Legacy Classes - Java Collections

Early version of java did not include the Collections framework. It only defined several classes and interfaces that provide methods for storing objects. When Collections framework were added in J2SE 1.2, the original classes were reengineered to support the collection interface. These classes are also known as Legacy classes. All legacy classes and interface were redesign by JDK 5 to support Generics. In general, the legacy classes are supported because there is still some code that uses them.

The following are the legacy classes defined by **java.util** package

1. Dictionary
2. HashTable
3. Properties
4. Stack
5. Vector

There is only one legacy interface called **Enumeration**

NOTE: All the legacy classes are synchronized

Enumeration interface

1. **Enumeration** interface defines method to enumerate(obtain one at a time) through collection of objects.
2. This interface is superseded(replaced) by **Iterator** interface.
3. However, some legacy classes such as **Vector** and **Properties** defines several method in which **Enumeration** interface is used.
4. It specifies the following two methods

```
boolean hasMoreElements() //It returns true while there are still
more elements to extract,

and returns false when all the elements have been enumerated.



Object nextElement() //It returns the next object in the
enumeration i.e. each call to nextElement() method

obtains the next object in the enumeration. It throws
NoSuchElementException when the

enumeration is complete.
```

Copy

Vector class

1. **Vector** is similar to **ArrayList** which represents a dynamic array.
2. There are two differences between **Vector** and **ArrayList**. First, Vector is synchronized while ArrayList is not, and Second, it contains many legacy methods that are not part of the Collections Framework.
3. With the release of JDK 5, Vector also implements Iterable. This means that Vector is fully compatible with collections, and a Vector can have its contents iterated by the for-each loop.
4. Vector class has following four constructor

```
5. Vector() //This creates a default vector, which has an initial
  size of 10.

6.

7. Vector(int size) //This creates a vector whose initial capacity is
  specified by size.

8.

9. Vector(int size, int incr) //This creates a vector whose initial
  capacity is specified by size and whose

10.increment is specified by incr. The increment specifies the number
  of elements to allocate each time

11.when a vector is resized for addition of objects.

12.
```

```
Vector(Collection c) //This creates a vector that contains the
elements of collection c.
```

Copy

Vector defines several legacy methods. Lets see some important legacy methods defined by **Vector** class.

Method	Description
void addElement(E element)	adds element to the Vector
E elementAt(int index)	returns the element at specified index
Enumeration elements()	returns an enumeration of element in vector
E firstElement()	returns first element in the Vector

E lastElement()	returns last element in the Vector
void removeAllElements()	removes all elements of the Vector

Example of Vector

```
import java.util.*;

public class Test

{

  public static void main(String[] args)

  {

     Vector<Integer> ve = new Vector<Integer>();

      ve.add(10);

      ve.add(20);

      ve.add(30);

      ve.add(40);

      ve.add(50);

      ve.add(60);


      Enumeration<Integer> en = ve.elements();


      while(en.hasMoreElements())

      {

           System.out.println(en.nextElement());

      }

    }


}
```

Copy

```
10
```

```
20

30

40

50

60
```

Hashtable class

1. Like HashMap, Hashtable also stores key/value pair. However neither **keys** nor **values** can be **null**.
2. There is one more difference between **HashMap** and **Hashtable** that is Hashtable is synchronized while HashMap is not.
3. Hashtable has following four constructor

```
4. Hashtable() //This is the default constructor. The default size is
   11.

5.

6. Hashtable(int size) //This creates a hash table that has an
   initial size specified by size.

7.

8. Hashtable(int size, float fillratio) //This creates a hash table
   that has an initial size specified by size

9. and a fill ratio specified by fillRatio. This ratio must be
   between 0.0 and 1.0, and it determines how full

10.the hash table can be before it is resized upward. Specifically,
   when the number of elements is greater

11.than the capacity of the hash table multiplied by its fill ratio,
   the hash table is expanded.

12.If you do not specify a fill ratio, then 0.75 is used.

13.

14.Hashtable(Map< ? extends K, ? extends V> m) //This creates a hash
   table that is initialized with the

15.elements in m. The capacity of the hash table is set to twice the
   number of elements in m.
```

```
The default load factor of 0.75 is used.
```

