1 Introduction to Java

Java, the programming language, was developed by Sun Microsystems™ and released to the world in 1995. It almost instantly became popular, primarily because it was an object oriented language that was similar in syntax (same `if` statements, etc.) to C/C++, had garbage collection (you didn't have to explicitly free memory), was portable, and had a very impressive standard library. In other words, it (arguably) made programming much easier than in C/C++.

In this introduction, we assume that you know some other programming language already, so the focus is on getting you up and running with Java, not on explaining basic programming concepts.

1.1 The "Hello World" program

Every new programming language book begins with the "Hello World" program. Here it is:

Listing 1: First Hello World program.

```
/*
 * Hello World program.
 */
public class HelloWorld {
    public static void main(String[] args){
        System.out.println("Hello World!");
    }
}
```

Hopefully this program is self explanatory. The way you compile this program is by using the `javac` tool that comes with the JDK (ie: Java SDK). You can get the JDK from Sun's website, java.sun.com

You compile the above program, from command line, with:

```
javac HelloWorld.java
```

After compiling, you'll have `HelloWorld.class` file. This is the compiled output (it contains bytecode). You run it (`HelloWorld.class` file) via:

```
java HelloWorld
Hello World!
```

Note that we compiled and ran the program from command line. While learning, it is highly recommended that you use the basic command line. Learning isn't work—productivity and ease are virtues of the work environment. While learning, do not take the easy road—otherwise you'll only learn the easy way.

1.2 Variables

Variables are the key to any program. There are variables called registers inside every CPU (Central Processing Unit). Every program ever written uses some form of variables. This

section is a very simple introduction to what variables are, and how they're used in Java programs.

Usually, a variable implies a memory location to hold one instance of one specific type. What this means is that if there is an integer variable, it can only hold one integer, and if there is a character variable, it can only hold one character.

There can be many different types of variables, including of your own type. A sample declaration for different variable types is given below.

```
boolean t;
byte b;
char c;
int i;
long l;
```

The above seems straight forward, and therefore doesn't need much explanation. Variable t is declared as **boolean** type (ie: true or false), and b as of **byte** type (integer from 0 to 255), etc.

The above variables are what's know as 'primitive types'. Primitive types in Java means that you don't have to create them, they're already available as soon as you declare them. (you'll see what this means when we deal with Objects) It also means that there is usually some hardware equivalent to these variables. For example, an **int** type, can be stored in a 32 bit hardware register.

The other types of variables are instances of classes, or Objects. Java is an Object Oriented language, and everything in it is an object. An object is an instance of a class. Your Java programs consist of classes, in which you manipulate objects, and make the whole program do what you want. This concept will be familiar to you if you've programmed in C++, if not, think of objects as structures. An example of a simple class would be:

Listing 2: A very simple object.

```
public class pSimpleObject{
    int i;
    public pSimpleObject(){
        i = 0;
    }
    public int get(){
        return i;
    }
    public void set(int n){
        i = n;
    }
}
```

As you can see, first we specify that the class is **public**, this means that it can be visible to other objects outside its file. We later say that it's a **class**, and give its name, which in this case is: pSimpleObject. Inside of it, the class contains an integer named i, and three functions. The first function named pSimpleObject(), is the constructor. It is called every time an object is created using this class. The set() and get() functions set and get the value of i respectively. One useful terminology is that functions in objects are not called

'functions', they're called methods. So, to refer to function set(), you'd say "method set()." That's all there is to objects!

The way you declare a variable, or in this case, an object of that class, is:

```
pSimpleObject myObject;
myObject = new pSimpleObject();
```

or

```
pSimpleObject myObject = new pSimpleObject();
```

The first example illustrates how you declare an object named myObject, of class pSimpleObject, and later instantiate it (a process of actual creation, where it calls the object's constructor method). The second approach illustrates that this can be done in one line. The object does not get created when you just declare it, it's only created when you do a **new** on it.

