

Vectors and Enumerations

(CISS-111 William Jojo)

Overview

The use of arrays has opened up many possibilities of collecting related data together. In fact, this is just the beginning of collecting data of varying forms into containers of varying shapes and sizes. The array is a useful and productive tool, but it also limited by the fact that we cannot increase the length of an already instantiated array object.

Enter the *vector*. A vector is essentially an array of objects, with the same limitations as any other form of array. The vector, however, takes the array to the next level, that is, it provides an intrinsic mechanism to extend the array when there is no longer any space to hold additional data. There are also mechanisms to shrink the space, insert into the array and remove objects from the middle of the array closing the gap made from the removal.

Vectors

As previously stated the vector allows a more complete, dynamic approach to arrays. This is achievable since, under the hood, we are dealing with an array of objects. Objects imply references, which are essentially addresses which further implies that we would be manipulating lists of references and not the data itself. In other words, if we were to add an item to or remove an item from the vector, we are not rearranging the *data*, only the *references* to the data.

Recall that with arrays we tend to populate the contents from beginning to end. This was, however, not required. If we had provided appropriate program logic, there would be nothing to stop us from inserting into the middle of the array or deleting a value from the beginning or middle. What was lacking in our previous discussion on arrays is that such functionality comes at a price. We would need to push all the values forward at the insertion point or pull the values back to the deletion point.

In addition, we would need to keep track of how many values were used in conjunction with

the length of the array. This would be necessary to know when to enlarge the array. You should be able to make the leap at this point that such an action to enlarge an array that we know cannot be enlarged would require us to review our knowledge of array copy methods, specifically, those involving creating a larger array at the same time or by consciously creating a larger array first then performing a copy.

Table 1 shows some basic members of the `Vector` class. Let us look at how to declare a variable of the `Vector` class:

```
Vector<String> rainbow = new Vector<String>();
```

This statement declares a vector reference variable called `rainbow`. It instantiates a new vector object of 10 elements and assigns it to `rainbow`.

Starting with JDK 5, it is recommended that you include the reference type in angle brackets (`<>`) as is demonstrated above since our intention is to store strings in the vector. If you do not indicate this, you will receive a warning similar to:

```
Note: vector.java uses unchecked or unsafe operations.  
Note: Recompile with -Xlint:unchecked for details.
```

The main concern with this is that the compiler will not check your objects as they are applied to certain methods. As a result it would be possible to add an `Integer` object into the vector by simply writing:

```
rainbow.addElement(56);
```

And as we know in JDK 5 and above, the autoboxing of 56 would turn it into an `Integer` object and would joyfully be added to the vector – not at all what we would like to have happen to our vector or string data. With the reference type included we would then receive a message similar to:

```
addElement(java.lang.String) in java.util.Vector<java.lang.String>  
cannot be applied to (int)
```

The `Vector` class is contained in the `java.util` package and the program in *Example 1* demonstrates some basic actions on vectors.

Instance variables, constructors and methods of the Vector class.

```
protected int elementcount;  
protected Object[] elementData;  
protected int capacityIncrement;
```

```
public Vector()
```

Create a vector of length 10 and no defined increment.

```
public Vector(int length)
```

Create a vector of specified length and no defined increment.

```
public Vector(int length, int increment)
```

Create a vector of specified length and increase length by increment when needed.

```
public void addElement(Object obj)
```

Add an object at the end.

```
public void insertElementAt(Object obj, int index)
```

Add an object at index. Throws `ArrayIndexOutOfBoundsException`.

```
public Object clone()
```

Returns a copy of the vector.

```
public boolean contains(Object obj)
```

Returns true if obj is within the vector, otherwise returns false.

```
public Object elementAt(int index)
```

Returns element at index. Throws `ArrayIndexOutOfBoundsException`.

```
public Object firstElement()
```

Returns the first element of the vector. Throws `NoSuchElementException`.

```
public Object lastElement()
```

Returns the last element of the vector. Throws `NoSuchElementException`.

```
public int indexOf(Object obj)
```

Returns first occurrence of obj or -1 if not found.

```
public int indexOf(Object obj, int index)
```

Returns first occurrence of obj starting at index or -1 if not found.

```
public boolean isEmpty()
```

Returns true if the vector is empty or false if it is not.

```
public int lastIndexOf(Object obj)
```

Returns last occurrence of obj or -1 if not found.

```
public int lastIndexOf(Object obj, int index)
```

Returns last occurrence of obj starting at index or -1 if not found.

<code>public void removeAllElements()</code> Removes all elements from the vector.
<code>public boolean removeElement(Object obj)</code> If exists, removes obj from vector and returns true; otherwise returns false.
<code>public void removeElementAt(int index)</code> If element at index exists, it is removed. Throws <code>ArrayIndexOutOfBoundsException</code> .
<code>public void setElementAt(Object obj, int index)</code> Sets the value of the element at index to obj. Throws <code>ArrayIndexOutOfBoundsException</code> .
<code>public int size()</code> Returns the number of elements contained in the vector.
<code>public String toString()</code> Returns a string representation of the the vector.

