

Data Structures G5029

Lecture 1

Kingsley Sage
Room 5C16, Pevensey III
khs20@sussex.ac.uk

© University of Sussex 2006

Lecture 1

- Introduction
 - This course introduces the Java programming language ...
 - ... and the use of data structures as a means of representing a wide range of tasks in general computing
 - Assessment is by coursework (two Java programming exercises) and by formal examination
- Resources
 - On-line course notes (by Peter Williams)
 - Lab sessions
 - Course web site at <http://www.informatics.sussex.ac.uk/courses/dats>

Primitive data types in Java

- The simplest data structures in the Java programming language are the built in primitive data types `byte`, `short`, `int`, `long`, `float`, `double`, `char` and `boolean`.
- Wrapper classes – the difference between `byte` and `Byte`, `int` and `Int` ...
- All numeric types in Java are signed (unlike C/C++).
- The expression “data structure” is usually used to refer to more complex ways and storing and manipulating data, such as arrays, queues, stacks etc ...
- We begin by discussing the simplest, but one of the most useful data structures, the **array**.

Array variables

- Array: a simple list of values referenced by an array index. Before use, an array must be properly initialised

```
int[] a; // a is a reference to an array of integers
a = new int[20]; // allocates OS storage for 20 integers
int[] a = new int[20]; // combines the above into one line
a[2] = 173; // gives the array element index 2 the value 173
           // array element 2 is the third item in the array
a[i] = 34; // array element i - i must be of type byte,
           // short or int
```

Array bounds and initialisation

- Array indexing in Java is zero-based (like C/C++, but unlike MATLAB).
- When Java arrays are created using new operator, their elements are automatically initialised, but it's better not to rely on it.

```
for (int x = 0; x <20; x++) {
    a[x] = 0;
}

// Arrays can also be created and initialised in one operation
// using a list of values ...

int[] a = {24,65,34,96,12}; // Correct with implied array size 5

int[] a;
a = {24,65,34,96,12}; // Not legal - why not? What's missing?
```

Array bounds and lengths

- When an array is created, its size is held in a public constant which can be accessed by the expression e.g. `a.length`. Why is this useful?
- Accessing an array element that is "out of bounds" generates a `ArrayIndexOutOfBoundsException` at runtime.

```
// To initialise an array in a totally safe manner ...

for (int i = 0; i < a.length; i++) {
    a[i]=0;
}
```

Assignment

- Arrays are treated as objects in Java, and as such, obey the same rules for equality and assignment.
- An array variable is a reference variable (like a pointer variable in C/C++) . It stores information about where to locate the array elements. All array variables required the same amount of memory storage.

```
int[] a = new int[20];

int[] b; // Just a reference variable ...

b = a;

// a[x] and b[x] refer to the same data - there is only one
// array. Any change to b[x] will change a[x] identically
// and vice versa
```

Assignment

- To actually copy an array takes a bit more work ...

```
int[] a = new int[20];

b = new int[a.length];
for (int i = 0; i < a.length; i++) {
    b[i] = a[i];
}
```

- Now there are 2 arrays and you can change one without affecting the other.

Equality

- Equality behaves in a similar manner ...
- `a == b` is true if and only if the array reference variables `a` and `b` stored the same address.
- More commonly, you might wish to check whether `a` and `b` are of the same length and have the same contents. In that case, you need to write your own simple method ...

Arrays as parameters

- Array parameters can be used as parameters to methods.
- When an array is passed to a method, only the reference is passed to the method, which makes for efficient passing of arrays to methods (what's the alternative?)
- Method has access to the elements of the array, via the reference, which means that they can be changed as the method chooses.
- The reference itself will not be changed outside the method.

```
int[] a = new int[20];  
  
a[0] = 3;  
foo(a);  
  
void foo(int[] x) {  
    x[1] = 4;  
}
```

Arrays of Objects

- Thus far, we have only considered arrays of elements of the various primitive types. All elements of such arrays must be of the same type. But we can also have arrays of objects of any class.
- It is important to remember that arrays must be initialised if they are to hold data that is different from the default initialisation.
- For arrays of Objects, the default initialisation is the `null` reference.

```
String[] words = new String[20];  
Object[] heap = new Object[20];
```

Multi-dimensional arrays

- The elements of an array can themselves be arrays. This is a special case of having arrays of Objects. This is known as a multi-dimensional array.

```
int a[][] = new int[10][20];  
  
// It can be initialised using ...  
for (int x=0;x<a.length;x++) {  
    for (int y=0;y<a[x].length;y++) {  
        a[x][y] = 0;  
    }  
}  
  
// Elements are accessed as a[x][y] and are useful for  
// representing matrices. Rows and columns do not necessarily  
// have to have the same number of elements
```

Array access and memory allocation

- One of the principal reasons why arrays are used so widely is that their elements can be accessed in constant time.
- This means that it takes the same time to access `a[1]` as `a[10]`.
- This is because the address of `a[x]` can be determined arithmetically by adding a suitable offset to the machine address of the head of the array.
- This relies on the fact that the elements of the array are stored in a contiguous block of memory.
- This has advantages and disadvantages. Arrays can also be allocated dynamically (size is only known at run time rather than compile time).
- Dynamically sized arrays are implemented as Vectors in Java (like C/C++)

Next time ...

- The first of our more complex and useful data structures
 - The stack