

Collections

You can create complex data structures (linked lists, queues, stack, trees...) in Java using objects and object references and by using heap memory with the *new* operator.

Java supports something called the **Java Collections Framework**, which provides the programmer with pre-packaged data structures as well as algorithms to work on those data structures.

Here, we will look at **interfaces, implementation classes, algorithms, and iterators**.

The Java Collections Framework provides you with ready components to use. You don't have to reinvent the wheel. The collections are standardized so they can easily be shared; they also encourage reusability. The collections can hold any type of data; what's more, a collection can hold objects/elements of different types.

Collections 2

A *collection* is a data structure (an object) that can hold other objects. The *collection interfaces* define the operations that can be performed on each type of collection . The *collections implementations* implement these operations in particular ways to provide customized solutions.

The most common interfaces in the framework are: *Set*, *List*. They both extend the root *Collection* interface. The *Collection* interface provides basic methods to manipulate collection objects. The framework also includes the *Map* interface for data structures containing both value and keys. Programmers can implement these interfaces specific to their application needs.

As well as the above interfaces, the framework also contains other interfaces useful when working with collection objects. The classes and interfaces of the collections framework are all members of the `java.util` package.

Collections 3

One of the classes in the framework is the class `Arrays` which provides static methods to manipulate arrays. You have seen this class before. The methods of this class are overloaded to work on both arrays of primitive types and objects. Here is a program which demonstrates some of the more common methods of this class: (Only the main method is shown)

```
public static void main(String[] arg){
    int[] a={3,1,2};          int[] b={1,2,3};
    if(Arrays.equals(a,b))
        System.out.println("Equal");
    Arrays.fill(a, 5);
    Arrays.sort(a);
        //return index of 4 in array
    System.out.println(Arrays.binarySearch(a,4));
    for(int i=0; i<3; i++)
        System.out.println(a[i]);
}
```

Collections 4

The Collection interface provides methods (operations) common to all collection objects such as: adding, clearing, comparing, determining a collection's size... There is also a class called Collections which contains static methods to manipulate collections polymorphically.

A *List* is an ordered collection that may contain duplicate values. Like arrays, Lists are zero based. Interface *List* is implemented by three classes in the framework: `ArrayList`, `LinkedList` and `Vector`.

The following program demonstrates the `ArrayList` class:

```
import java.util.*;
public class myArrayList{
    public static void main(String[] arg){
        ArrayList aList=new ArrayList();
        aList.add("Xyz");
    }
}
```

Collections 5

```
aList.add(new Integer(1));
aList.add(new Double(2.5));
aList.add(new Integer(3));
aList.add(2,new Integer(6));
System.out.println(aList.get(1));
System.out.println(aList.size());
System.out.println(aList.indexOf("Xyz"));
Iterator j=aList.iterator();
while(j.hasNext())
    if(j.next() instanceof String)
        j.remove();
System.out.println(aList.size());
for(int i=0; i<aList.size(); i++)
    System.out.println(aList.get(i));
}
}
```

Collections 6

The programs: first creates an `ArrayList` object and then using the method `add`, appends or adds some elements to the list. More specifically, it adds a string, an integer, a double, another double, and one last integer. The last method takes two parameters, the first one specifies the position in the list where the element will be stored. The previous element at the position and all subsequent elements will be shifted to the right.

The `get` method takes the index of an element as a parameter and returns that element. An `IndexOutOfBoundsException` exception will be thrown if you provide an out of range index. The `size` method returns the size of the `ArrayList` object.

The method `indexOf` the index of the first occurrence of the argument in this list; returns -1 if the object is not found.

The next line, creates an iterator object which can be used to iterate through collection objects. The iterator object has only three methods:

Collections 7

`hasNext` checks to see if there are more elements in the list. `Next` returns the next element in the list and `remove` removes the current element reference by the iterator object.

Each `ArrayList` instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an `ArrayList`, its capacity grows automatically. The `ArrayList` class has another constructor which takes a parameter to specify the initial capacity of the list:

```
ArrayList(int initialCapacity)
```

This constructs an empty list with the specified initial capacity.

Notice that the `ArrayList` can hold objects of any type, as you saw in this example. You cannot, however, directly store primitive type data like integers and doubles in the list.

Collections 8

Some useful ArrayList methods are:

- boolean `addAll(Collection c)` //group operation

Appends all of the elements in the specified Collection to the end of this list, in the order that they are returned by the specified Collection's Iterator.

-boolean `contains(Object elem)`

Returns true if this list contains the specified element.

- boolean `isEmpty()`

Tests if this list has no elements.

-Object `remove(int index)`

Removes the element at the specified position in this list.

-Object `set(int index, Object element)`

Replaces the element at the specified position in this list with the specified element.

