5COSC001W - OBJECT ORIENTED PROGRAMMING

Lecture 5: Introduction to Collections (ArrayLists) and Arrays

Dr Dimitris C. Dracopoulos

email: d.dracopoulos@westminster.ac.uk

1 Collections

The Collections is a set of classes found in the `java.util` package.

Compared with arrays, collection classes:

- Can hold a number of objects as elements (arrays can store both primitives and objects).
- They can have an unlimited number of objects. The size of a collection increases dynamically as more objects are added into it.
- Although primitives cannot be stored directly, wrapper classes can be used to store the value of a primitive in a wrapper object. Such an object is then stored in a collection class.

2 The ArrayList class

An example of a class belonging to Java Collections. An unlimited number of objects can be stored in an `ArrayList`.

Example:

```
import java.util.*;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<String>();

        // Add three elements in the list
        al.add("aa");
```

```
        al.add("bb");
        al.add("ccc");

        for (int i=0; i < al.size(); i++) {
            String s = al.get(i);
            System.out.println(s);
        }

        // remove second element from the list
        al.remove(1);

        System.out.println("After remove(), al contains:");
        for (String e : al)
            System.out.println(e);
    }
}
```

When the above program is run, it displays:

```
aa
bb
ccc
After remove(), al contains:
aa
ccc
```

Note that `ArrayLists` which can only store a specific type, such as in the above example (`al` holds string objects), are called *generics* or *parameterised types*. This is because prior to Java 1.5, only non-parameterised collections classes were allowed. Non-parameterised collections can store objects of any type (even a mixture of objects belonging to different classes).

3 Wrappers and Storing Numbers in Collections

Primitives cannot be stored directly in a collection class. The Java library contains wrapper classes which correspond to primitives, as their objects are capable to store a primitive value.

The wrapper classes are:

- `Byte`
- `Short`
- `Integer`
- `Long`
- `Float`
- `Double`

- `Boolean`
- `Character`

The code below creates an object of `Double` class, which holds a double value.

```
Double d1 = new Double(3.1);
Double d2 = 3.1;  // automatically creates a Double object from 3.1
```

The two above statements are equivalent. The second form can only be used in Java version 1.5 or later. The automatic conversion of a primitive to the corresponding wrapper class (e.g. `double` to a `Double`) is called autoboxing, while the opposite conversion is called `unboxing` (e.g. automatically converting a `Double` to a `double`).

Example:

```
import java.util.*;

public class WrapperExample {
    public static void main(String[] args) {
        // create an ArrayList object storing Double objects
        ArrayList<Double> a = new ArrayList<Double>();

        Double d1 = new Double(5.4);
        a.add(d1);
        a.add(11.2);  // autoboxing occurs (double -> Double conversion)

        //a.add(new Integer(2));  // Error!

        // get the second element from arraylist
        Double d2 = a.get(1);

        // get the 1st element - unboxing occurs (Double -> double conversion)
        double d3 = a.get(0);

        System.out.println("d2=" + d2 + ", d3=" + d3);
    }
}
```

When the above example is run, it displays:

```
d2=11.2, d3=5.4
```

4 Non-parameterised ArrayLists

Although non-recommended, an `ArrayList` object can be declared to store objects of any type. In such cases, explicit casting is required when obtaining an element from the list.

This is illustrated in the example below:

```
import java.util.*;

public class ArrayListExample2 {
    public static void main(String[] args) {
        ArrayList l1 = new ArrayList();

        l1.add(new Integer(11));
        l1.add(new Integer(3));
        l1.add(new Integer(55));

        for (int i=0; i < l1.size(); i++) {
            Integer k1 = (Integer) l1.get(i);  // Cast is required!
            System.out.println(k1);
        }
    }
}
```

Getting an element from list in `l1` requires a cast. If the cast is omitted, the compiler will report an error.

5 Arrays

A constant number of primitive types or objects can be stored in an array. This is done for convenience, for example instead of declaring ten `int` variables, an array to store ten `int`s can be declared. Using the array, the ten different integers will be manipulated using the same symbol (array name) and a different index.

