Scientific Programming Using Object-Oriented Languages
Module 5b: Arrays and Collections

Aims of Module 5b:

- Be able to store and manipulate collections of objects.
- Understand what happens when Java passes a collection as a parameter in a function call.
- Be familiar with the following:
  - Arrays, ArrayLists and HashMaps
  - Iterators and for loops for looping over collections

Introduction to Collections

- So far have to declare a separate variable or create a new object each time we read a “unit” of data
- Not suitable for large, variable or unknown quantities of data
- Many (probably most) applications do not deal with small, known amounts of data:
  - particle trajectories in a high-energy physics experiment;
  - spectroscopic data from an astronomical survey;
  - shapes in a drawing program;
  - customers in sales software…
- Two ways of doing this in Java:
  - arrays: relatively low-level, simple, limited;
  - collections: variety of types, support many operations, use object orientation to support sophisticated software requirements.

Arrays

- Contiguous block of memory with a variable (or object reference) in each “slot”.
- Syntax is as follows:

```
// An array storing two integer values:
int[] ks = new int[2];
ks[0] = 45; ks[1] = 46;
// An array storing some double values:
double[] ad = {0.1, 0.2, 0.3, 0.4, 0.5, 0.8, 1.5};
// An array storing 10 Point objects:
Point[] mypoints = new Point[10];
mypoints[0] = new Point(10,11);
mypoints[0].x = 12;
// ... etc.
```

- The size of the array can be given by a variable, so it need not be known at compile time, but once the array has been created its size cannot be altered.

Using Arrays

- The elements of an array with n elements are numbered [0] to [n-1]
- The number of elements is given by the data member `length`:

```
// Declare an array to store six integers:
int[] values = new int[6];
// Loop over the elements:
for (int k = 0; k < values.length; k++) {
   values[k] = values[k]*values[k];
}
// The following line will produce an error:
int ix = values[6];

// Another way of looping if you don't need to change
// the array contents:
int sum=0;
for (int j : values) {sum += j;}
```

Arrays and Functions

- An array can be passed as a parameter or used as the return value of a function:

```
public static int[] negativeList(int[] data) {
    int[] nData = new int[data.length];
    for (int i = 0; i< data.length; ++i){
        nData[i] = -data[i];
    }
    return nData;
}
```

- Arrays are objects, and array variables are references, so have to be careful as function can change the contents of the array passed.
- It is good practice to return a new array, as in this example, rather than modifying the input array.

Multi-Dimensional Arrays

- A multi-dimensional array in Java is an array of arrays:

```
int[][] grid = new int[4][5];
grid[0][0] = 1; // first element
grid[3][4] = 1; // last element
// or we can initialise as follows:
int[][] twoDGrid = { {1,2,3,4}, // row-1
                     {11,12,13,14}, // row-2
                     {11,12,13,14} }; // row-3
int nRows = twoDGrid.length;
int nCols = twoDGrid[0].length;
```

- Each row is a separate 1D array so they can even have different lengths: be careful!
- If you go beyond three dimensions it might be better to consider alternative approaches, perhaps using a custom class.

The Java Collections Framework

- The Java collections framework goes beyond the basic capabilities of arrays, providing a number of
  - *interfaces*, representing different types of collection (more in module 6);
  - *implementations*, concrete classes that you can use;
  - *algorithms*, enabling you to process data in various types of collection.
- The collection classes fall into a few broad types:
  - set
  - list: we will use the `ArrayList` class
  - queue
  - map (not a collection in exactly the same sense but part of the same framework): we will use the `HashMap` class

Generics

- *Generics* is the name given to a useful feature introduced in Java 1.5
- We will explain generics mostly by example in using the collection classes, but a brief overview follows…

- Like an array, a collection may contain objects of any type.
- In module 6 we will see how we could use type `Object` to refer to elements of a collection and store different types of object in the same collection.
- In general we know at compile time what type of object we will store.
- To avoid errors when we try to extract a `ParticleDetector` from a collection of `Integer`s, we need to tell the compiler what type we are using.
- This is where generics come in…

The `ArrayList` Class

- An `ArrayList` is basically a more sophisticated 1D array.

```
ArrayList<Point> mypoints = new ArrayList<Point>();
ArrayList<Integer> mynumbers = new ArrayList<Integer>();
for (int x = 0; x < 10; x++){
    for (int y = 0; y < 10; y++) {
        Point p = new Point(x,y);
        mypoints.add(p);
        mynumbers.add(x*y);
    }
}
```

- Here we see the syntax for using generics
  - `ArrayList` is a *generic type*
  - `ArrayList<Integer>` is a *parameterized type*
  - `Integer` is a *type parameter*
- Note that we do not specify at the beginning how many elements we will add to the `ArrayList`.

The `ArrayList` Class (Continued)

- We retrieve elements from an `ArrayList` as follows:

```
Point p = mypoints.get(2);
int val = mynumbers.get(1);
```

- There are many other methods available for manipulating `ArrayList`s:

```
ArrayList<String> list = new ArrayList<String>();
list.add("hello”); // add to end
list.add("and"); // add to end
list.add("welcome"); // add to end
list.add(2,"another"); // add 3 at position 2
list.set(3,"word");
String o = list.get(1);
list.remove(2);
list.clear();
boolean contains_and = list.contains("and");
int location = list.indexOf("and");
```

Boxing and Unboxing

- A collection can only hold objects, not primitive values.
- That is why we use `ArrayList<Integer>` not `ArrayList<int>`
- But we are apparently storing and retrieving **`int`** values.
- The conversions from **`int`** to `Integer` (*boxing*) and from `Integer` to **`int`** (*unboxing*) are carried out automatically.
- We came across this type of conversion in module 2.

Looping Over A Collection

- Two main techniques: for loops and iterators.
- Using an iterator is more flexible: we can change the contents of the collection as we move through it:

```
Iterator<String> it = list.iterator();
while (it.hasNext()) {
    String s = it.next();
    it.remove(); // removes current element
}
```

- A for loop is much simpler:

```
for (String word : list) {
    System.out.println(word);
}
```

- Similar to the other type of for loop we have already come across.

Sorting A Collection

- The Collections class has static methods for operating on collections:

```
ArrayList<Integer> nums = new ArrayList<Integer>();
for (int i = 10; i > 0; i--) {
    nums.add(i);
}
Collections.sort(nums);
```

- For numeric types like Integer, the default behaviour is to sort into increasing order.
- We can change this behaviour if we want to: see notes for details.

The `HashMap` Class

- A HashMap (like other types of map) is a look-up table:
  - contains keys and values
  - can find the value corresponding to a given key
  - keys must be unique
  - keys and values must both be objects, but (un)boxing can be used
- Examples include
  - dictionary: word → definition
  - financial software: account number → bank record

```
HashMap<String,String> m = new HashMap<String,String>();
m.put("one","eins");
m.put("two","zwei");
m.put("three","drei");
System.out.println(m.get("one"));
```

Looping Over A `HashMap`

- A HashMap is not just a collection of elements.
- But we can access the keys or values (or key-value pairs) as collections:

```
for (String s : m.keySet()) {
   System.out.println(s+" translates as "+m.get(s));
}
```

or:

```
for (Map.Entry<String,String> entry : m.entrySet()) {
   System.out.println(entry.getKey()+" translates as "+
       entry.getValue());
}
```