If you're familiar with C/C++, think of objects as pointers. First, you declare it, and then you allocate a new object to that pointer. The only limitation seems to be that you can't do math on these pointers, other than that, they behave as plain and simple C/C++ pointers. (You might want to think of objects as references however.)

1.3 Arrays

One of the most basic data structures, is an array. An array is just a number of items, of same type, stored in linear order, one after another. Arrays have a set limit on their size, they can't grow beyond that limit. Arrays usually tend to be easier to work with and generally more efficient than other structural approaches to organizing data.

For example, lets say you wanted to have 100 numbers. You can always resort to having 100 different variables, but that would be a pain. Instead, you can use the clean notation of an array to create, and later manipulate those 100 numbers. For example, to create an array to hold 100 numbers you would do something like this:

```
int[] myArray;
myArray = new int[100];
```

or

```
int[] myArray = new int[100];
```

or

```
int myArray[] = new int[100];
```

The three notations above do exactly the same thing. The first declares an array, and then it creates an array by doing a **new**. The second example shows that it can all be one in one line. And the third example shows that Java holds the backwards compatibility with C++, where the array declaration is: **int** myArray[]; instead of **int**[] myArray;. To us, these notations are exactly the same. I do however prefer to use the Java one.

Working with arrays is also simple, think of them as just a line of variables, we can address the 5th element (counting from 0, so, it's actually the 6th element) by simply doing:

```
int i = myArray[5];
```

The code above will set integer i to the value of the 5th (counting from 0) element of the array. Similarly, we can set an array value. For example, to set the 50th element (counting from 0), to the value of i we'd do something like:

```
myArray[50] = i;
```

As you can see, arrays are fairly simple. The most convenient way to manipulate arrays is using loops. For example, lets say we wanted to make an array of 100 elements hold numbers from 1 to 100 respectively, and later add seven to every element inside that array (ignore our reasons for it). This can be done very easily using two loops. (actually, it can be done in one loop, but I am trying to separate the problem into two)

```
int i;
for(i=0;i<100;i++)
    myArray[i] = i;
for(i=0;i<100;i++)
    myArray[i] = myArray[i] + 7;
```

In Java, we don't need to remember the size of the array as in C/C++. Here, we have the length variable in every array, and we can check its length whenever we need it. So to print out any array named: myArray, we'd do something like:

```
for(int i = 0;i<myArray.length;i++)
    System.out.println(myArray[i]);
```

This will work, given the objects inside the myArray\lstinline are printable, (have a corresponding toString() method), or are of primitive type.

One of the major limitations on arrays is that they're fixed in size. They can't grow or shrink according to need. If you have an array of 100 max elements, it will not be able to store 101 elements. Similarly, if you have less elements, then the unused space is being wasted (doing nothing).

Java API provides data storage classes, which implement an array for their storage. As an example, take the java.util.Vector class, it can grow, shrink, and do some quite useful things. The way it does it is by reallocating a new array every time you want to do some of these operations, and later copying the old array into the new array. It can be quite fast for small sizes, but when you're talking about several megabyte arrays, and every time you'd like to add one more number (or object) you might need to reallocate the entire array; that can get quite slow. Later, we will look at other data structures where we won't be overly concerned with the amount of the data and how often we need to resize.

Even in simplest situations, arrays are powerful storage constructs.

1.4 Array Stack

The next and more serious data structure we'll examine is the Stack. A stack is a FILO (First In, Last Out), structure. For now, we'll just deal with the array representation of the stack. Knowing that we'll be using an array, we automatically think of the fact that our stack has to have a maximum size.

A stack has only one point where data enters or leaves. We can't insert or remove elements into or from the middle of the stack. As mentioned before, everything in Java is an object, (ie: it's an Object Oriented language), so, lets write a stack object!

