access. List can contain duplicate elements. Frequently-used implementations of List include ArrayList, LinkedList, Vector and Stack.
- Set<E>: models a mathematical set, where no duplicate elements are allowed. Frequently-used implementations of Set are HashSet and LinkedHashSet. The sub-interface SortedSet<E> models an ordered and sorted set of elements, implemented by TreeSet.
- Queue<E>: models queues such as First-in-First-out (FIFO) queue and priority queue. It sub-interface Deque<E> models queues that can be operated on both ends. Implementations include PriorityQueue, ArrayDeque and LinkedList.

Map<K,V> Interface:

The interface Map<K,V>, which takes two generic types K and V and read as Map of Key type K and Value type V, is used as a collection of of "key-value pairs". No duplicate key is allowed. Frequently-used implementations include HashMap, Hashtable and LinkedHashMap. Its sub-interface SortedMap<K, V> models an ordered and sorted map, based on its key, implemented in TreeMap.

Take note that Map<K,V> is not a sub-interface of Collection<E>, as it involves a pair of objects for each element.

List<E> Interface and Implementations

In practice, it is more common to program on the one of the sub-interfaces of Collection: List, Set, or Queue, instead of the super-interface Collection. These sub-interfaces further refine the behaviors of the Collection.

A List<E> models a resizable linear array, which supports indexed access. Elements in a list can be retrieved and inserted at a specific index position based on an int index. It can contain duplicate elements. It can contain null elements. You can search a list, iterate through its elements, and perform operations on a selected range of values in the list. Lists are the most commonly-used data structures.

The List<E> interface declares the following abstract methods, in additional to its super-interfaces. Since List has a positional index. Operation such as add(), remove(), set() can be applied to an element at a specified index position.

```
// Operations at a specified index position
void add(int index, E element)    // add
E set(int index, E element)       // replace
E get(int index)                  // retrieve without remove
E remove(int index)               // remove last retrieved
int indexOf(Object obj)
int lastIndexOf(Object obj)
// Operations on a range fromIndex (inclusive) toIndex (exclusive)
List<E> subList(int fromIndex, int toIndex)
......

// Operations inherited from Collection<E>
```

```
int size()
boolean isEmpty()
boolean add(E element)
boolean remove(Object obj)
boolean contains(Object obj)
void clear();
......
```

The abstract superclass AbstractList provides implementation to many of the abstract methods declared in the List, Collector, and Iterable interfaces. However, some methods such as get(int index) remains abstract. These methods will be implemented by the concrete subclasses such as ArrayList and Vector.

ArrayList<E> & Vector<E> - Implementation Classes for List<E>

- ArrayList<E> is the best all-around implementation of the List<E> interface. Many useful methods are already implemented in AbstractList but overridden for efficiency in ArrayList (e.g., add(), remove(), set() etc.).
- Vector<E> is a legacy class (since JDK 1.0), which is retrofitted to conform to the Collection Framework (in JDK 1.2). Vector is a synchronized implementation of the List interface. It also contains additional legacy methods (e.g., addElement(), removeElement(), setElement(), elementAt(), firstElement(), lastElement(), insertElementAt()). There is no reason to use these legacy methods - other than to maintain backward compatibility.

ArrayList is not synchronized. The integrity of ArrayList instances is not guaranteed under multithreading. Instead, it is the programmer's responsibility to ensure synchronization. On the other hand, Vector is synchronized internally.

Stack<E> - Implementation Class for List<E>

Stack<E> is a last-in-first-out queue (LIFO) of elements. Stack extends Vector, which is a synchronized resizable array, with five additional methods:

```
E push(E element)       // pushes the specified element onto the
top of the stack
E pop()                 // removes and returns the element at the
top of the stack
```

```
E peek()                // returns the element at the top of
stack without removing
boolean empty()         // tests if this stack is empty
int search(Object obj)  // returns the distance of the specified
object from the top
                        //  of stack (distance of 1 for TOS), or
-1 if not found
```

LinkedList<E> - Implementation Class for List<E>

- LinkedList<E> is a double-linked list implementation of the List<E> interface, which is efficient for insertion and deletion of elements, in the expense of more complex structure.
- LinkedList<E> also implements Queue<E> and Deque<E> interfaces, and can be processed from both ends of the queue. It can serve as FIFO or LIFO queue.

Converting a List to an Array - toArray()

The super-interface Collection defines a method called toArray() to create an array based on this list. The returned array is free for modification.

```
Object[] toArray()      // Object[] version
<T> T[] toArray(T[] a)  // Generic type version
```

Java

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;
class Main
{
   public static void main(String[] args)
   {
      List<String> cq1 = new ArrayList<String>();
      cq1.add("Code");
      cq1.add("Quotient");
      cq1.add("Gopal");

```

```
        // Use the Object[] version
        Object[] cq2 = cq1.toArray();
        System.out.println(Arrays.toString(cq2));

        String[] cq3 = cq1.toArray(new String[0]);
        cq3[2] = "Girdhar";   // modify the returned array
        System.out.println(Arrays.toString(cq3));
        System.out.println(cq1);
    }
}
```