Appendix C

General formula for calculating memory usage of java objects

It is interesting to detail the heap memory used by a Java object, since according to SQuO prototype this measure was important to observe its behavior and to determine the contribution of this thesis project. The memory used by a Java object consists of [Co083]:

- an object header from few bytes of "housekeeping" information
- memory for primitive fields, according to their size (Table 1)
- memory for reference fields (4 bytes each)
- padding: potentially a few "wasted" unused bytes after the object data, to make every object start at an address that is a convenient multiple of bytes and reduce the number of bits required to represent a pointer to an object

Table 1 presents the byte size of the different Java primitive data types.

Java type	Bytes required
Boolean	1
Byte	
Char	2
Short	
Integer	4
Float	
Long	8
Double	

Object size granularity

Every object occupies a number of bytes that is a multiple of 8. If the number of bytes required by an object is not 8 multiple, round it up to the next multiple of 8 is required. This means, for example, a bare Object takes up 8 bytes. An instance of a class with a single boolean field takes up 16 bytes: 8 bytes of header, 1 byte for the boolean and 7 bytes of "padding" to make the size up to a multiple of 8. An instance with *eight* boolean fields will also take up 16 bytes: 8 for the header, 8 for the boolean; since this is already a multiple of 8, no padding is needed. Finally, an object with two long fields, three integer fields and a boolean will take up [Co083]:

- 8 bytes for the header;
- 16 bytes for the 2 longs (8 each);
- 12 bytes for the 3 ints (4 each);
- 1 byte for the boolean;
- a further 3 bytes of padding, to round the total up from 37 to 40, a multiple of 8.

Memory used by Java arrays

In Java there exist arrays from different dimensions. An array is a special type of object. A multi-dimensional array is an array of arrays i.e. a two-dimensional array, every row is a separate array object [Co084].

Single-dimension array

A single-dimension array is a single object. As expected, the array has the usual object header. However, this object head is 12 bytes to accommodate a four-byte array length. Then comes the actual array data which, as you might expect, consists of the number of elements multiplied by the number of bytes required for one element, depending on its type. The memory usage for one element is 4 bytes for an object reference; for a list of the memory usage of primitive types. If the total memory usage of the array is not a multiple of 8 bytes, then the size is rounded up to the next multiple of 8 (just as for any other object) [Co084].

Two-dimensional array

In a language such as C, a two-dimensional array (or indeed any multidimensional array) is essentially a one-dimensional array with judicious pointer manipulation. This is not the case in Java, where a multidimensional array is actually a set of nested arrays. This means that every row of a two-dimensional array has the overhead of an object, since it actually *is* a separate object [Co084].

For example, let's consider a 10x10 int array. Firstly, the "outer" array has its 12-byte object header followed by space for the 10 elements. Those elements are object references to the 10 arrays making up the rows. That comes to 12+4*10=52 bytes, which must then be rounded up to the next multiple of 8, giving 56. Then, each of the 10 rows has its own 12-byte object header, 4*10=40 bytes for the actual row of ints, and again, 4 bytes of padding to bring the total for that row to a multiple of 8. So in total, that gives 11*56=616 bytes. That's a bit bigger than if you'd just counted on 10*10*4=400 bytes for the hundred "raw" ints themselves.

Multidimensional arrays

For arrays of more than 2 dimensions, the above logic repeats: each row of the "outer" array is now an *array of references* to a further array, which contains the actual primitive data (or references if it is an object array) [Co084].