Listing 3: Simple array based integer stack.

```
/*
 * simple array based integer stack
 */
public class pArrayStackInt{
    protected int head[];
    protected int pointer;

    public pArrayStackInt(int capacity){
        head = new int[capacity];
        pointer = -1;
    }
    public boolean isEmpty(){
        return pointer == -1;
    }
    public void push(int i){
        if(pointer+1 < head.length)
            head[++pointer] = i;
    }
    public int pop(){
        if(isEmpty())
            return 0;
        return head[pointer--];
    }
}
```

As you can see, that's the stack class. The constructor named pArrayStackInt() accepts an integer. That integer is to initialize the stack to that specific size. If you later try to push() more integers onto the stack than this capacity, it won't work. Nothing is complete without testing, so, lets write a test driver class to test this stack.

Listing 4: Test class for pArrayStackInt.

```
/*
 * class to test pArrayStackInt class.
 */
class pArrayStackIntTest{
    public static void main(String[] args){
        pArrayStackInt s = new pArrayStackInt(10);
        int i,j;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = (int)(Math.random() * 100);
            s.push(j);
            System.out.println("push: " + j);
        }
        while(!s.isEmpty()){
            System.out.println("pop: " + s.pop());
        }
```

```
        System.out.println("Done ;-)");
    }
}
```

The test driver does nothing special, it inserts ten random numbers onto the stack, and then pops them off. Writing to standard output exactly what it's doing. The output gotten from this program is:

```
starting...
push: 33
push: 66
push: 10
push: 94
push: 67
push: 79
push: 48
push: 7
push: 79
push: 32
pop: 32
pop: 79
pop: 7
pop: 48
pop: 79
pop: 67
pop: 94
pop: 10
pop: 66
pop: 33
Done ;-)
```

As you can see, the first numbers to be pushed on, are the last ones to be popped off. A perfect example of a FILO structure. The output also assures us that the stack is working properly.

Now that you've had a chance to look at the source, lets look at it more closely.

The pArrayStackInt class is using an array to store it's data. The data is **int** type (for simplicity). There is a head data member, that's the actual array. Because we're using an array, with limited size, we need to keep track of it's size, so that we don't overflow it; we always look at head.length to check for maximum size.

The second data member is pointer. Pointer, in here, points to the top of the stack. It always has the position which had the last insertion, or -1 if the stack is empty.

The constructor: pArrayStackInt(), accepts the maximum size parameter to set the size of the stack. The rest of the functions is just routine initialization. Notice that pointer is initialized to -1, this makes the next position to be filled in an array, 0.

The isEmpty() function is self explanatory, it returns true if the stack is empty (pointer is -1), and false otherwise. The return type is **boolean**.

The push(**int**) function is fairly easy to understand too. First, it checks to see if the next insertion will not overflow the array. If no danger from overflow, then it inserts. It first

increments the pointer and then inserts into the new location pointed to by the updated pointer. It could easily be modified to actually make the array grow, but then the whole point of "simplicity" of using an array will be lost.

The **int** pop() function is also very simple. First, it checks to see if stack is not empty, if it is empty, it will return 0. In general, this is a really bad error to pop of something from an empty stack. You may want to do something more sensible than simply returning a 0 (an exception throw would not be a bad choice). Then, it returns the value of the array element currently pointed to by pointer, and it decrements the pointer. This way, it is ready for the next push or pop.

I guess that just about covers it. Stack is very simple and is very basic. There are tons of useful algorithms which take advantage of this FILO structure. Now, lets look at alternative implementations.

Given the above, a lot of the C++ people would look at me strangely, and say: "All this trouble for a stack that can only store integers?" Well, they're probably right for the example above. It is too much trouble. The trick I'll illustrate next is what makes Java my favorite Object Oriented language.

In C, we have the **void**∗ type, to make it possible to store "generic" data. In C++, we also have the **void**∗ type, but there, we have very useful templates. Templates is a C++ way to make generic objects, (objects that can be used with any type). This makes quite a lot of sense for a data storage class; why should we care what we're storing?

