<< Lecture 9 >>

Preamble

- **Homework due this Wednesday**
- Problem solving—preamble to **polymorphism**
    - 3 ways to solve the array concatenation problem
        - Direct solution
        - Alternative solution
        - **Data structure solution**
            - JukeBox class—encapsulating complexity

The problem

The problem we studied today was the one of creating a new array that represented the concatenation of two previous arrays. Namely, if we had two arrays:

A = { A[0], A[1], A[2], A[3], … , A[n-1] },

and

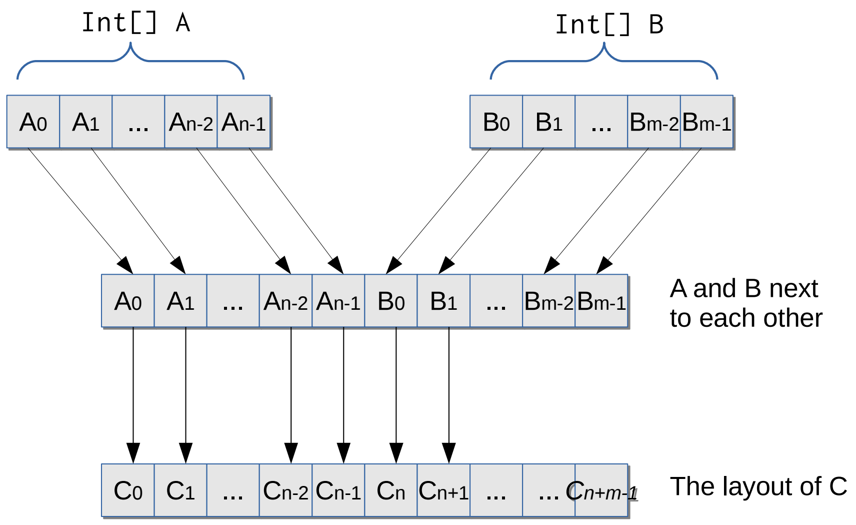
B = { B[0], B[1], B[2], … , B[m-1] },

then, we want to return a new array C first containing the elements of A and then the elements of B, in the original order, i.e.,

C = { A[0], A[1], … , A[n-1] , B[0], B[1], … , B[m-1] }.

Above, n and m are just A.length and B.length, respectively.

More visually, we are creating C from A and B as follows:


Int[] A
Int[] B
A0 A1 ... An-2 An-1
B0 B1 ... Bm-2 Bm-1
A0 A1 ... An-2 An-1 B0 B1 ... Bm-2 Bm-1
A and B next to each other
C0 C1 ... Cn-2 Cn-1 Cn Cn+1 ... ... Cn+m-1
The layout of C

Observe that the i-th component of A corresponds to the i-th component of C. But this does not hold with B. The 0-th component of B occupies the n-th component of C, the 1st component of B occupies the (n+1)-th component of C, and so forth. In general, the i-th component of B occupies the (n+i)-th component of C. Naturally, the length of C is n+m.

Direct solution

The direct solution corresponds to the more intuitive and widely used one: the imperative (procedural) one. This is the solution that you would see in code everywhere.

```
/**
 * Method acDirect takes to integer arrays and returns
 * a new array containing the elements of a and then b
 * in their original order. It performs a concatenation.
 */
public static int[] acDirect(int[] A, int[] B) {
    int[] result = new int[A.length + B.length];
    // copy the elements of A
    for (int i = 0; i < A.length; i = i + 1) {
        result[i] = A[i];
    }
    // copy the elements of B
    for (int i = 0; i < B.length; i = i + 1) {
        result[a.length + i] = B[i];
    }
    return result;
}
```

We directly copy the i-th component of A in the i-th component of C and then we copy the i-th component of B in the (A.length+i)-th component of C.

Alternative solution

We could reflect about the way we solve this problem. For example, we could think about it like this: for each element in A, we want to copy the element *after* the last used position in C, and we want to do the same with each element for B. We want to fill C up sequentially with the elements from A and then B.

```
/**
 * Method acShort takes to integer arrays and returns
 * a new array containing the elements of a and then b
 * in their original order. It performs a concatenation.
 * This method's behavior is identical to acDirect's.
 */
public static int[] acShort(int[] A, int[] B) {
     int[] result = new int[A.length + B.length];
     int i = 0;
     for (int x : A) {  // first, put the elements from A
         result[i] = x;
         i = i + 1;
     }
     for (int x : B) {  // second, put the elements from B
         result[i] = x;
         i = i + 1;
     }
     return result;
}
```

We can shrink the code even further. We notice that we use index i and we then increment it by one. Doesn't this sound much like *i++*?

```
/**
 * Method acShort takes to integer arrays and returns
 * a new array containing the elements of a and then b
 * in their original order. It performs a concatenation.
 * This method's behavior is identical to acDirect's.
 */
public static int[] acShort(int[] A, int[] B) {
    int[] result = new int[A.length + B.length];
    int i = 0;
    // let us put the elements of A and B in result
    for (int x : A) result[i++] = x;
    for (int x : B) result[i++] = x;
    return result;
}
```

By the way, I am using the alternative notation of for, the one that automatically traverses the array.

