

The Evolution of JAVA Technology for Embedded Systems

Jon Hoskin, the Chief Technology Officer of Insignia Solutions, looks into the challenges faced when using Java in embedded systems. Size, performance and predictability are the three key areas under scrutiny. In order to look at the current situation more objectively, "Java technology" is separated into two aspects: 1) specification, and 2) implementation. It is proposed that the current specifications for Java platforms are not at fault, but rather, most current implementations are not viable for many embedded systems. The article begins with an introduction to the Java platform before going onto examine the key challenges with references to size/functionality trade offs, dynamic compilation and precise garbage collection.

Some embedded systems developers have concluded that current implementations of the Java specifications are either too big, too slow, too unpredictable, and/or functionally incomplete for their use. This article discusses alternative ways to implement Java platforms to effectively address these legitimate concerns.

Sun Microsystems estimates there are more than 700,000 developers using Java technology today. While there are many documented success stories regarding the use of Java platforms for corporate or enterprise applications, there are few public illustrations of Java technology for embedded systems development. In order to look at the current situation more objectively, we will separate the term "Java technology" into two aspects: 1) specification, and 2) implementation. As a longtime developer and provider of virtual machine technologies, it is this author's opinion that the current specifications for Java platforms (including Enterprise Java, PersonalJava, and EmbeddedJava; see Figure 1) are not at fault, but rather, most current

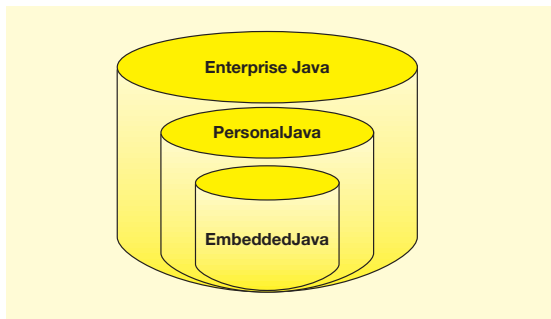


Figure 1. Java Specifications

implementations, particularly for PersonalJava and EmbeddedJava, are not viable for many embedded systems. The article begins with an introduction to the Java platform that will orient readers who are new to the subject, and possibly serve as a refresher for those already acquainted with Java technology.

JAVA TECHNOLOGY DESIGNED FOR THE INTERNET

Sun Microsystems introduced the Java platform in 1995 as a new programming language and runtime environment ideally suited for Internet-related applications. Building on its long-standing corporate theme, "The Network Is the Computer", Sun recognized that more powerful microprocessors were beginning to be used in many consumer devices, and increasingly, they have been connected to networks. Network connectivity began with workstations and personal computers, but was quickly followed by printers, scanners, copiers, and various other types of office equipment. More recently, newer devices such as personal digital assistants, Web televisions, pagers, smart cell phones, and even wristwatches have all been enhanced with microprocessors and connected to networks. Prior to Sun's introduction of the Java platform, the network was viewed primarily as a vast system for storing and serving up relatively static information, and the phrase "The Network Is the Virtual Disk Drive" was probably more fitting. Today, Java technology promises to extend the usefulness of networks by providing an efficient means for storing and distributing dynamic and extensible functionality to networked computing devices.

The Java architecture is comprised of several distinct but inter-related technologies, each of which is defined in specifications from Sun Microsystems. These technologies include the Java programming language, the Java virtual machine, and the Java API (see Figure 2). We will briefly discuss the Java programming language and the Java API before exploring the detailed implementation alternatives for the Java virtual machine. Together, the Java programming language and APIs define the "interfaces" between the developer and the Java virtual machine. Each Java virtual machine implementation must be able to execute Java programs, but the way in which this occurs is left to the discretion of the developer of the Java virtual machine itself.