Collections 9

-Object[] [toArray\(\)](#)

Returns an array containing all of the elements in this list in the correct order.

-void [trimToSize\(\)](#)

Trims the capacity of this ArrayList instance to be the list's current size.

For a complete coverage of the methods of this class, consult the JDK documentation or a textbook.

The next collection implementation we are going to look at is the LinkedList class.

A `LinkedList`, like an `ArrayList` also implements the `List` interface but is more efficient for situations where you would insert/remove elements in the middle of a sequence. Removing and inserting elements in the

Collections 10

middle of an array, Vector or ArrayList are very expensive operations. The LinkedList implementation does not suffer from this drawback. Because its elements/object references are not stored sequentially. This class can also be used to represent different types of queues and stacks.

```
LinkedList staff=new LinkedList();
staff.add(" Xyz ");
staff.add(" Wyz ");
staff.add(" Zyz ");
Staff.add(2," Third ");
Staff.remove(" Wyz ");
iterator it=staff.iterator();
for(int i=0; i<3; i++)
    System.out.println(it.next());
```

This would output: Xyz Third Zyz

Collections 11

But a linked list object in Java has references to both the next element and the previous element. The `Iterator` interface we saw earlier only has a forward reference to the next element. To facilitate bi-directional traversing of linked list, Java provides the `ListIterator` interface.

```
LinkedList staff=new LinkedList();
staff.add(" Xyz ");
staff.add(" Wyz "); staff.add(" Zyz ");
staff.add(2," Third ");
staff.remove(" Wyz ");
ListIterator lt=staff.listIterator();
for(int i=0; i<staff.size(); i++)
    System.out.println(lt.next());
for(int i=0; i<staff.size(); i++)
    System.out.println(lt.previous());
```

The output: Xyz Third Zyz Zyz Third Xyz

Collections 12

You can use the set method to change the value of an element of a list:

```
staff.set(0, "One");
```

Or you can use the listIterator:

```
Object oldValue=it.next();  
it.set(newValue);
```

The following program creates two linked lists, merges them and then removes every second element from the second list. Finally the program removes all elements that exist in the second list from the first list:

```
public class LinkedListTest  
{  
    public static void main(String[] args)  
    {  
        List a = new LinkedList();  
        a.add("ABC"); a.add("DEF"); a.add("GHI");  
        List b = new LinkedList();  
        b.add("JKL"); b.add("MNO"); b.add("PQR"); b.add("STU");
```

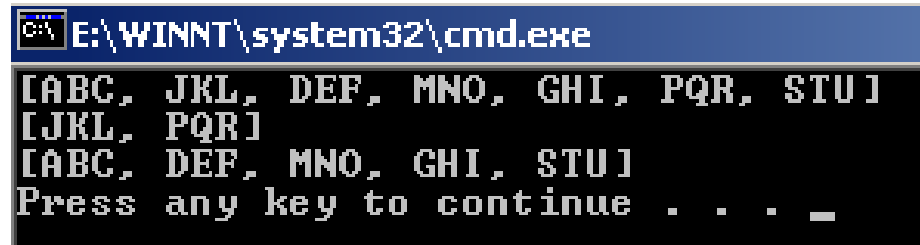
Collections 13

```
    // merge the words from b into a
    ListIterator aIter = a.listIterator();
    Iterator bIter = b.iterator();
    while (bIter.hasNext())
    {   if (aIter.hasNext()) aIter.next();
        aIter.add(bIter.next());
    }
    System.out.println(a);
    // remove every second word from b
    bIter = b.iterator();
    while (bIter.hasNext())
    {   bIter.next(); // skip one element
        if (bIter.hasNext())
        {   bIter.next(); // skip next element
            bIter.remove(); // remove that element
        }
    }
}
```

Collections 14

```
        System.out.println(b);  
        // bulk operation: remove all words in b from a  
        a.removeAll(b);  
        System.out.println(a);  
    }  
}
```

The program outputs:



```
E:\WINNT\system32\cmd.exe  
[ABC, JKL, DEF, MNO, GHI, PQR, STU]  
[JKL, PQR]  
[ABC, DEF, MNO, GHI, STU]  
Press any key to continue . . . _
```

You should use a linked list only when you have to perform many insertions/deletions of elements in the **middle** of a list. In other cases, you should use an `ArrayList`, as it is more efficient.

Collections 15

There are several methods which can be used to view a LinkedList as a queue or stack:

```
void addFirst(Object ob);
void addLast(Object ob);
Object getFirst();
Object getLast();
Object removeFirst();
Object removeLast();

LinkedList queue = ...;           //First-In-First-Out
queue.addFirst(element);
Object object = queue.removeLast();

LinkedList stack = ...;          //First-In-Last-Out
stack.addFirst(element);
Object object = stack.removeFirst();
```