The way of Data Structures

By giving protagonism to the object where we will be storing data, we are one step away from creating an object that effectively does the work for us. Creating such object that handles the processing of data **is the data structures' way.** The course will be heavily centered on defining and analyzing data structures (mathematically even) in the very near future.

In the style of data structures, I propose the following solution:

- To create an object that contains the result array,
- that keeps track of the last index used in the array,
- so to put the elements of the array continuing after the last index used.

In a way, you could say that I want to *stack* the elements of the supplied arrays onto the result array. (We will see that a stack is a well-defined technical concept in computer science, and it is one the data structures we will study.)

```
/**
 * The ArrayClassContainer class is an object that contains an integer
 * array into which it copies information from other arrays.
 */

class ArrayClassContainer{

    // fields: the result and the current index

    int[] result;
    int i = 0;


    // constructor

    public ArrayClassContainer(int size) {
        result = new int[size];
        // i = 0; this is an alternative
    }
    // this method copies the content of anArray into result, sequentially
    // after the last elements copies

    public void putArray(int[] anArray) {
        for(int x: anArray) {
            result[ i++ ] = x;
            // if index i becomes too big, we restart it
            if (i >= result.length) i = 0;
        }
    }

    // it is in the culture of Java that, if you want to see the contents
    // of a field, you “need” to create a method that does so

    public int[] getArray() {
        return result;
    }

}
```

Observe that the implementation of the above class does not make assumptions on the number of arrays being copied into result; we could copy as many arrays as we wanted to. (Also note that, if we copy too much information into result, we will go back to writing at the beginning of the array.)

With above class, we can now define another array concatenation method:

```
/**
 * Same as acDirect and acShort, but by using the newly defined class
 */

public static int[] acClass(int[] a,int[] b) {
    ArrayClassContainer acc = new ArrayClassContainer( a.length + b.length );
    acc.putArray(a);
    acc.putArray(b);
    return acc.getArray();
}
```

Full code with the examples above

Below is the full working code of what I wrote during the lecture. For a file called **Demo.java**.

```
public class Demo {

    // this method prints the contents of an integer array into the
    // standard output (text console)

    public static void printAr(int[] x) {
        for (int c : x) {
            System.out.print(c + " ");
        }
        System.out.println();
    }

    // the main--this method is called when we run Demo

    public static void main(String[] args) {
        int[] p = {1, 2, 3, 4, 5};
        int[] q = {-1, -2, -3, -4};
        printAr(p);
        printAr(q);

        // let us c
        int[] c = acClass(p, q);
        printAr(c);
    }

    /**
     * Method acDirect takes two integer arrays and returns
     * a new array containing the elements of a and then b
     * in their original order. It performs a concatenation.
     */

    public static int[] acDirect(int[] a, int[] b) {
        int[] result = new int[a.length + b.length];
        // copy elements a
        for (int i = 0; i < a.length; i = i + 1) {
            result[i] = a[i];
        }
        // copy the elements of b
        for (int i = 0; i < b.length; i = i + 1) {
            result[a.length + i] = b[i];
        }
        return result;
    }

    /**
     * Method acShort takes two integer arrays and returns
     * a new array containing the elements of a and then b
     * in their original order. It performs a concatenation.
     * This method's behavior is identical to acDirect's.
     */

    public static int[] acShort(int[] a, int[] b) {
      int[] result = new int[a.length + b.length];
      int i = 0;

      for (int x : a) {
          result[i++] = x;
      }
      for (int x : b) {
          result[i++] = x;
      }
      return result;
    }

    /**
     * Method acClass takes two integer arrays and returns
     * a new array containing the elements of a and then b
     * in their original order. It performs a concatenation.
     * This method's behavior is as acDirect's and acShort's.
     */

    public static int[] acClass(int[] a,int[] b) {
        ArrayClassContainer acc = new ArrayClassContainer( a.length+b.length);
        acc.putArray(a);
        acc.putArray(b);
        return acc.getArray();
    }

}
```

```
/**
 * The ArrayClassContainer class is an object that contains an integer
 * array into which it copies information from other arrays.
 */

class ArrayClassContainer{

    // fields: the result and the current index

    int[] result;
    int i = 0;

    // constructor

    public ArrayClassContainer(int size) {
        result = new int[size];
        // i = 0; this is an alternative
    }

    // this method copies the content of anArray into result, sequentially
    // after the last elements copies

    public void putArray(int[] anArray) {
        for(int x: anArray) {
            result[ i++ ] = x;
            if (i >= result.length) i = 0;
        }
    }

    // it is in the culture of Java that, if you want to see the contents
    // of a field, you “need” to create a method that does so

    public int[] getArray() {
        return result;
    }

}
```