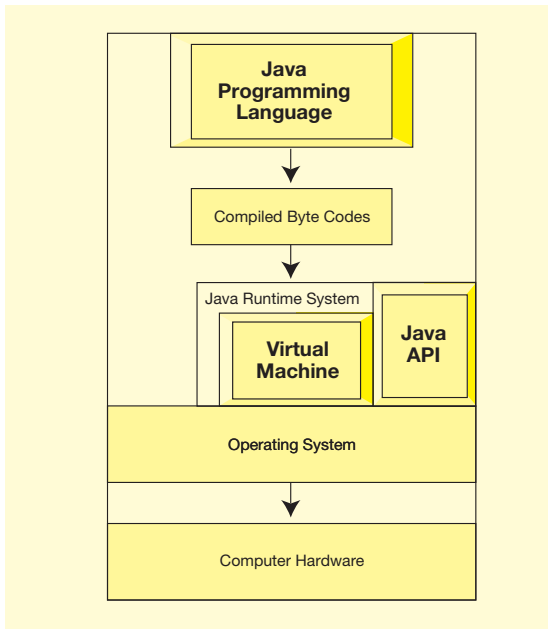


Figure 2. The Java Architecture

OVERVIEW OF THE JAVA PROGRAMMING LANGUAGE

The Java programming language inherits much of its syntactical rules, and therefore familiarity, from C/C++. In order to make Java technology more robust, however, some fundamental changes were made to the language definition. For example, the Java language does not permit direct access to memory nor is pointer arithmetic part of the language. To directly access memory, for example, developers may write a "native method" in C and call it from the Java application via the Java Native Interface (JNI). The Java language supports a true implementation of an array object. Unlike C/C++, the Java runtime performs array bounds checking to insure the application does not attempt to access any element beyond the end of the defined array. The Java language also prevents implicit type-casting to objects of dissimilar types, especially when precision might be lost (e.g. float typecast to an int). Although these features of the Java language may be viewed by some as restrictions in the language, there is emerging proof from actual development projects that Java technology can reduce overall development time and improve the reliability of the application – a key concern for embedded developers. Viewed at a higher level, Java technology is a pure object-oriented language. For example, the "Hello World" program can only be implemented as a class object (see Figure 3). Projects implemented with Java technology should realize the benefits of code reuse and a higher degree of maintainability over time. The key element of the Java language that is highly relevant to networks, is its size as an executable. Since Java applications are intended to be distributed easily across networks, it was important to maintain a compact representation of the code. Java class files entered by the developer are compiled into the efficient and portable bytecode instruction set which is the representation executed by the Java virtual machine. Rather than requiring an

```

package jeode.examples;
/* HelloWorldApp.java */
import java.lang.System;
class HelloWorldApp {
/**
 * "HelloWorld!" program.
 */
public static void main
 (String args[ ]) {
 // Write to stdout.
 System.out.println("Hello World!");
 }
}
  
```

Figure 3. Source code for "Hello World"

independent set of run-time libraries, like C/C++, multi-threading is inherent to the Java language.

The Java API can be described at either the individual detailed level or as a higher level collection of functionality. The hierarchy of definitions works something like this: at the highest level, everything can be referred to simply as "Java technology". Below that, Sun has defined some natural sub-setting of Java technology described previously as specifications for Enterprise Java, PersonalJava, and EmbeddedJava APIs. These specifications are intended to define the various functional elements and class libraries that are most likely to address the requirements of certain categories of applications. The objective is to help ensure that a Java application written for one category of computing device will be able to run on another computing device in the same category. These specifications for Java runtime generally require a functionally equivalent implementation of the Java virtual machine and specify a suitable subset of the Java class libraries. There are roughly 12 primary class "packages" defined (see Figure 4), which collect and organize similar and relat-

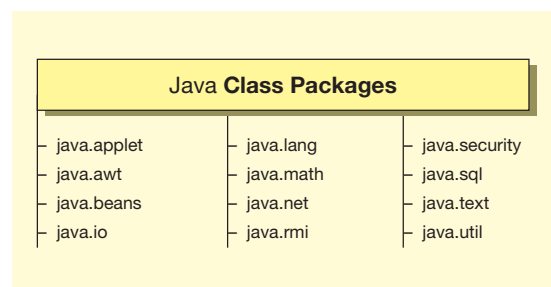


Figure 4. Java Class Packages

ed class libraries. Each package contains multiple interfaces and class libraries, each of which in turn contains multiple fields and class methods (see Figure 5). The Enterprise Java API is specified to contain all of the defined Java class libraries. The PersonalJava API is smaller, but still requires its complete specification to be implemented. The EmbeddedJava API, on the other hand, allows configuration of all of the class libraries.