The way Java implements these kinds of generic classes is by the use of parent classes. In Java, every object is a descendant of the Object class. So, we can just use the Object class in all of our structures, and later cast it to an appropriate type. Next, we'll write an example that uses this technique inside a generic stack.

Listing 5: Simple object based stack.

```
/*
 * simple object based stack
 */
public class pArrayStackObject{
    protected Object head[];
    protected int pointer;

    public pArrayStackObject(int capacity){
        head = new Object[capacity];
        pointer = -1;
    }
    public boolean isEmpty(){
        return pointer == -1;
    }
    public void push(Object i){
        if(pointer+1 < head.length)
            head[++pointer] = i;
    }
    public Object pop(){
        if(isEmpty())
```

```
            return null;
        return head[pointer--];
    }
}
```

The above is very similar to the **int** only version, the only things that changed are the **int** to Object. This stack, allows the push() and pop() of any Object. Lets convert our old test driver to accommodate this new stack. The new test module will be inserting java.lang.Integer objects (not int; not primitive type).

Listing 6: Test driver for object based stack.

```
/*
 * test driver class for object stack.
 */
class pArrayStackObjectTest{
    public static void main(String[] args){
        pArrayStackObject s = new pArrayStackObject(10);
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = new Integer((int)(Math.random() * 100));
            s.push(j);
            System.out.println("push: " + j);
        }
        while(!s.isEmpty()){
            System.out.println("pop: " + ((Integer)s.pop()));
        }
        System.out.println("Done ;-)");
    }
}
```

And for the sake of being complete, we'll include the output. Notice that here, we're not inserting elements of **int** type, we're inserting elements of java.lang.Integer type. This means, that we can insert any Object.

```
starting...
push: 45
push: 7
push: 33
push: 95
push: 28
push: 98
push: 87
push: 99
push: 66
push: 40
pop: 40
pop: 66
```

```
pop: 99
pop: 87
pop: 98
pop: 28
pop: 95
pop: 33
pop: 7
pop: 45
Done ;-)
```

That just about covers stacks. The main idea you should learn from this section is that a stack is a FILO data structure. After this section, non of the data structures will be working with primitive types, and everything will be done solely with objects. (now that you know how it's done...)

And now, onto the array relative of Stack, the Queue.

1.5 Array Queues

A queue is a FIFO (First In, First Out) structure. Anything that's inserted first, will be the first to leave (kind of like the real world queues.) This is totally the opposite of what a stack is. Although that is true, the queue implementation is quite similar to the stack one. It also involves pointers to specific places inside the array.

With a queue, we need to maintain two pointers, the start and the end. We'll determine when the queue is empty if start and end point to the same element. To determine if the queue is full (since it's an array), we'll have a boolean variable named full. To insert, we'll add one to the start, and mod (the %, mod operator) with the size of the array. To remove, we'll add one to the end, and mod (the %, mod operator) with the size of the array. Simple? Well, lets write it.

Listing 7: Array based queue.

```
/*
 * array based queue.
 */
public class pArrayQueue{
    protected Object[] array;
    protected int start,end;
    protected boolean full;

    public pArrayQueue(int maxsize){
        array = new Object[maxsize];
        start = end = 0;
        full = false;
    }

    public boolean isEmpty(){
        return ((start == end) && !full);
    }
```

```
    public void insert(Object o){
        if(!full)
            array[start = (++start % array.length)] = o;
        if(start == end)
            full = true;
    }

    public Object remove(){
        if(full)
            full = false;
        else if(isEmpty())
            return null;
        return array[end = (++end % array.length)];
    }
}
```

That's the queue class. In it, we have four variables, the array, the start and end, and a **boolean** full. The constructor pArrayQueue(**int** maxsize) initializes the queue, and allocates an array for data storage. The isEmpty() method is self explanatory, it checks to see if start and end are equal; this can only be in two situations: when the queue is empty, and when the queue is full. It later checks the full variable and returns whether this queue is empty or not.