- Java arrays are objects (they are allocated in the heap).

5.1 Declaring an array

```
int a[];
```

After the above declaration, `a` is a variable which can hold a reference to an array of any size containing `int`.

Note that arrays can store not only primitives, but objects as well. For example, assume that class `Book` is defined, then the following is valid:

```
Book library[] = new Book[500];
```

5.2 Creating an array

```
a = new int[5];
```

The size of an array cannot be changed once it is created. After the creation of array `a`, the situation is as shown in Figure 1.

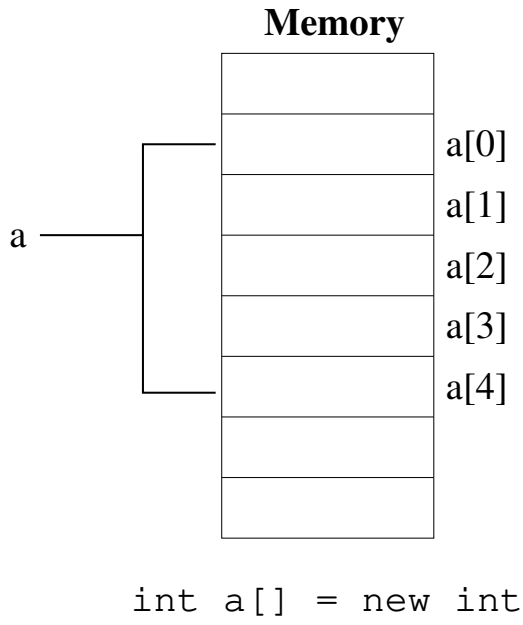


Figure 1: The creation of an array in the heap, with the statement `int a[] = new int[5];`.

Example:

```
/**
    A class to simulate the operation of a lottery
*/
public class Lottery {
    int results[];

    /**
        Constructs a lottery object with empty results
    */
    public Lottery() {
        results = new int[6];
    }

    /**
       Simulates the lottery draw by filling in array results.
       The random generator should be called normally, but
       this class demonstrates arrays so for simplicity numbers
       are fixed.
    */
    public void draw() {
        results[0] = 11;
        results[1] = 45;
        results[2] = 3;
        results[3] = 24;
        results[4] = 12;
        results[5] = 31;
    }
```

```
    /**
        Prints on the screen the latest draw results
    */
    public void printResults() {
        System.out.println("The latest lottery results are:");
        for (int i=0; i < results.length; i++)
            System.out.print(results[i] + " ");

        System.out.println();
    }

    public static void main(String[] args) {
        Lottery lot = new Lottery();
        lot.draw();
        lot.printResults();
    }
}
```

Note that in order to get the size of an array, its `length` field can be accessed. Thus, `results.length` returns 6 which is the size of array `results`.

When the above program is run, it displays:

```
The latest lottery results are:
11 45 3 24 12 31
```

5.3 Initialising Arrays

There are two ways to initialise an array:

- Assign a value to each element individually.

  ```
  double b[] = new double[10];
  b[0] = 5.0;
  b[1] = 1.2;
  ```

- Use an "array initialiser" at the point of declaration:

  ```
  String weekdays[] = {"Mon", "Tue", "Wed", "Thu", "Fri"};
  ```

  or

  ```
  String weekdays[] = new String[] {new String("Mon"),
                                    new String("Tue"),
                                    new String("Wed"),
                                    new String("Thu"),
                                    new String("Fri")};
  ```

  The latter form of "array initialiser" can be used to initialise an array not only where the array is declared, but also in a statement.

6 Two Dimensional Arrays (Arrays of Arrays)

A 2-dimensional array in Java is an array of an array.

```
Book mybooks [] [];  // declaration of an array of array
mybooks = new Book[10][12];  // an array[10] of array[12]
```

The above declares an array `mybooks` with 10 elements, each element containing an array with 12 `Book` elements. Then an array 10 of array 12 is created and assigned to `myBooks`. Thus, `mybooks[0]` contains an array capable of holding 12 `Book` objects, and `mybooks[1]` contains another array capable of storing `Book` objects.