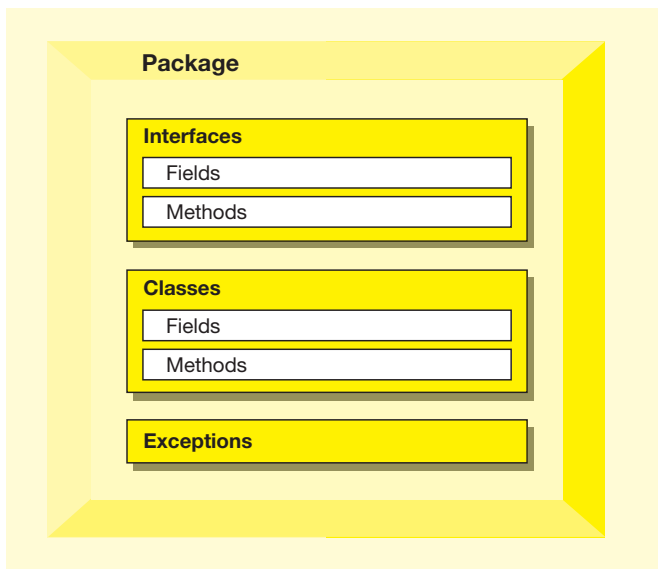


Figure 5. Java Package Contents

The vendors of Java virtual machines have taken two different approaches in providing Java class libraries. Some vendors supply only a subset of the Java class libraries that are termed "optional". Other vendors supply all of the class libraries and allow the developer to select which optional libraries are required for their applications. When it comes to comparing the memory footprint of the various Java platform implementations, you'll need to take care that you are making an apples-to-apples comparison including the class libraries.

EVOLUTION OF THE VIRTUAL MACHINE

The virtual machine (VM) is not a new concept or practice, although the state-of-the-art in computing power has made new applications for virtual machines more viable. More than 20 years ago, IBM pioneered much of the early development of virtual machine technology. The VM operating system for IBM computer systems is one of the best examples of the first implementations of the virtual machine concept. IBM's objective was to logically partition a single physical computer system into multiple virtual machines to create the appearance that each user had his or her own complete and separate computer system. The solution was to develop an abstraction that provided an exact duplicate of the underlying machine. Each user ran their own virtual machine completely isolated from all other virtual machines so there were no security problems. This application of virtual machine technology was well suited to the time-sharing requirements of large mainframe computers. More recently, a new application of virtual machines has come into fashion as a means of solving system compatibility problems. For instance, there are many applications available for DOS/Windows on Intel-based systems, and users of non-Intel based systems would like to be able to run some of these applications. The solution was to create a PC virtual machine for the non-Intel-based systems, despite resource constraints. A DOS/Windows

application is run on this virtual machine, and its Intel opcodes are translated into the native instruction set. (See sidebar story for an example of a commercial PC virtual machine.)

As with all virtual machines, the Java virtual machine defines an abstract computer. Its specifications define the functionality that every Java virtual machine must provide, but allow almost unlimited freedom to the designers of each implementation. For example, each Java virtual machine can use any technique to execute Java bytecodes. In fact, the Java virtual machine can be implemented in software or hardware, or varying degrees both. This flexibility in the specification for the Java virtual machine was intended to allow for implementation on a wide variety of computers and embedded devices.