The insert(Object) method, accepts an Object as a parameter, checks whether the queue is not full, and inserts it. The insert works by adding one to start, and doing a mod with array.length (the size of the array), the resulting location is set to the incoming object. We later check to see if this insertion caused the queue to become full, if yes, we note this by setting the full variable to true.

The Object remove() method, doesn't accept any parameters, and returns an Object. It first checks to see if the queue is full, if it is, it sets full to false (since it will not be full after this removal). If it's not full, it checks if the queue is empty, by calling isEmpty(). If it is, the method returns a **null**, indicating that there's been an error. This is usually a pretty bad bug inside a program, for it to try to remove something from an empty queue, so, you might want to do something more drastic in such a situation (like an exception throw). The method continues by removing the end object from the queue. The removal is done in the same way insertion was done. By adding one to the end, and later mod it with array.length (array size), and that position is returned.

There are other implementations of the same thing, a little re-arrangement can make several **if** (...) statements disappear. The reason it's like this is because it's pretty easy to think of it. Upon insertion, you add one to start and mod, and upon removal, you add one to end and mod. Easy?

Now that we know how it works, lets actually test it, with modified test driver from the stack example, so, here it comes:

Listing 8: Test driver for array based queue.

```
/*
 * test driver for pArrayQueue
```

```
 */
class pArrayQueueTest{
    public static void main(String[] args){
        pArrayQueue q = new pArrayQueue(10);
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = new Integer((int)(Math.random() * 100));
            q.insert(j);
            System.out.println("insert: " + j);
        }
        while(!q.isEmpty()){
            System.out.println("remove: " + ((Integer)q.remove()));
        }
        System.out.println("Done ;-)");
    }
}
```

As you can see, it inserts ten random java.lang.Integer Objects onto the queue, and later prints them out. The output from the program follows:

```
starting...
insert: 3
insert: 70
insert: 5
insert: 17
insert: 26
insert: 79
insert: 12
insert: 44
insert: 25
insert: 27
remove: 3
remove: 70
remove: 5
remove: 17
remove: 26
remove: 79
remove: 12
remove: 44
remove: 25
remove: 27
Done ;-)
```

Compare this output to the one from stack. It's almost completely different.That's it for this array implementation of this FIFO data structure.

1.6 Array Lists

The next step up in complexity is a list. Most folks prefer to think of a list as a linked list (and we'll go over that later), but what most people miss, is that lists can also be efficiently implemented using arrays. An abstract list has no particular structure; it just has to allow for the insertion and removal of objects from both ends, and some way of looking at the middle elements.Technically, this is a 'deque' data structure.

A list is kind of a stack combined with a queue; with additional feature of looking at the middle elements. Preferably, a list should also contain the current number of elements. Well, lets not just talk about a list, but write one!

Listing 9: Array based list.

```
/*
 * array based list
 */
public class pArrayList{
    protected Object[] array;
    protected int start,end,number;

    public pArrayList(int maxsize){
        array = new Object[maxsize];
        start = end = number = 0;
    }
    public boolean isEmpty(){
        return number == 0;
    }
    public boolean isFull(){
        return number >= array.length;
    }
    public int size(){
        return number;
    }
    public void insert(Object o){
        if(number < array.length){
            array[start = (++start % array.length)] = o;
            number++;
        }
    }
    public void insertEnd(Object o){
        if(number < array.length){
            array[end] = o;
            end = (--end + array.length) % array.length;
            number++;
        }
    }
    public Object remove(){
        if(isEmpty())
            return null;
```

```
        number--;
        int i = start;
        start = (--start + array.length) % array.length;
        return array[i];
    }
    public Object removeEnd(){
        if(isEmpty())
            return null;
        number--;
        return array[end = (++end % array.length)];
    }
    public Object peek(int n){
        if(n >= number)
            return null;
        return array[(end + 1 + n) % array.length];
    }
}
```

The class contains four data elements: array, start, end, and number. The number is the number of elements inside the array. The start is the starting pointer, and the end is the ending pointer inside the array (kind of like the queue design).