Copy

Example of Hashtable

```
import java.util.*;
```

```
class HashTableDemo

{

  public static void main(String args[])

  {

    Hashtable<String,Integer> ht = new Hashtable<String,Integer>();

    ht.put("a",new Integer(100));

    ht.put("b",new Integer(200));

    ht.put("c",new Integer(300));

    ht.put("d",new Integer(400));


    Set st = ht.entrySet();

    Iterator itr=st.iterator();

    while(itr.hasNext())

    {

      Map.Entry m=(Map.Entry)itr.next();

      System.out.println(itr.getKey()+" "+itr.getValue());

    }

  }

}
```

Copy

```
a 100

b 200

c 300

d 400
```

Difference between HashMap and Hashtable

Hashtable	**HashMap**
Hashtable class is synchronized.	HashMap is not synchronized.

Because of Thread-safe, Hashtable is slower than HashMap	HashMap works faster.
Neither **key** nor **values** can be null	Both **key** and **values** can be null
Order of table remain constant over time.	does not guarantee that order of map will remain constant over time.

Properties class

1. **Properties** class extends **Hashtable** class.
2. It is used to maintain list of value in which both key and value are **String**
3. **Properties** class define two constructor

```
4. Properties() //This creates a Properties object that has no
  default values

5.
```

```
Properties(Properties propdefault) //This creates an object that
uses propdefault for its default values.
```

Copy

6. One advantage of **Properties** over **Hashtable** is that we can specify a default property that will be useful when no value is associated with a certain key.

   Note: In both cases, the property list is empty

7. In Properties class, you can specify a default property that will be returned if no value is associated with a certain key.

Example of Properties class

```
import java.util.*;



public class Test

{



   public static void main(String[] args)
```

```
{

  Properties pr = new Properties();

  pr.put("Java", "James Ghosling");

  pr.put("C++", "Bjarne Stroustrup");

  pr.put("C", "Dennis Ritchie");

  pr.put("C#", "Microsoft Inc.");

  Set< ?> creator = pr.keySet();


  for(Object ob: creator)

  {

       System.out.println(ob+" was created by "+
pr.getProperty((String)ob) );

  }


}

}
```

Copy

```
Java was created by James Ghosling

C++ was created by Bjarne Stroustrup

C was created by Dennis Ritchie

C# was created by Microsoft Inc
```

Stack class

1. Stack class extends Vector.
2. It follows last-in, first-out principle for the stack elements.
3. It defines only one default constructor

   ```
   Stack() //This creates an empty stack
   ```

4. If you want to put an object on the top of the stack, call push() method. If you want to remove and return the top element, call pop() method. An EmptyStackException is thrown if you call pop() method when the invoking stack is empty.

You can use peek() method to return, but not remove, the top object. The empty() method returns true if nothing is on the stack. The search() method determines whether an object exists on the stack and returns the number of pops that are required to bring it to the top of the stack.

Example of Stack

```
import java.util.*;



class StackDemo {

public static void main(String args[]) {

Stack st = new Stack();

st.push(11);

st.push(22);

st.push(33);

st.push(44);

st.push(55);

Enumeration e1 = st.elements();



while(e1.hasMoreElements())

System.out.print(e1.nextElement()+" ");



st.pop();

st.pop();



System.out.println("\nAfter popping out two elements");



Enumeration e2 = st.elements();



while(e2.hasMoreElements())

System.out.print(e2.nextElement()+" ");



}
```

```
}
```

Copy

```
11 22 33 44 55

After popping out two elements

11 22 33
```

Dictionary class

1. Dictionary is an abstract class.
2. It represents a key/value pair and operates much like Map.
3. Although it is not currently deprecated, Dictionary is classified as obsolete, because it is fully superseded by Map class.