CLASS FILE LOADING AND EXECUTION

At the top level, a Java virtual machine's primary purpose is to load Java class files and execute them (see Figure 6). The Java class loader subsystem is respon-

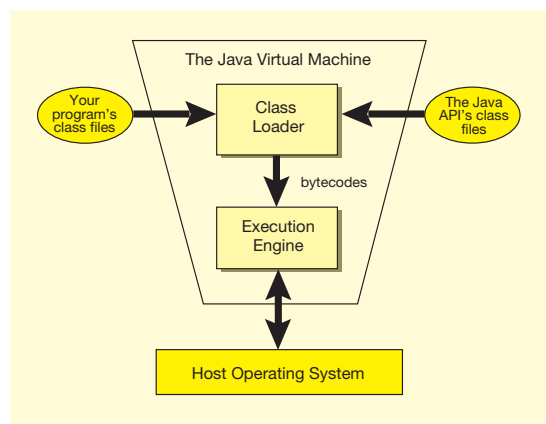


Figure 6. Java Virtual Machine

sible for locating and importing the bytecodes for classes. In addition, it must also verify the correctness of loaded classes, allocate and initialize memory for the class fields, and provide partial resolution of symbolic references. Depending on a particular application's requirements, or the network environment, some or all of the Java class loader subsystem may not be required. For example, Java class files that are loaded over a network down to a Java platform-enabled computing device are generally assumed to be "untrusted". Since it would not be wise to allow the Java application from an unknown source to run on your computing device without being validated, the verifier is a critical component of this Java virtual machine (see Figure 7). On the other hand, if you only download Java applications from your corporate server, the "screening" that is normally performed by the verifier may not be required, and the verifier itself can be configured out of this implementation of the Java virtual machine. At the

A JAVA ENVIRONMENT FOR EMBEDDED SYSTEMS

Developers have expressed their desire to use Java technology for embedded systems since the Java platform was introduced four years ago. Up until now, however, they have had concerns about the viability of Java technology because current implementations were either too big, too slow, or too unpredictable for their application needs. In March this year Insignia Solutions of Fremont, California, released Jeode 1.0, its implementation of the Java specification for embedded systems, and its Embedded Virtual Machine (EVM), which address many of these issues.

For the past ten years, Insignia has been delivering PC virtual machines to users of non-Intel platforms, enabling them to run DOS and Windows applications. Thinking back ten years or so, you will recall that the Macintosh computer in those days had relatively limited compute resources (e.g. Motorola 68000 CPU, 1/2 to 1 MB RAM, and 10-20 MB disk drive). Although Insignia did not know it at the time, this challenge of developing an efficient PC virtual machine for the resource constrained Mac was the beginning of Insignia's "training" in building Java virtual machines suitable for embedded systems.

FULLY COMPATIBLE JAVA TECHNOLOGY

A few companies have attempted to address the issue of "too big" by removing functionality from their implementation of the standard Java specifications. Unfortunately this has two problems: 1) the Java platform principle of "write once-run anywhere" is violated, and 2) embedded systems have a very wide range of functional needs, and it is impossible for the Java platform vendor to guess what is required in all cases. Insignia's approach to solving this problem is to provide a completely functional version of Java technology and provide developers with tools to configure and tune the Jeode EVM for their specific embedded applications (see Figure 1). By using these tools, the developer is able to optimize size, functionality, and performance (see Figure 2). Depending on the functionality selected, and required performance objectives, the developer

