MSc Programming in Java
Specimen Solutions

Stephen Gilmore
Laboratory for Foundations of Computer Science
The University of Edinburgh

April 2002 examination

1. (a) From the sandbox, an applet cannot:
   - read from or write to the local filesystem, preventing such operations as the creation or deletion of files and directories (a security issue—integrity);
   - perform networking operations, preventing any snooping on local privileged ports (a security issue—confidentiality);
   - make use of system facilities to exit the Java interpreter, spawn new run-time processes or dynamically load code (a safety issue);
   - make use of AWT facilities which would allow the applet to spoof the on-screen appearance of another program (a security issue—confidentiality);
   - access system properties or create non-local threads (a safety issue);
   - load or re-define core Java packages, including the `ClassLoader`, apply reflection to non-public members of a class, or register its own `SecurityManager` object (security and safety issues).

   (b) The Java run-time system component with the responsibility for policing the sandbox restrictions is the *SecurityManager* of the `java.lang.SecurityManager` class.

   The sandbox model works because all of the core Java classes which perform security-sensitive operations first ask permission of the resident `SecurityManager`. If the sensitive operation is being invoked either directly or indirectly from untrusted code then the `SecurityManager` throws an exception to prevent the operation taking place.

   (c) Java 1.1 has an event-driven programming model based around the idea of *event sources* and *event listeners*. An event listener is an interface which is implemented by an *event adapter*. To create particular kinds of event listeners it is often easiest to subclass an event adapter.

   Along with the event model of Java 1.1 was introduced the language construct of *inner classes*. These are particular local classes used to create and package methods locally to the statements in the code where they are attached to a UI object.

   (d) The typical concern was that an applet would *spoof* the appearance of another application, inviting the user to type in passwords or other sensitive data, which would then be relayed back to the host computer. This is prevented by badging each window with the message *warning: untrusted applet window*.

(e) The `paint()` method is below.

```
public void paint(Graphics g) {
    for (int x = 0 ; x < 8 ; x++) {
        for (int y = 0 ; y < 8 ; y++) {
            if ((x + y) % 2 == 0)
                g.setColor(Color.white);
            else
                g.setColor(Color.black);
            g.fillRect(x * 100, y * 100, 100, 100);
        }
    }
}
```

(f) The `mousePressed()` method is below.

```
public void mousePressed(MouseEvent e) {
    row = e.getY() / 100;
    column = e.getX() / 100;
}
```

2. (a) Using threads is an example of *concurrency* in a program. In many applications it can be natural to think of several threads of execution within a program, each making progress on its work. It may be useful to structure the program in this way even if ultimately the threads are *multi-tasked* onto a single processor.

   The concurrent programming paradigm can be contrasted with the *parallel programming* paradigm where several physical processors are available and the use of parallelism in the program is to speed the execution of the program by making use of all of the available processors. In contrast, multi-tasking threads onto a single processor might actually slow the execution of the program (but it provides a more natural programming environment for the application programmer and thereby reduces the potential for error).

   (b) One way is to declare a class to be a subclass of `java.lang.Thread`. This subclass should *override* (re-define) the `run()` method of class `Thread`. An instance of the subclass can then be allocated with `new` and started using the `start()` method.

   The other way to create a thread is to declare a class that implements the `java.lang.Runnable` interface. That class then implements the `run()` method. An instance of the class can then be allocated, passed as an argument to one of the constructors of class `Thread`, and started using the `start` method.

   In both cases the `run()` method is a public void method

   ```
   public void run()
   ```

   Java only supports single inheritance so the second method must be used to make a subclass of `MyClass` into a thread.

   (c) In object-oriented programming the local state of an object is typically stored in private fields, modified in a controlled way through accessor methods.

   If it would be problematic for more than one thread at a time to invoke a method the method can be marked with the access modifier `synchronized`.

   Each object contains a single *lock* (also called a *monitor*) which is acquired by the first thread to invoke a synchronised method on that object. No other synchronised method can be invoked until the first thread finishes and releases the lock. Static synchronised methods lock the `Class` object for a class.

   (d) Daemon threads are low-priority threads in the Java Virtual Machine distinguished by the fact that the Java Virtual Machine exits when the only threads running are daemon threads. The `setDaemon()` method must be called before the thread is started. The `isDaemon()` method can be used to test whether or not a thread is a daemon thread.

   (e) The use of these methods was found to encourage poor concurrent programming style.

   - Use of the `stop()` method on a thread causes it to unlock all the objects that it has locked. If any of these were in an inconsistent state, other threads may now view these objects in an inconsistent (or *damaged*) state.
   - Use of the `suspend()` method is very likely to cause a *deadlock*. If the target thread holds a resource when it is suspended, no thread can access the resource until the target thread is resumed. If the thread that would resume

the target thread attempts to lock the resource prior to calling resume, deadlock results.

