electronic design

print | close

11 Myths About Embedded Java

Electronic Design
Simon Ritter
Thu, 2016-04-28 16:50

1. Java is slow.

Back in 1995, when Java was first released, the Java virtual machine (JVM) used very simple algorithms internally to run code. All bytecodes were interpreted; thread synchronization was primitive, and memory management simplistic.

With over 20 years of development, things have moved on so that the JVM now uses adaptive compilation and much more sophisticated thread management, with a variety of heap layouts and garbage-collection algorithms to choose from (including a pauseless collector in the Zing JVM from Azul).

Most code will now run as fast, if not faster in some cases (due to optimizations that can be applied by the JVM, but not the compiler) than natively compiled code.

Related

Java 8 Arrives

Q&A: Zulu Embedded Builds on OpenJDK

Prototyping With Java On Cortex-M3/M4

2. Java is dead.

The death of Java has been forecast almost since it was first released, but it shows no sign of dying or even declining in its popularity. A look at things like the TIOBE index or the RedMonk Programming Language Rankings consistently shows Java as the number one or two language used by developers.

3. Java is open source.

Strictly speaking, Java is not open source (because Java is ultimately just a trademark). What was the Sun (and now the Oracle) implementation of the Java Development Kit (JDK) was released under a GPL license (with Classpath exception) at the end of 2006. This is the OpenJDK project and, since Java SE 7, has been the reference implementation for the Java SE specification, as defined through the Java Community Process (JCP).

4. Java does not suffer from memory leaks.

In languages like C and C++, the developer is responsible for all memory management, both allocating and explicitly signaling that it's no longer required (through calls like malloc and free). Java has automatic memory management. When an object is instantiated, the JVM allocates space for it on the heap. When the application e no longer has any references to an object, its space can be reclaimed by the garbage collector that runs iodically in the background.

The key point here is that, in order to be garbage-collected, all references must be removed. If an application maintains references to objects even when they're not required and keeps allocating more objects, this will have the same effect as a memory leak in the traditional sense. Eventually, the JVM will run out of free memory to allocate to new objects.

5. Multi-threading in Java is hard.

Before Java SE 5, this would have been a fair comment. To write cooperative multi-threaded code, you only really had four APIs to work with (from the Thread class): wait, sleep, interrupt, and notify. This was very hard to get right.

To make life easier, the Concurrency utilities (designed by Doug Lea, Professor of Computer Science at New York University) were added to the standard class libraries. This provided higher-level abstractions such as semaphores, mutexes, read-write locks, and atomic operations. Since then, some new APIs have been added, as well as the introduction of the fork-join framework for specific, recursively decomposable problems.

In JDK 8, with the addition of the stream API, it's now possible to make stream operations parallel simply by changing one method call from stream to parallelStream. It doesn't get much simpler than that.

6. Java will be replaced by Scala.

When Scala was first released, many people thought that it would quickly be adopted as a replacement for Java. After all, it didn't have any of the baggage of backwards compatibility that Sun and Oracle had worked so hard to maintain. It compiled to bytecodes and class files so that it could take advantage of all the great benefits of the JVM. It was functional. What more could you want?

Well, it seems that most developers were (and still are) quite happy with Java. With the recent changes in JDK 8 providing a functional style of programming through Lambda expressions and the Stream API, Java has shown it can continue to evolve to meet developer's needs. According to RedMonk's Programming Language Rankings, Scala is number 14; the TIOBE index puts it way down in 30th place.

7. Java is too big for embedded applications.

In 1996, the Java class libraries contained a little over 200 classes. JDK 8 has over 4000, which is excellent for developers because they don't have to write their own list class or find a third-party library that does it for them. The downside to all of this great functionality is size: A full Java Runtime Environment (JRE) needs over 40 MB of storage space, which is too big for many embedded applications.

In JDK 8, steps were taken to address this with the introduction of three compact profiles, the smallest of which only requires a little over 10 MB of storage. In JDK 9, scheduled for release in Spring 2017, the JDK will be fully modularized so that application developers can select only the modules they need to run their application.

8. Java user interfaces are horrible.

The Abstract Windowing Toolkit (AWT) and Swing go right back to the late 1990s, and it's certainly true that things have moved on considerably from bland, gray rectangular boxes for a UI. However, although not part of the Java SE specification, there's the very powerful JavaFX UI toolkit. Originally developed by Sun, it was taken over by Oracle and is shipped as part of its standard JDK binary distribution. This was also made open source and can either be built for specific embedded projects, or third-party binaries can be used.

Java suffers from big pauses for garbage collection.

It all depends on what your application does and how you configure the JVM in terms of heap size and garbage-collector settings. It's quite possible to have many applications run indefinitely without any noticeable pauses. For applications that require low-latency guarantees, JVMs like Zing from Azul provide truly pauseless garbage collection.

10. Java applications take longer to develop.

Some people think that Java is too verbose and uses too much "boilerplate" code. Whilst it is true that, as a language, Java does require more text than some other languages, this isn't necessarily a disadvantage. Languages like Perl are very terse and really don't require any boilerplate code at all. However, I defy most normal developers to write a Perl script that runs to more than a couple of pages and be able to understand what it does six months later. Developers spend lots of time maintaining code. Having code that's readable and easy to understand (like Java) makes that job a lot easier.

With modern IDEs like NetBeans, IntelliJ and Eclipse developers have to spend far less time typing in code than they did back in the days of vi and emacs. Templates and automatic code completion make developers far more productive.

Lambda expressions and API's streams also helped to eliminate some of the unnecessary boilerplate code in Java. Expect to see more eliminated in future releases.

11. Java is like JavaScript.

The naming of JavaScript was one of the biggest mistakes ever in computer software marketing. Aside from both being imperative programming languages along with a little bit of syntax, they have almost nothing in common. It's better to say that Java is to JavaScript as car is to carpet.

Looking for parts? Go to SourceESB.

Source URL: http://electronicdesign.com/dev-tools/11-myths-about-embedded-java