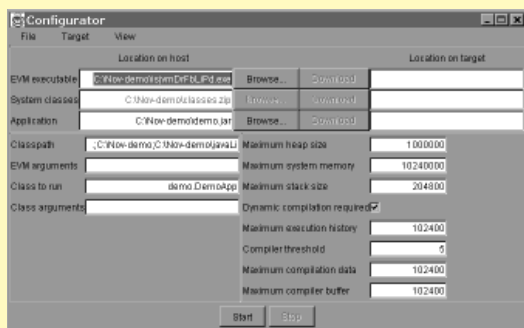


Figure 1. Jeode Configurator

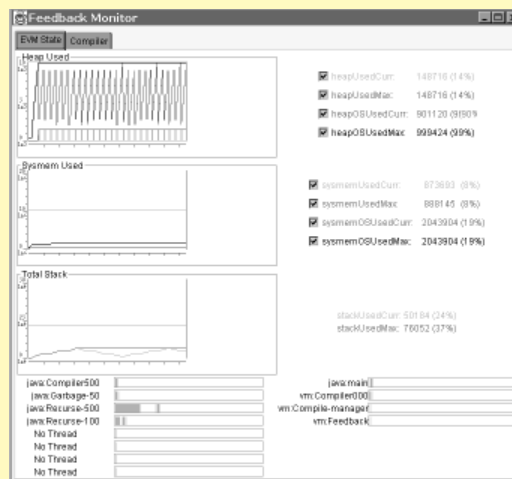


Figure 2. Optimizing EVM size, functionality, performance, and predictability

is able to configure an EVM that runs in as little as 330 KB, and scales up to a full configuration at 2.7 MB of ROM, including all Java class libraries and required dll's.

DYNAMIC COMPILATION FOR FASTER EXECUTION

To address the issue of "too slow," Insignia has leveraged its ten years of experience in delivering *adaptive dynamic compiler* technology. This technology differs from *JIT* technology in that it requires less memory, requires no disk storage or virtual memory, and dynamically compiles only the code that represents the current performance bottlenecks in the application. The remainder of the application code will continue to be interpreted. As the application runs, the Jeode EVM determines which code segments are executing most frequently, then compiles and stores them into the configured amount of code buffer memory. Once the allotted code buffers are used, the Jeode EVM may recycle the buffers to optimize performance in the given footprint. For embedded systems, it is critical this dynamic compilation process not block the main application. Therefore, the adaptive dynamic compiler operates as a thread and will be preempted by any higher priority threads in the system. Based on in-house tests, the adaptive dynamic compiler provides an average of six times the performance of an interpreted-only Java virtual machine, and runs in roughly one-quarter the memory footprint of a JIT-based Java virtual machine (see Figure 3).

PRECISE CONCURRENT GC FOR MORE PREDICTABLE BEHAVIOR

The third concern expressed by embedded systems developers is that current implementations of the Java specifications are too unpredictable for

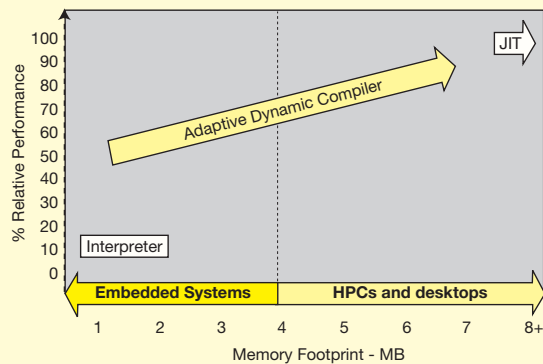


Figure 3. Optimizing Size and Performance

their applications. Predictability is largely determined by the “garbage collection” (GC) strategy that is used to manage memory. Insignia has implemented *precise* garbage collection technology that is fully concurrent, and provides memory compaction to eliminate memory fragmentation. The garbage collector runs as a preemptible thread and its priority can be set by the application. Unlike a *conservative* garbage collector, the precise garbage collector will free up all unreferenced objects, and avoids memory leaks. Insignia believes the key to appropriate memory management for embedded systems is to provide the proper core garbage collection technology and give the application control over when this process executes.

To further improve predictability, the amount of memory used by the Jeode EVM can be bounded, and the usage of memory can adapt dynamically depending on the current needs of the application. For example, the garbage collector can shrink the heap to make more system memory available for dynamic compilation. In addition, each of the Jeode EVM components will gracefully cope with any failure to acquire more memory. If a memory request fails, for example, the Jeode EVM may throw an exception that can be appropriately managed, rather than having the system crash.