Midi Music Player

For the remaining of the lecture, I faced a strange Windoze problem... At any rate, I wanted to demonstrate the complicated code can be isolated in a class. By doing so, we can reuse the code in a simpler manner later on. The reusability of the code depends, in part, on how simple is the *interface* of the class created.

For example, the following class uses my midi player:

```
public class JukeBox {
    public static void main(String[] args) {
        SomeMidiPlayer smp = new SomeMidiPlayer();
        smp.playMidi("/midi/SixDivert.mid");
        smp.pause(30);
        smp.playMidi("/midi/Greensleeves.mid");
        smp.pause(30);
        smp.stop();
        smp.kill();
    }
}
```

The midi player, on the other hand, is more complicated and contains *dirty* code; however, since its complexity has been encapsulated, it is easy to use. (You don't need to understand this code; the whole point is that it is advanced and complicated.)

```
package jukebox;

/**
 *
 * @author mmonsalv
 */
import javax.sound.midi.*;
import java.io.*;

public class SomeMidiPlayer {

    Sequencer seq = null;
    boolean loaded = false;

    public SomeMidiPlayer() {
        openSystem();
    }

    public boolean openSystem() {
        if (seq!=null) return true;
        try {
            Runtime R = Runtime.getRuntime();
            seq = MidiSystem.getSequencer();
            seq.open();
        } catch (Exception e) {
            seq = null;
        }
        return (seq!=null);
    }

    /**
     * Closes the sequencer, freeing it's internal state. It is necessary
     * to kill the sequencer before the application finishes, unless System.exit
     * is summoned (forceful exit).
     */
    public void kill() {
        if (seq!=null) {
            seq.close();
        }
    }

    /**
     * Loads a midi song into the midi music player.
     *
     * @param res The address of the midi resource to be loaded. I generally
     * place midi music in the midi folder inside src; in such case, res would
     * be "/res/song.mid" for playing file song.mid.
     *
     * @return True if the song was effectively found and loaded. False
     * otherwise.
     */
    public boolean loadMidi(String res) {
        if (seq == null) {
            return false;
        }
        boolean status;
        try {
            if (seq.isOpen()) {
                seq.stop();
            }
            InputStream is = Runtime.getRuntime().getClass().getResourceAsStream(res);
            seq.setSequence( is );
            status = (seq.getSequence() != null);
        } catch (Exception e) {
            System.out.println(e);
            status = false;
        }
```

```java
        this.loaded = status;
        return status;
    }

    /**
     * Plays the midi song that is already loaded (if any) into the player.
     *
     * @return True if there was a song loaded and could be played.
     */
    public boolean play() {
        if (seq == null || !loaded) {
            return false;
        }
        if (seq.isRunning()) {
            seq.stop();
        }
        System.out.println("Has seq? "+seq.getSequence());
        seq.start();
        return true;
    }

    /**
     * Stops the reproduction of a midi song, if there was one playing.
     *
     * @return True if no song is being played.
     */
    public boolean stop() {
        if (seq == null || !loaded) {
            return true;
        }
        if (seq.isRunning()) {
            seq.stop();
        }
        return true;
    }

    /**
     * Plays a midi song by name.
     *
     * @param res The path to the resource. @see loadMidi
     *
     * @return True is successful. This occurs if the midi sequencer could
     * start, the song was found, it was loaded, it was recognized, and it
     * started playing.
     */
    public boolean playMidi(String res) {
        return !loadMidi(res) || play();
    }

    /**
     * Tries to put the program's current thread on pause (music and other
     * parts of the program will still run).
     *
     * @param seconds The number of seconds to pause.
     *
     * @return True if the pause occurred without internal inconveniences. False
     * could mean that no pause occurred.
     */
    public boolean pause(double seconds) {
        boolean retval = true;
        try {
            Thread.sleep((int) (1000 * seconds));
        } catch (Exception e) {
            retval = false;
        }
        return retval;
    }

}
```