Table 1: Various members of the Vector class.

In *Example 1*, we our previous declaration of `rainbow` and begin to add some elements to the vector using the `addElement()` method. The program proceeds to alternate adding and printing the contents of the vector with the implied use of the `toString()` method which is called whenever an object is to be displayed.

The use if `insertElementAt()` and `removeElement()` were also employed to show the flexibility of vectors.

It is important to see how the vector automatically make a hole for values inserted into the middle of the vector and closes the gap left by a value that have been removed. We have even seen in this example adding to the front which pushes all of the existing values down one position.

Note that the `for` loop could also have been written using a `foreach` loop as follows:

```
for (String s : rainbow)
    System.out.println(s);
```

Of course, we lose the detail of indexes printed along with the vector element data values.

```

import java.util.*;

public class colorvector
{
    public static void main(String[] args)
    {
        Vector<String> rainbow = new Vector<String>();
        int x, len;

        rainbow.addElement("Orange");
        rainbow.addElement("Yellow");
        System.out.println(rainbow);

        // forgot about Red!
        rainbow.insertElementAt("Red", 0);
        System.out.println(rainbow);

        rainbow.addElement("Blue");
        rainbow.addElement("Indigo");
        rainbow.addElement("Violet");
        System.out.println(rainbow);

        // forgot about Green!
        rainbow.insertElementAt("Green", 3);
        System.out.println(rainbow);

        len = rainbow.size();
        for ( x = 0; x < len; x++)
            System.out.println("Value at index " + x + " is " +
                               rainbow.elementAt(x));
    }
}

```

Example 1: Program demonstrating Vectors using the colors of the rainbow.

A sample run of the program in Example 1 is shown below:

```
[Orange, Yellow]
[Red, Orange, Yellow]
[Red, Orange, Yellow, Blue, Fuchsia, Indigo, Violet]
[Red, Orange, Yellow, Green, Blue, Indigo, Violet]
Value at index 0 is Red
Value at index 1 is Orange
Value at index 2 is Yellow
Value at index 3 is Green
Value at index 4 is Blue
Value at index 5 is Indigo
Value at index 6 is Violet
```

Enumerations

Recall our set of primitive data types such as `int`, `float`, `char` and `double`. These data types have sets of values and operations that can be applied to them. We have also created more robust data types, called *classes*. These classes used the primitive data types, and/or other classes to make something more efficient and useful than a simple integer.

Java allows us to create our own data types and we can specify the exact set of values that the data type may possess. The data types are called *enumerations* and are identified by the use of the `enum` reserved word. A few examples of enumerations are shown below:

```
public enum Suit {Spades, Hearts, Clubs, Diamonds};

public enum Rank {Two, Three, Four, Five, Six, Seven, Eight,
                 Nine, Ten, Jack, Queen, King, Ace};
```

Each value shown in the braces is an *identifier* and each value is considered an enumeration constant. In addition, each `enum` is a special type of *class* and each enumeration constant of this class are `public static` reference variables to objects of the `enum` type.

In other words, `Spades` is a reference variable of type `Suit` and `King` is a reference variable of type `Rank` since both `Suit` and `Rank` are classes. So we can create declarations like the following:

```

Suit cardSuit;
Rank cardRank;

cardSuit = Suit.Hearts;
cardRank = Rank.King;

System.out.println(cardRank + " of " + cardSuit);

```

Since each enumeration constant is a reference variable and a member of the enumeration class, we can access the values with the dot operator which assigned the object `Hearts` to `cardSuit` and `King` to `cardRank`. The output of the above is:

```
King of Hearts
```

Methods available for the enum type.
<p>public final int compareTo(E obj) Returns a negative, zero or positive value base on the ordinal of this object being less than, equal to or greater than the ordinal of <code>obj</code>, respectively.</p>
<p>public final boolean equals(Object obj) Returns true if <code>obj</code> is equal to this enum constant.</p>
<p>public final int ordinal() Returns the ordinal value of this enum constant.</p>
<p>public final String name() Returns the name of this enum constant as declared.</p>
<p>public String toString() Returns the string representation of an enum constant.</p>
<p>public static E[] values() Automatically generated, implicitly declared method to return an array containing the constants of this enum.</p>
<p>public static E valueOf(String) Automatically generated, implicitly declared method to return the enum constant with the specified name.</p>

Table 2: Enumeration type methods.

Each enumeration constant in the enumeration type has a specific value. This is known as the *ordinal value* and all ordinal values are implicitly `final`. The first enumeration constant is assigned the ordinal value 0, the next is 1 and so on. So for our `Suit` enumeration, `Spades` is 0, `Hearts` is 1, `Clubs` is 2 and `Diamonds` is 3.