PLATFORMS SUPPORTED

Insignia’s Jeode software is currently available for Windows CE and NT, VxWorks, and LINUX operating systems. It is supported on x86, ARM, Hitachi SH, MIPS, and PowerPC processors. Other OS/processor platforms may be made available by customer request. In addition, source code and porting kits are available for customers that require other platforms for their embedded applications.

For further information on the Jeode software call +44 1628 539 500 or visit the companies web site at <http://www.insignia.com>.

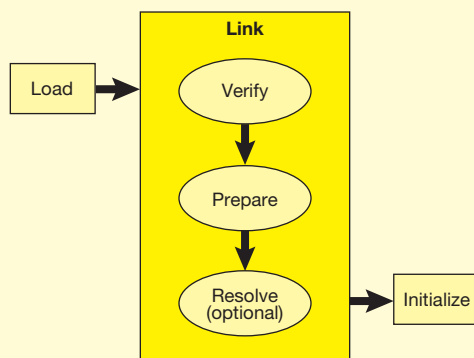


Figure 7. Class Loader Subsystem

far end of the functionality spectrum, the entire class loader subsystem may be completely removed. Of course, this also eliminates the opportunity to run dynamic Java applications on the computing device, but this may be totally appropriate for certain classes of embedded devices. Similar to the class libraries discussed previously, the degree of completeness will affect the memory footprint of the various Java virtual machine implementations, and vendors of Java virtual machines have taken slightly different approaches in

providing this functionality. Sun Microsystems, and its licensees of Java technology, provide a complete set of functionality and allow the developer some flexibility in configuring the required Java virtual machine. Other non-licensed vendors may choose to implement and provide whatever portions of the subsystem they feel is required. Given the wide diversity of requirements for embedded systems, it would seem a full-featured Java virtual machine with the maximum flexibility to configure would be most ideal.

During execution, there are two aspects of any Java virtual machine implementation that has the most impact on the suitability of Java technology for the specific application: 1) the methods used for executing the bytecodes, and 2) the management of system resources – primarily memory. Unlike the previous examples of Java class libraries and the Java class loader, where functionality may be reduced in order to shrink the size of the memory footprint for the Java virtual machine, the subsystems for the execution of bytecodes and the management of memory must be functionally complete for all implementations. Three important attributes about the final application, or perhaps even the suitability of Java technology for the application, will be determined by how well the vendor of the Java virtual machine has implemented bytecode execution and memory management. The

first attribute, which we have discussed considerably, is the memory footprint. The second, which is mostly affected by the methods used for bytecode execution, is performance. The final attribute of the application, which is subject to how well the vendor has implemented the memory management subsystem, is predictability. Taken together then, bytecode execution and memory management have the greatest effect on the suitability of Java technology for embedded development – is it small enough, fast enough, and predictable enough in its behavior for my application?

COMPILED VERSUS INTERPRETED TECHNOLOGY

The default method for executing bytecodes by a Java virtual machine is interpreted execution. Whereas C/C++ applications are compiled to a native instruction set, and are stored in a single executable file, Java applications are compiled to Java bytecodes, and are stored in separate class files for loading and execution by the Java virtual machine. When the Java application is run, the Java virtual machine loads the class files and interprets the bytecodes as a stream of instructions. Typically, an interpreted-only method for execution by a Java virtual machine will result in performance which is 10 to 15 times slower than an equivalent natively compiled C/C++ program. In the plus column, an interpreted-only version of a Java virtual machine generally has a relatively small memory footprint – implementations are typically 1/2 to 2 MB of ROM. However, there are other techniques that can improve the speed of bytecode execution. For example, just-in-time (JIT) compiling can enhance application performance up to 10 times over interpreted-only execution. Rather than only interpreting bytecodes, each time the Java virtual machine encounters new bytecodes for the first time, it invokes the JIT compiler to compile that code to native instructions. These native instructions are then stored by the Java virtual machine and are reused the next time the code is required by the Java application. Performance comes at a cost, however. JIT technology typically requires four times the amount of memory of an interpreted-only version of a Java virtual machine, compiles all code that it encounters, and often requires a disk and virtual memory for paging the compiled code segments. Because of this, JIT-based execution is suitable for computing platforms that have powerful CPUs, substantial system memory (> 32 MB), and generally a fast disk drive to support swapping. It is not suitable, however, for the vast majority of embedded systems which are generally far more resource constrained.