The constructor, pArrayList(), and methods isEmpty(), isFull(), and size(), are pretty much self explanatory. The insert() method works exactly the same way as an equivalent queue method. It just increments the start pointer, does a mod (the % symbol), and inserts into the resulting position.

The insertEnd(Object) method, first checks that there is enough space inside the array. It then inserts the element into the end location. The next trick is to decrement the end pointer, add the array.length, and do a mod with array.length. This had the affect of moving the end pointer backwards (as if we had inserted something at the end).

The Object remove() method works on a very similar principle. First, it checks to see if there are elements to remove, if not, it simply returns a **null** (no Object). It then decrements number. We're keeping track of this number inside all insertion and removal methods, so that it always contains the current number of elements. We then create a temporary variable to hold the current position of the start pointer. After that, we update the start pointer by first decrementing it, adding array.length to it, and doing a mod with array.length. This gives the appearance of removing an element from the front of the list. We later return the position inside the array, which we've saved earlier inside that temporary variable i.

The Object removeEnd() works similar to the insert() method. It checks to see if there are elements to remove by calling isEmpty() method, if there aren't, it returns **null**. It then handles the number (number of elements) business, and proceeds with updating the end pointer. It first increments the end pointer, and then does a mod with array.length, and returns the resulting position. Simple?

This next Object peek(**int** n) method is the most tricky one. It accepts an integer, and we need to return the number which this integer is pointing to. This would be no problem if we were using an array that started at 0, but we're using our own implementation, and the list doesn't necessarily start at array position 0. We start this by checking if the parameter

n is not greater than the number of elements, if it is, we return **null** (since we don't want to go past the bounds of the array). What we do next is add n (the requested number) to an incremented end pointer, and do a mod array.length. This way, it appears as if this function is referencing the array from 0 (while the actual start is the incremented end pointer).

Note that the above is a bit confusing, since the start and end of the list seem to be reversed (ie: we're counting elements from end).

As was said earlier, the code you write is useless, unless it's working, so, lets write a test driver to test the list class. For the test driver, we convert the Queue test code (while adding some new checks):

Listing 10: Test code for array based list.

```
class pArrayListTest{
    public static void main(String[] args){
        pArrayList l = new pArrayList(10);
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<5;i++){
            j = new Integer((int)(Math.random() * 100));
            l.insert(j);
            System.out.println("insert: " + j);
        }
        while(!l.isFull()){
            j = new Integer((int)(Math.random() * 100));
            l.insertEnd(j);
            System.out.println("insertEnd: " + j);
        }
        for(i=0;i<l.size();i++)
            System.out.println("peek "+i+": "+l.peek(i));
        for(i=0;i<5;i++)
            System.out.println("remove: " + ((Integer)l.remove()));
        while(!l.isEmpty())
            System.out.println("removeEnd: " + ((Integer)l.removeEnd()));
        System.out.println("Done ;-)");
    }
}
```

The test driver is nothing special, it inserts (in front) five random numbers, and the rest into the back (also five). It then prints out the entire list by calling peek() inside a for loop. It then continues with the removal (from front) of five numbers, and later removing the rest (also five). At the end, the program prints "Done" with a cute smiley face ;-)