Some enumeration methods are shown in *Table 2*. The following nested `foreach` loops shows one method of iterating through the suits and ranks of a deck of cards:

```
for (Suit suit : Suit.values())
    for (Rank rank : Rank.values())
        System.out.println(rank + " of " + suit);
```

The variable `suit` is of `enum` type `Suit` and the `values()` method of the `enum` type is used to get the list of values to use in the `foreach` loop. Similarly, the `rank` variable is of `enum` type `Rank` and the `values()` method is used to feed the inner `foreach` loop.

As we stated earlier, an `enum` type is a special type of class and the `enum` constants are reference variables to the objects of that `enum` type. By stating that this is a class, we can immediately perceive data members, constructors and methods within the `enum` type.

Therefore, let us set up some ground rules for the `enum` type:

- Enumeration types are defined with `enum` and not `class`.
- `enum` types are implicitly `final` since `enum` constants should not be modified.
- `enum` constants are implicitly `static`.
- You may declare reference variables of the `enum` type, but you cannot instantiate objects using the `new` operator.
- Constructors are implicitly `private`.

In *Example 2*, we have an enumeration of United States coins: penny, nickel, dime, quarter and half-dollar. Our main method uses the `Coins` `enum` type as the reference type for `Vector` to create a `pocket` of between 1 and 10 random coins. Each coin in the `pocket` is randomized.

```

import java.util.*;

public class uscoins
{
    public enum Coins
    {
        Penny (0.01), Nickel (0.05), Dime (0.10), Quarter (0.25),
        HalfDollar(0.50);

        private final double value;

        private Coins(double value)
        {
            this.value = value;
        }

        public double getValue()
        {
            return value;
        }
    }

    public static void main(String[] args)
    {
        // Out pocket is a vector rather than an array.
        Vector<Coins> pocket = new Vector<Coins>();
        int x, numCoins, randomCoin;
        // values() returns the enumeration as an array
        Coins[] definedCoins = Coins.values();
        double pocketValue = 0.0;

        // from 1 to 10 random coins in our pocket.
        numCoins = (int)(Math.random() * 10 + 1);
        System.out.printf("There are %d coins in our pocket.%n", numCoins);

        // add a random coin
        for (x = 0; x < numCoins; x++)
        {
            randomCoin = (int)(Math.random() * definedCoins.length);
            pocket.addElement(definedCoins[randomCoin]);
        }

        // Display the list of coins and calculate the pocket value
        System.out.println("Our pocket contains:\n" + pocket);
        for (Coins coin : pocket)
            pocketValue = pocketValue + coin.getValue();

        System.out.printf("%nPocket value is $%.2f.%n%n", pocketValue);
    }
}

```

Example 2: Program demonstrating enum type as special class using U.S. coins.

The enumeration is designed to provide information about 5 possible coins whose names and values are clearly fixed and should never be changed. For example, a nickel is always called a nickel and is always worth 5 cents or .05 of one dollar. The enumeration type called `Coins` defines 5 enumeration constants: `Penny`, `Nickel`, `Dime`, `Quarter` and `HalfDollar`.

Recall from our discussion of user-defined classes that if no constructor is specified, then a default constructor is provided for you and that all instance variables are initialized to their type defaults. Since our `Suit` and `Rank` enumeration types had no such definitions, there were no instance variables to initialize and there was a *default* constructor provided for the special enumeration type.

Each enumeration constant is a call to a constructor (default or user-provided) which creates that constant as a `final` reference variable of an object of that enumeration type.

So `Penny(0.01)` is a call to the `Coin(double value)` constructor which creates `Penny` as a reference variable that points to an object of enum type `Coins` whose instance variable value is 0.01. The same thing is true for `Nickel`, `Dime`, `Quarter` and `HalfDollar`. for their respective values.

The `main()` method creates the `pocket` vector, randomizes the number of coins, then for each coin to be placed in the `pocket`, picks a random coin from the list of possible coins and puts it into `pocket` with `addElement()`.

After the vector is filled, we then display the coins in `pocket`, sum the values of the coins with a `foreach` loop and display the total. A few sample runs are shown below:

```
$ java uscoins
There are 10 coins in our pocket.
Our pocket contains:
[Nickel, Dime, Quarter, Penny, Penny, Penny, Nickel, HalfDollar, HalfDollar,
Dime]

Pocket value is $1.58.

$ java uscoins
There are 5 coins in our pocket.
Our pocket contains:
[HalfDollar, HalfDollar, Dime, HalfDollar, HalfDollar]

Pocket value is $2.10.
```

```
$ java uscoins  
There are 4 coins in our pocket.  
Our pocket contains:  
[Dime, Nickel, Nickel, Quarter]  
  
Pocket value is $0.45.
```

These results are from a Unix command line and the invocation is shown in bold.