DYNAMICALLY COMPILED VERSUS A JIT

A bytecode execution method more suitable for embedded systems is adaptive dynamic compilation. Similar to a JIT, adaptive dynamic compilation utilizes on-the-fly compilation technology to improve the performance of bytecode execution. Unlike a JIT, however, adaptive dynamic compilers are generally smaller in size, more selective of the bytecodes that they compile to native code, do not require virtual memory, and

adapt to the available system memory. The size of an adaptive dynamic compiler is typically measured in terms of 10's of Kbytes of memory for the compiler itself. During interpreted execution of the bytecodes, the Java virtual machine will monitor the application and determine where the execution bottlenecks reside. The Java virtual machine will then invoke the adaptive dynamic compiler, which may be implemented as a thread, to compile the segment of bytecodes, which are executing repeatedly. Depending on the implementation, the adaptive dynamic compiler may compile the entire class, a single method, or only a block within a method. The resulting native instructions are stored in memory only for fast access. The Java virtual machine may then find and invoke compilation for the next most frequently executed code and this process is repeated until all available code buffers have been used. Because the dynamic compiler is adaptable, when it encounters code that is executing at greater frequency than some previously stored native instructions, it will compile these new bytecodes and store them in a code buffer containing less frequently used code. The user can configure the amount of system memory used for compilation, and the size and number of code buffers used for the compiled native instructions. Applications may also be given explicit control over the adaptive dynamic compiler's thread priority during execution for greater predictability, providing more suitable behavior for embedded systems.

RELIABILITY AND PREDICTABILITY ISSUES

The efficient management of memory is particularly critical for embedded systems where predictable behavior is required. The Java virtual machine plays the central role in managing memory. In fact, the application itself only makes requests for memory allocation for new objects and does not explicitly release unnecessary memory. The release of memory is managed automatically by the Java virtual machine. Each Java virtual machine implements a "garbage collector" to provide memory management. There are several different implementations of garbage collectors, and some are more suitable for embedded systems than others (see Figure 8). One of the primary concerns of embedded systems developers is that if the garbage collector runs a complete batch cycle, and cannot be preempted, this will render Java technology unsuitable for the embedded real time application (i.e. unpredictable). Some implementations of garbage collectors are referred to as "incremental". This suggests that they can recycle memory in a stepwise fashion rather than garbage-collecting all memory in a single pass. Although this garbage collector may not be preemptible, this type of implementation should lead to more predictable behavior than a batch garbage collector. Since the incremental garbage collector may still block the application, however, the risk remains that the pauses will impact the application. If the garbage collector is defined as "concurrent", this suggests it performs garbage collection incrementally, is fully preemptible, and should provide the most predictable behavior of all.