The output from this test driver is given below. You should examine it thoroughly, and make sure you understand what's going on inside this data structure.

```
starting...
insert: 14
insert: 72
insert: 71
```

```
insert: 11
insert: 27
insertEnd: 28
insertEnd: 67
insertEnd: 36
insertEnd: 19
insertEnd: 45
peek 0: 45
peek 1: 19
peek 2: 36
peek 3: 67
peek 4: 28
peek 5: 14
peek 6: 72
peek 7: 71
peek 8: 11
peek 9: 27
remove: 27
remove: 11
remove: 71
remove: 72
remove: 14
removeEnd: 45
removeEnd: 19
removeEnd: 36
removeEnd: 67
removeEnd: 28
Done ;-)
```

If you really understand everything up to this point, there is nothing new anybody can teach you about arrays (since you know all the basics). There are however public tools available to simplify your life. Some are good, some are bad, but one that definitely deserves to have a look at is the java.util.Vector class; and that's what the next section is about!

1.7 The Vector

The java.util.Vector class is provided by the Java API, and is one of the most useful array based data storage classes you'll encounter in the Java API. Conceptually, a 'vector', is a growing array; as more and more elements are added onto it, the array grows. There is also a possibility of making the array smaller.

But wait a minute, all this time we've been led to believe that arrays can't grow or shrink, and it seems Java API has done it. Not quite. The java.util.Vector class doesn't exactly 'grow', or 'shrink'. When it needs to do these operations, it simply allocates a new array (of appropriate size), and copies the contents of the old array into the new array. Thus, giving the impression that the array has changed size. Don't you just love encapsulation?

All these memory operations can get quite expensive if a Vector is used in a wrong way. Since a Vector has a similar architecture to the array stack we've designed earlier, the best

and fastest way to use a Vector is to do stack operations. Usually, in programs, we need a general data storage class, and don't really care about the order in which things are stored or retrieved; that's where java.util.Vector comes in very useful.

Using a Vector to simulate a queue is very expensive, since every time you insert or remove, the entire array has to be copied (not necessarily reallocated but still involves lots of useless work).

Vector allows us to view its insides using an Enumerator; a class to go through objects. A sample program that uses java.util.Vector for its storage follows.

Listing 11: Test code for java.util.Vector

```
import java.util.*;

class pVectorTest{
    public static void main(String[] args){
        Vector v = new Vector(15);
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = new Integer((int)(Math.random() * 100));
            v.addElement(j);
            System.out.println("addElement: " + j);
        }
        System.out.println("size: "+v.size());
        System.out.println("capacity: "+v.capacity());

        Enumeration e = v.elements();
        while(e.hasMoreElements())
            System.out.println("e: "+(Integer)e.nextElement());

        System.out.println("Done ;-)");
    }
}
```

The example above should be self explanatory (if you paid attention during test programs for the previous data structures). The main key difference is that this one doesn't actually remove objects at the end; we just leave them inside. Removal can be accomplished very easily, and if you'll be doing anything cool with the class, you'll sure to look up the API specs.

Printing is accomplished using an Enumerator; which we use to march through every element printing as we move along. We could also have done the same by doing a for loop, going from 0 to v.size(), doing a v.elementAt(**int**) every time through the loop. The output from the above program follows:

```
starting...
addElement: 6
addElement: 39
addElement: 74
```

```
addElement: 66
addElement: 13
addElement: 17
addElement: 42
addElement: 77
addElement: 49
addElement: 52
size: 10
capacity: 15
e: 6
e: 39
e: 74
e: 66
e: 13
e: 17
e: 42
e: 77
e: 49
e: 52
Done ;-)
```

You should notice that when we print the size and capacity, they're different. The size is the current number of elements inside the Vector, and the capacity, is the maximum possible without reallocation.

A trick you can try yourself when playing with the Vector is to have Vectors of Vectors (since Vector is also an Object, there shouldn't be any problems of doing it). Constructs like that can lead to some interesting data structures, and even more confusion. Just try inserting a Vector into a Vector ;-)

That covers the Vector class. If you need to know more about it, you're welcome to read the API specs for it. You're also encouraged to look at java.util.Vector source code, and see for yourself what's going on inside that incredibly simple structure.