CS 241 — Introduction to Problem Solving and Programming

Applied Topics

Toward a better array, part I:
Vectors

April 15, 2005

Arrays

What are arrays good for?

What's bad about arrays (even when they are the "right tool" for the job)?

Arrays

Use arrays when your data

- Is uniform
- Is sequenced
- Needs to be treated uniformly
- Needs to be iterated (looped) over
- Has a size unknown until runtime

Problems with arrays

- Arrays cannot grow
- You cannot add to the middle of an array
- The length is not necessarily equal to the number of legitimate elements
- You must do everything (eg, finding a specific element) yourself

Arrays

Solution:

Write your own class that works like an array (and is essentially a wrapper for an array) but allows for shrinking and growing, inserting and deleting, etc.

Write this once for all, and use it forever, never needing to use an array again.

UberArray

Possible interface:

- `setElementAt(index, value)`
- `getElementAt(index)`
- `insertElementAt(index, value)`
- `removeElementAt(index)`
- `addElement(value)`

UberArray

```
public class UberArray {

    /**
     * An array to hold the elements of the "uberarray"
     */
    private int[] internal;

    /**
     * Constructor. Initially this uberarray is empty; internally, use
     * an array of length zero.
     */
    public UberArray() {
        internal = new int[0];
    }
```

UberArray

```
/**
 * Set the value of a certain position.
 * Throw an exception if the index is negative; grow if it is
 * beyond the current bounds; set the appropriate array value.
 * @param index The position to set
 * @param value The value to set in that position
 */
public void setElementAt(int index, int value) {
    if (index < 0)
        throw new ArrayIndexOutOfBoundsException("Negative index");
    if (index >= internal.length)
        grow(index + 1);
    internal[index] = value;
}
```

UberArray

```
/**
 * Read the value at a certain position.
 * Throw an exception if the index is out of bounds;
 * otherwise, read from internal.
 * @param index The position to read
 * @return The value at that position
 */
public int getElementAt(int index) {
    if (index < 0 || index >= internal.length)
        throw new ArrayIndexOutOfBoundsException("No element at "
                                                 + index);
    return internal[index];
}
```

UberArray

```
/**
 * Insert an element at a position, shifting elements in greater
 * positions by one position.
 * Grow one position; shift everything with position greater than
 * index; store value at index.
 * @param index The position where to insert
 * @param value The value to insert
 */
public void insertElementAt(int index, int value) {
    grow(internal.length + 1);
    for (int i = internal.length - 1; i >= index; i--)
        internal[i] = internal[i - 1];
    internal[index] = value;
}
```

UberArray

```
/**
 * Insert an element at the back.
 * @param value The value to insert
 */
public void addElement(int value) {
    setElementAt(internal.length, value);
}
```

UberArray

```
    /**
     * Increase the structure by one unit.
     * Create a new array, copy old values into new, set
     * internal to the new array.
     * @param newSize The size to grow to
     */
    private void grow(int newSize) {
        int[] newInternal = new int[newSize];
        for (int i = 0; i < internal.length; i++)
            newInternal[i] = internal[i];
        internal = newInternal;
    }
}
```

UberArray

Drawbacks?

UberArray

Drawbacks:

- There is still a lot left to do (for example, searching)
- We need such a class for any type we may want to store (work around for reference types: make one that works just on objects)
- We don't have to write a class like this. One already exists.

Vector

Java comes with a class called `Vector`, which acts like a growable array of objects.

A `Vector` contains an array, maintains a range of legitimate indices (where real elements have been stored), and grows as necessary.

Important concepts: a `Vector` has both a size, the number of real elements it contains, and a capacity, the number of elements it could contain without growing.

Except for optimization purposes, you can ignore capacity.

Vector constructors

```
Vector()

Vector(int initialCapacity)
```

Vector methods

Array-like methods:

`Object elementAt(int index)`

`void setElementAt(int index)` *(Differs from the one we wrote— will not make the Vector grow)*

Vector methods

Specialized element retrieval methods

```
Object firstElement()
```

```
Object lastElement()
```

Vector methods

Growing and shrinking methods

```
void addElement(Object obj)

void insertElementAt(Object obj, int index)

void removeElementAt(int index)
```

Vector methods

Size/capacity management methods

```
int capacity()

boolean isEmpty()

int size()

void trimToSize()
```

Vector methods

Searching methods

```
boolean contains(Object elem)

int indexOf(Object elem)

int indexOf(Object elem, int index)

int lastIndexOf(Object elem)

int lastIndexOf(Object elem, int index)

boolean removeElement(Object obj)
```

Vector methods

Other methods

```
void clear()

Object clone()

void copyInt(Object[] anArray)

Object[] toArray()

String toString()
```

Using Vectors

Let's try using Vectors on a simple example in place of arrays. One problem: simple examples are on primitive types; Vectors can use only objects.