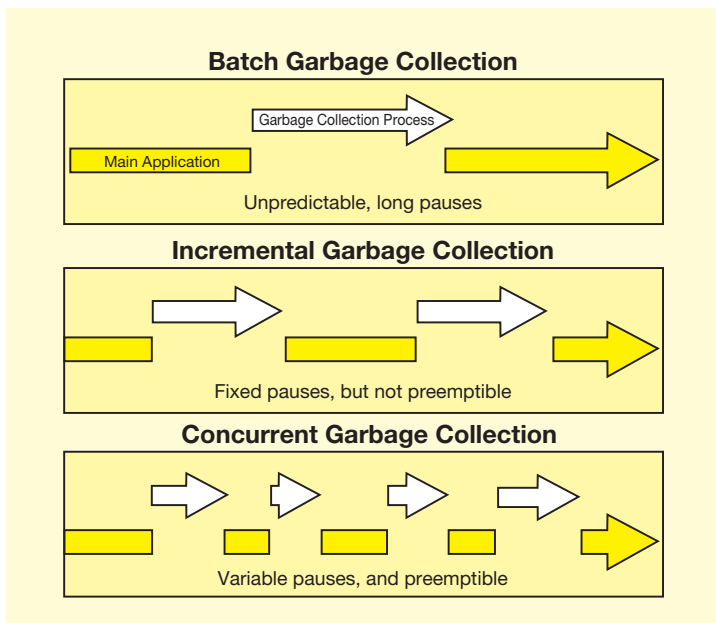


Figure 8. Garbage Collection Implementations

In addition to its mode of execution (i.e. batch, incremental, or concurrent), garbage collectors may perform their duties with varying degrees of efficiency and effectiveness. There are two basic approaches to separating live objects from garbage: "reference counting" and "tracing". Reference counting is an older garbage collection technique, and involves maintaining a reference count for each object on the heap. Essentially, the reference count is incremented for each new reference to a given object and decremented when the reference to an object goes out of scope. All objects with a reference count of zero can be garbage collected. However, among other disadvantages, reference counting suffers from the overhead of incrementing and decrementing the reference count each time the object is referenced.

PRECISE VERSUS CONSERVATIVE GARBAGE COLLECTION

A more suitable method for Java virtual machines is the tracing garbage collector technique. Tracing garbage collectors trace the object reference tree starting with root nodes. Objects that are encountered during the trace are marked, and after the trace is complete, unmarked objects can be garbage collected. During the tracing process, garbage collectors may either use either a "precise" or a "conservative" approach to identifying references to objects. The conservative garbage collector may not distinguish between genuine object references and look-alikes (for example, 32-bit integers), and although this approach may be slightly faster, it may also lead to memory leaks. A precise garbage collector, on the other hand, understands the differences between true object references and look-alikes, and frees all unreferenced objects appropriately. One final important aspect of garbage collection to consider is whether the garbage collector has a strategy to combat heap fragmentation. Given the limited amount of memory available for most

embedded systems, a concurrent, precise, compacting garbage collection strategy should provide for the most predictable system behavior.

IN SUMMARY

The specifications for the various Java platforms have generally been well conceived. If not perfect at first, they are certainly evolving in the appropriate direction to address the requirements of the identified classes of computing devices. Now that the specifications are stabilizing, and the vendors are getting down to the business of delivering suitable implementations of the Java specifications, this author expects to see an increase in the use of Java technology for embedded systems. With the implementation of the appropriate technologies described in this article, many embedded developers may soon discover that the benefits of the Java platform can be delivered in a

package that is small enough, fast enough, and predictable enough for their next embedded development project. ■

Dr. Hoskin was appointed to his current position in May 1999, having been a member of the Insignia team since 1992. As engineering director of the company's virtual machine business, he was instrumental in the design, development and successful release of the company's Jeode platform. As a senior technical advisor, Dr. Hoskin represents Insignia Solutions at important technical consortia, and functions as the company's technical ambassador to the industry. His mission is to scrutinize technology and market trends and to guide the long-term direction of the company's products.

As director, Dr. Hoskin led Insignia's research team through the entire engineering process from the investigation and prototype phases, through the development of the Jeode platform, eventually overseeing a team of more than 40 engineers. Dr. Hoskin also held key managerial and technical positions in the research and development group for SoftWindows, the company's first virtual machine-based product line. Prior to joining Insignia Solutions, he held positions at several leading-edge systems software companies in the United Kingdom. Dr. Hoskin earned a doctor of philosophy degree in mathematics from Wolfson College, Oxford University, and is known throughout the technical community for numerous articles published in technical journals. He earned an advanced degree in mathematics from Churchill College, Cambridge University, and received a Bachelor of Arts degree First Class from the University of Warwick, Warwickshire, graduating with honors.