Because object declarations as the one above, do not create objects, the elements of an array of an array must be created first, before using the array in any meaningful way (see Figure 2):

```
mybooks[i][j] = new Book();
```

This is also illustrated in the example below:

```
class Book {
    String colour;
}


public class ArrayExample {
    public static void main(String[] args) {
        Book mybooks [] [] = new Book[10][12];  // an array[10] of array[12]
        System.out.println(mybooks[0][0]);

        mybooks[0][0] = new Book();
        System.out.println(mybooks[0][0]);
    }
}
```

When the program is run it prints:

```
null
Book@f6a746
```

Note that an array of an array does not need to contain the same number of elements in each row, as the example `ArrayExample2` in the next section shows:

7 Looping over Arrays - The for-each loop

Java 1.5 introduced a new form of the `for` loop, which can be conveniently used to traverse arrays and Collections (e.g. `ArrayLists`).

Syntax:

```
for (element : name)
    statement
```

where *name* is the name of the array or collection the elements of which need to be traversed.

The new "enhanced loop" is shown in the example below:

```
public class ArrayExample2 {
    public static void main(String[] args) {
        String a[] = new String[3];
        a[0] = "aa";
        a[1] = "bb";
        a[2] = "cc";

        System.out.println("Array a contains:");
        for (String i : a)
            System.out.println(i);

        /* an array of an array containing different number
           of elements in each row */
        int myNumbers[][] = new int[][] {
                                    {0},
                                    {0,1},
                                    {0,1,2},
                                    {0,1,2,3}};

        System.out.println("\nArray myNumbers contains:");
        for (int[] r : myNumbers) {  // for each row
            for (int c : r) {     // for each element in current row
                 System.out.print(c + " ");
            }
            System.out.println();
        }
    }
}
```

The above code displays:

```
Array a contains:
aa
bb
cc

Array myNumbers contains:
0
0 1
0 1 2
0 1 2 3
```

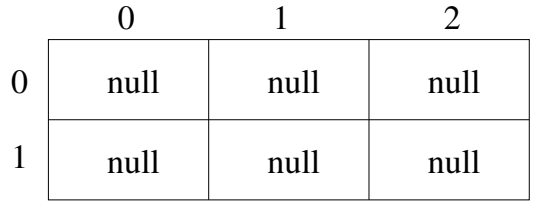
8 Another example on arrays

Although arrays cannot change their size, variables references to arrays can be made to point to a different array. This is shown in the code below and illustrated in Figure 3:

```
public class ArrayReferencesExample {
    public static void main(String[] args) {
        int a[] = new int[3];
        a[0] = -1;
        a[1] = 5;
        a[2] = 4;

        int b[];
        b = a;
        System.out.println("a is located at address " + a +
                           ", b is located at address " + b);

        a = new int[5];
        for (int i=0; i < 5; i++)
            a[i] = i+1;

        System.out.println("After a = new int[5]");
        System.out.println("a is located at address " + a +
                           ", b is located at address " + b);
        System.out.println("\na contains: ");
        for (int n : a)
            System.out.print(n + " ");

        System.out.println("\nb contains: ");
        for (int n : b)
            System.out.print(n + " ");
        System.out.println();  // add a newline
    }
}
```

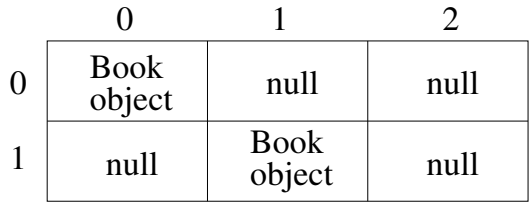
When the above example is run, it displays:

```
a is located at address [I@126b249, b is located at address [I@126b249
After a = new int[5]
a is located at address [I@f6a746, b is located at address [I@126b249

a contains:
1 2 3 4 5
b contains:
-1 5 4
```



(a) After:

