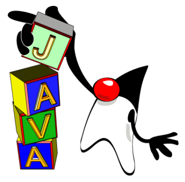
Java Programming 2 – Lecture #13 – Jeremy.Singer@glasgow.ac.uk



JVM Memory Layout

Runtime memory in the Java virtual machine is organized into distinct areas[1] as you know from CS2.

The **stack**[2] is used to store local variables that belong to a single method. When the method is invoked, a stack frame for that method is pushed onto the stack. When the method returns, the corresponding stack frame is popped from the stack. This means that local variables are only alive from the time the method is called until the time the method returns.

The **heap** is used to store data that is dynamically allocated via the `new` keyword, such as objects and arrays. Heap-allocated data remains alive so long as it is reachable from a root reference (e.g. a local variable on a stack frame or a global variable in a static field). When heap-allocated data is no longer reachable, it may be garbage collected. Garbage collection is triggered when the heap fills up, or when the program invokes `System.gc()`[3]. If there is not enough space in the heap for new data to be allocated, then the virtual machine throws an `OutOfMemoryError`.

Each class file contains a **constant pool** area, which stores compile-time constants such as string literals and integer values. Also runtime linking information (method names, etc) is stored in the constant pool. When classes are loaded at runtime, the constant pool information is copied into the virtual machine constant pool area which is shared between all runtime threads.

Inserting into ArrayLists

Last week we saw that we could compare objects that implement the Comparable interface. Let's use this to insert objects into an ArrayList to maintain a sorted order. Effectively, this is the basis for the *insertion sort* algorithm which you will learn more about in ADS2 next semester.

```
list = new ArrayList<Comparable>();
for (String arg : args) {
  boolean inserted = false;
  for (int i=0; i<list.size() && !inserted; i++) {
    if (list.get(i).compareTo(arg)>0) {
      // first list element that's greater than arg
      list.add(i, arg);
      inserted = true;
    }
  }
  if (!inserted) {
    // add arg to end of list (largest element)
    list.add(arg);
  }
}
```

[1] See http://docs.oracle.com/javase/specs/jvms/se5.0/html/Overview.doc.html#1732 for full details.
[2] In fact there is one stack per thread, but we will not consider multi-threaded code just yet.
[3] See http://docs.oracle.com/javase/7/docs/api/java/lang/System.html

Functional Operations on Lists

Imagine you want to compute a sum of squares for a list of integers. Using the Java idioms we know already, the code might like similar to:

```
int sum = 0;
for (int i : list) {
  squareList.put(i*i);
}
for (int square : squareList) {
  sum += square;
}
```

It would be possible to fuse the two loops into a single loop body, although the code might look less clear and an optimizing compiler will probably do this anyway. In either case, the code looks messy. This kind of map / reduce computation is better expressed using a functional idiom.

The next version of Java (i.e. Java 8) should support lambdas, which are anonymous functions that can be applied to streams of data. The above example might be rewritten as:

```
int sum = list.stream().map(i -> {i*i}).sum();
```

This code is simpler and more intuitive in every way. The programmer intention is explicitly seen in the program. This code is functional, elegant and parallelizable. Note that the precise syntax for lambda expressions is subject to change. Java 8 pre-release binaries are available now[4]. Other JVM languages (e.g. Scala, Clojure, Groovy) already have extensive support for functional operations on list-like data structures.

Example in Scala

Scala is a JVM language that has more functional characteristics. Here is an interactive session I typed into the Scala online interpreter[5].

```
val l = List(1,2,3,4,5)
l: List[Int] = List(1, 2, 3, 4, 5)
val l2 = l.filter(n=>n%2==0)
l2: List[Int] = List(2, 4)
val l3 = l.map(n=>n*n)
l3: List[Int] = List(1, 4, 9, 16, 25)
val sum = l3.foldLeft(0)(_+_)
sum: Int = 55
```

[4] Download from https://jdk8.java.net/
[5] Try for yourself at http://www.simplyscala.com/