Fighting A Losing Battle

Nowhere in the IA-32 architecture does its enduring popularity show more than in the instruction set. What was once a simple accumulator-based architecture in the days of the 8080 has grown through the years into a vast and complicated array of instructions. IA-32 has become a RISC-like chip with numerous bolt-on extras and a bewildering array of addressing modes.

When approaching such a landscape as a Java developer, it is very tempting to revert to type and start writing classes as if the more structure you code, the simpler the problem will get. This approach would be fine, if not ideal, were we developing a disassembler. In such a system with so much object creation, there is also inevitably a large amount of garbage collection.

This results in a double speed penalty. Not only do we suffer the overhead of large amounts of object allocation, but we also suffer from frequent garbage collections. In a modern generational garbage-collected environment (of which the Sun JVMs are an example), small, short-lived objects are created in the young generation and almost all young-generation collection algorithms are stop-the-world. So a decoder with large amounts of object churn will suffer poor performance not only from unnecessary object allocation, but also from a large number of very short GC pauses while the collector cleans up all the object churn.

For this reason it was quite important to reduce object churn within the decoder that drives the interpreted execution. In the real mode decoder, this minimalist approach results in a 6,500-line class with just 42 instances of the "`new`" keyword:

- 4 × `new boolean[]` at class load time (`static final`)
- 3 × `new Operation()` for a rotating buffer
- 2 × `new int[]` for the expanding buffer in `Operation`
- 33 × `new IllegalStateException()` on exception paths

Once an instance of the decoder is created, the only necessary object construction is for expansion of the `int[]` buffers in `Operation`. Once the buffers have expanded, there is no object construction and no garbage collection, and therefore there is pause-less decoding.

The design of the decoder illustrates what we consider one of the important tenets of programming (and in this case, of Java):

> Just because you can, it doesn't mean you should.

In this case, just because a JVM can do automated garbage collection, it doesn't mean you are forced to exercise it. In a performance-critical section of code, approach object instantiation with caution. Beware of silent object instantiation done on your behalf with classes such as `Iterator`, `String`, and varargs calls.