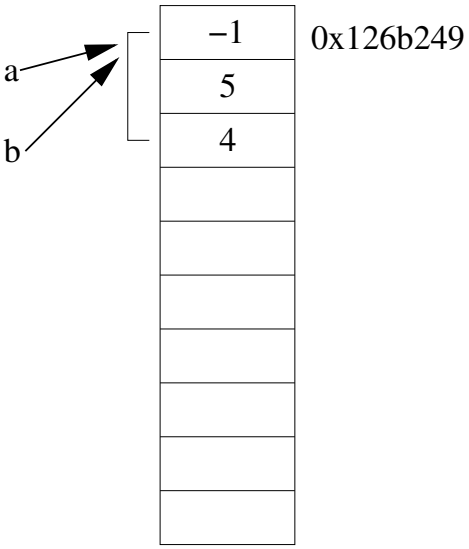
```
Book m [] [] = new Book[2][3];
```



(b) After:

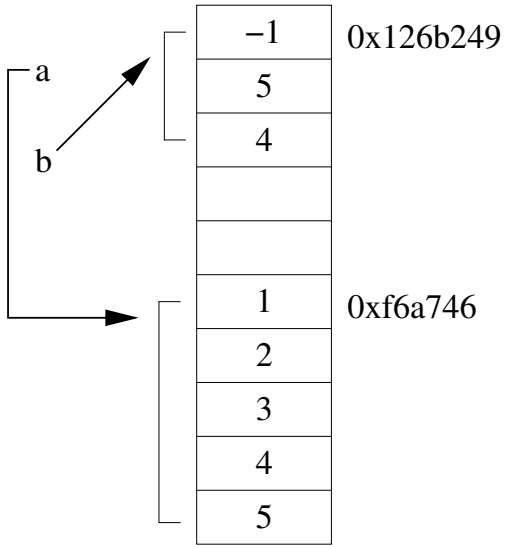
```
m[0][0] = new Book();
m[1][1] = new Book();
```

Figure 2: Object elements in an array must be created explicitly, before using the array in a useful way. In (a), an array with 2x3 declared objects is defined. This implies that the objects have not been created yet, and the elements of the array cannot be used. In (b), elements `m[0][0]` and `m[1][1]` are assigned to an object and from that point these elements can be used.



a) After:

```
int a[] = new int[3];
a[0] = -1;
a[1] = 5;
a[2] = 4;

int b[];
b = a;
```



b) After:

```
a = new int[5];
for (int i=0; i < 5; i++)
    a[i] = i+1;
```

Figure 3: Variables references to arrays can be changed, so as to point to a different array than the one initially pointing to. The statements in b) are executed after the statements in a) are executed.

9 The Arrays class

The `Arrays` library class located in package `java.util`, provides a number of useful utilities (in the form of *static* methods[1] to manipulate arrays.

These include:

- Fill parts (or the whole) of the array with values.
- Compare arrays element by element
- Search.
- Sort.

Similarly with all other Java library classes, the full details of `Arrays` can be found in the Java API documentation.

10 Collections vs Arrays

One might wonder why not choose a Collection class (e.g. an `ArrayList`) as opposed to an array, all of the time. After all, collections provide an unlimited capacity to store elements.

The reason behind choosing arrays instead of collections, is computational efficiency. If the number of elements to be stored in a data structure is constant, then arrays should be preferred.

The computational efficiency of arrays over collections has to do with two main reasons:

1. As collections provide an unlimited capacity, when the initial capacity that a collection class has allocated is reached, additional memory should be allocated to increase the capacity. The allocation of extra capacity from the JVM requires some extra CPU time.
2. Some collections, are implemented in such a way that random access of elements is not possible. This means, that to access e.g. the third element, all elements before the third one must be traversed. A `LinkedList` is implemented in this way.

   This is not true for `ArrayList`s because their internal implementation uses an array to store the elements. Therefore there is no time overhead associated with the random access of an element in `ArrayList`s. However, because of the first reason, arrays are still more efficient and faster than `ArrayList`s

[1] A *static* method is a method which can be called directly on the class, without creating an object of the class. For example, assuming class `A` contains a method called `foo()`, the method can be called as `A.foo()`. More details on this will be given in subsequent lecture notes.