(f) This class implementation is relatively routine. The only obvious way to go wrong is to forget to prevent the invoking thread from reading the result before the `run()` method has completed. Thus we `wait()` in the `getResult()` method and we `notify()` from the `run()` method.

```
class Fib extends Thread implements Fibonacci {
    private int n;
    private int result;
    private boolean ready;

    public synchronized int getResult() {
        try {
            if (!ready)
                wait();
        } catch (InterruptedException ie) {}
        return result;
    }

    public synchronized void setInput(int n) {
        this.n = n;
    }

    public synchronized void run() {
        if (n == 0) result = 0;
        else if (n == 1) result = 1;
        else {
            Fib nMinusTwo = new Fib();
            nMinusTwo.setInput(n-2);
            nMinusTwo.start();
            Fib nMinusOne = new Fib();
            nMinusOne.setInput(n-1);
            nMinusOne.start();
            result = nMinusTwo.getResult() + nMinusOne.getResult();
        }
        ready = true;
        notify();
    }
}
```

3. (a) JIT compilers are essentially fast traditional compilers that translate the JVM bytecodes into native machine code on-the-fly. A JIT runs inside the virtual machine on the actual execution machine, and compiles each method the first time it is executed.

   Because a JIT compiler runs in "user time", it is severely constrained in terms of compile speed. This entails a trade-off that makes it more difficult to perform advanced optimisations, which mean that the compiled code is not as fast as might be expected.

   Further, traditional optimisations which can be applied to most programming languages cannot be applied to Java because of, among other things, the ability to perform dynamic loading of Java classes.

   Alternatives to using a JIT compiler include basic interpretation at one extreme and "way-ahead-of-time" compilation to native code on the other.

   (b) The operation of *garbage collection* is the maintenance, and periodic re-organisation of the heap to delete objects which will never be used again and compact together those which will be used again, freeing up unused space between.

   Garbage collection algorithms can be classified into copying garbage collectors and non-copying garbage collectors.

   A copying garbage collector divides the heap into two *semispaces*. Allocation takes place in the *from* space until it is full. Then objects are collected and copied into the *to* space and the semi-spaces switch roles and allocation can begin again.

   The major advantages of a copying memory manager include speed of object allocation, and the fact that the heap can be compacted during collection. This compaction results in faster access to objects from memory (fewer *page faults*, better *cache* performance).

   The advantages of a non-copying garbage collector include faster garbage collection (the objects are not being copied) and better use of the available memory.

   (c) The Java compiler implements nested classes by unwinding the nesting in order to produce a list of top-level classes. In order for this process to produce the correct results the compiler may need to rename classes in order to prevent name clashes. Further it will have to allow the class which was nested continued access to the fields and methods of the class which enclosed it.

   This latter adjustment is achieved by passing the enclosing object as an additional argument to all the constructors of its member classes. It is stored in a private field within the nested class and used as needed. In order to gain access to the private fields of the enclosing class the compiler generates accessor methods which the nested class can call.

   This compilation scheme is necessary because the Java Virtual Machine has no notion of nested classes and so these must be removed in the compilation process.

   (d) The Java *byte-code verification* process ensure that the code does not violate Java namespace restrictions and also checks that:

   (i). The code is valid Java Virtual Machine code and does not contain illegal byte codes.

   (ii). It does not overflow or underflow the run-time stack, thereby crashing the Java interpreter and leaving it in a vulnerable state.

(iii). It does not use registers incorrectly or convert datatypes incorrectly, thereby preventing memory arithmetic and *forging pointers*.

(e) (i). In the Jalapeño virtual machine many of the run-time system methods which would normally be written in C and called as native methods are implemented in Java.
The Jalapeño implementation must therefore occasionally escape from some of the Java language's restrictions in order to implement these run-time services.
However, this escape route from the Java language must not be offered to code which is not an authorised part of the virtual machine itself.

(ii). The run-time services of the Jalapeño virtual machine are collected into a distinguished class (called `MAGIC`).
When implementing methods such as the garbage collector or the dynamic linking method, the `MAGIC` class must manipulate Java object references in ways not allowed by the Java language such as turning an object reference into a raw address (an `int`), performing integer arithmetic on it, and turning it back into an object reference again.
Care must be taken when implementing these methods because the copying garbage collector of the Jalapeño virtual machine would cheerfully move an object to another location in memory, thereby invalidating the raw address which had been obtained from the object reference. For this reason, garbage collection is disabled while such methods are executing.

(iii).
- Exploitation of high-performance processors — requiring specialised compilers
- Shared-memory multiprocessor scalability — to make use of the server architecture
- Supporting large numbers of threads — providing service for many users simultaneously
- Continuous availability — running for months at a time
- Rapid response — serving requests within tight time deadlines, which places restrictions on the type of garbage collection which can be used
- Efficient library usage — server applications are often built from library components, which must be efficient
- Graceful degradation — slowing down under increase in workload instead of crashing