Solution: Use wrapper classes

Java provides classes to contain primitive values so they can be treated as objects.

`Double`, `Integer`, `Character` . . .

Wrapper classes

Here's all you need to know about `Double`:

Constructor:
`Double(double value)`

Getter method:
`double doubleValue()`

Averager

```
import java.util.Vector;

public class Averager {
    public static void main(String[] args) {

        // Collection to hold the inputted values
        Vector values = new Vector();
```

Averager

```
import java.util.Vector;

public class Averager {
    public static void main(String[] args) {

        // Collection to hold the inputted values
        Vector values = new Vector();
```

Gives directions to the compiler where to find the `Vector` class. (Use as incantation?)

Averager

```
import java.util.Vector;

public class Averager {
    public static void main(String[] args) {

        // Collection to hold the inputted values
        Vector values = new Vector();
```

Creating a `Vector`, instead of `new double[..]`. We don't know the size and don't need to know.

Averager

```
do {
    // The current value the user inputs
    double current =
        DocsIO.readdouble("Next input-> ");
    // Wrapper so that we can treat double as an object
    Double currentObj = new Double(current);
    values.addElement(currentObj);
} while(DocsIO.readchar("Another (y/n)? ") == 'y');
```

Averager

```
do {
    // The current value the user inputs
    double current =
        DocsIO.readdouble("Next input-> ");
    // Wrapper so that we can treat double as an object
    Double currentObj = new Double(current);
    values.addElement(currentObj);
} while(DocsIO.readchar("Another (y/n)? ") == 'y');
```

Wrap the value in a Double, since Vectors can store only objects.

Averager

```
do {
    // The current value the user inputs
    double current =
        DocsIO.readdouble("Next input-> ");
    // Wrapper so that we can treat double as an object
    Double currentObj = new Double(current);
    values.addElement(currentObj);
} while(DocsIO.readchar("Another (y/n)? ") == 'y');
```

Adding the next element; notice we don't need an index.

Averager

```
double sum = 0;          // running total sum
double product = 1;     // running total product
for (int i = 0; i < values.size(); i++) {
    // current value we're working on
    double current = ((Double) values.elementAt(i)).doubleValue();
    sum += current;
    product *= current;
}
```

Averager

```
double sum = 0;          // running total sum
double product = 1;     // running total product
for (int i = 0; i < values.size(); i++) {
    // current value we're working on
    double current = ((Double) values.elementAt(i)).doubleValue();
    sum += current;
    product *= current;
}
```

Loop header as usual. Note `values.size()` instead of `values.length`.

Averager

```
double sum = 0;          // running total sum
double product = 1;     // running total product
for (int i = 0; i < values.size(); i++) {
    // current value we're working on
    double current = ((Double) values.elementAt(i)).doubleValue();
    sum += current;
    product *= current;
}
```

Since whatever comes out of a `Vector` is known only to be an `Object`, we must cast it to what we know it is. Note use for casting.

Averager

```
double sum = 0;          // running total sum
double product = 1;     // running total product
for (int i = 0; i < values.size(); i++) {
    // current value we're working on
    double current = ((Double) values.elementAt(i)).doubleValue();
    sum += current;
    product *= current;
}
```

Extract the primitive double from the wrapper.

Averager

```
        // arithmetic mean
        double arithMean = sum / values.size();
        // geometric mean
        double geoMean = Math.pow(product, 1.0  / values.size());
        System.out.println("Arithmetic mean: " + arithMean);
        System.out.println("Geometric mean: " + geoMean);
    }
}
```

Drawbacks

So, why ever use arrays again?

Drawbacks

So, why ever use arrays again?

- Arrays are simple
- `Vectors` take up more memory
- Storing primitives in a `Vector` is a hassle
- Even though `Vectors` can do it, inserting and removing is grossly inefficient; if you have to do it a lot, don't use a `Vector` (use a linked list. . . Monday's topic)
- Very often the size of data is known when you need to create the array or `Vector`

Summary

- Uses of arrays
- Limitations of arrays
- `Vector`
- Size versus capacity
- `Vector` methods (You do not have to memorize any, but be familiar enough with them that you can use a summary like the handout quickly; know how all work)
- Wrapper classes (Understand well enough to follow examples)
- Use for casting with `Vector`
- Drawbacks to `Vector`