# SMALL-SCALE JAVA VIRTUAL MACHINES

Lauri Aarnio

**Abstract:** Java 2 Platform, Micro Edition (J2ME) provides a comprehensive application development platform for creating networked products and applications for the consumer and embedded market. J2ME is targeted for devices such as mobile phones, pagers and PDAs having limited hardware and network resources. The C and K virtual machines are the heart of J2ME technology. They are Java virtual machines designed from the ground up for small-memory, limited-resource and network connected devices. This paper introduces the J2ME architecture in brief and then focuses on the C and K virtual machines. Although Java Card technology is not part of the J2ME, the Java Card virtual machine is also discussed here. The paper introduces the most important technical features and limitations of these small-scale Java virtual machines.

## 1 INTRODUCTION

Sun Microsystems has grouped Java into different editions: Enterprise, Standard, Micro and Java Card (Figure 1). The regular Java is Java 2 Standard Edition (J2SE). J2SE includes the Java Virtual Machine (JVM), plus a core set of Java runtime classes and their associated, native libraries. J2SE is designed to be used in writing applications that run on a desktop computer and have a sophisticated graphical user interfaces. Java 2 Enterprise Edition (J2EE) builds on top of J2SE and adds classes for server-side code development. These add-ons are for example servlets, Java server pages and Enterprise JavaBeans. Java 2 Micro Edition, or J2ME for short, and Java Card are attempting to make Java smaller, so that Java applications could be written for devices with limited hardware and network resources. Java Card enables applications written in Java to be run on extremely limited devices, smart cards. In this paper, the focus will be on J2ME and Java Card technologies, and especially on their virtual machines.

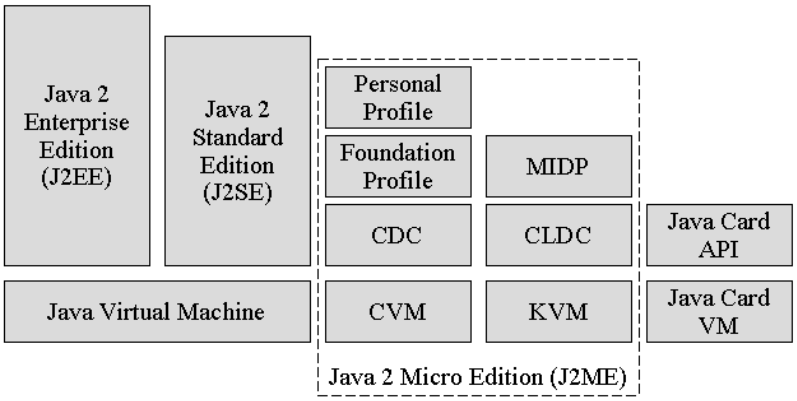


Figure 1. Java 2 platform editions [1].

For years, Sun Microsystems has been blasting out the Java mantra: "Write once, run anywhere." However, the reality of developing Java applications for devices with limited resources has tarnished that dream. Relatively little effort has been spent making Java implementations smaller. However, there is a large number of consumer devices and embedded systems that would benefit from a small Java implementation supporting the Java language. Developing Java applications to run in devices such as mobile phones, pagers, PDAs and smart cards has returned Java to its original target environment – small spaces. The Java language was initially targeted towards consumer devices but over time the Java platform evolved more and more towards the needs of desktop and enterprise computing. Java libraries have become larger and more comprehensive to meet the needs of the large applications. This evolution has made the libraries too large and unsuitable for the majority of small, resource-constrained devices.

A Java virtual machine mediates between the application and the underlying platform, converting the application's bytecode into machine-level code appropriate for the hardware and operating system being used. In addition to governing the execution of bytecode, the virtual machine handles related tasks such as managing the system memory, providing security against malicious code, and managing multiple threads of program execution. In January 1998, the Spotless project [2] was started at Sun Microsystems Laboratories to investigate the use of the Java language in resource-constrained devices. The goal of the project was to build a Java virtual machine that would fit in less than one-tenth of the typical size. The new Java virtual machine should have the following characteristics:

- Small size
- Portability
- Ease of use and readability of the source code

The Spotless project turned out to be successful. The product version of the Spotless virtual machine is nowadays called the K Virtual Machine (KVM) which is introduced in more depth in section 2.2.

The full implementation of the Java virtual machine is far too large to fit on even the most advanced resource-constrained devices available today. Even if the Java virtual machine could be made to fit, there would not be much space left over for class libraries and application code. Small spaces require equally small virtual machines that are functionally limited (subsets of the standard Java). In the following chapters three small-scale Java virtual machines: CVM, KVM and Java Card VM, are discussed (Figure 2).

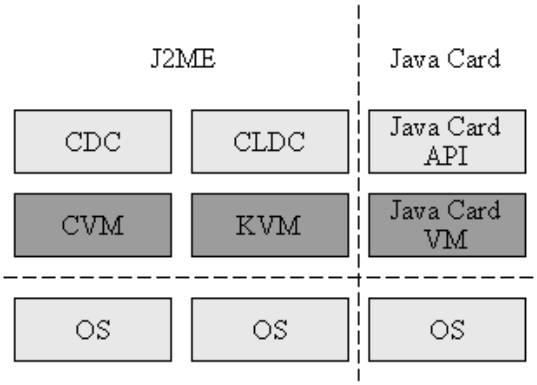


Figure 2. Small-scale Java Virtual Machines [3].

# 2 JAVA 2 MICRO EDITION

## *2.1 Overview*

To meet the demand for Java applications in the rapidly growing consumer and embedded markets, Sun Microsystems has extended the scope of Java technology with Java 2 Micro Edition (J2ME). J2ME is targeted for devices such as mobile phones, pagers and PDAs having limited hardware and network resources.

As the J2ME environment is itself very small, industries have great flexibility in defining only what they need for a class of device. Sun Microsystems has partnered with a number of companies (Ericsson, Motorola, Nokia, NTT DoCoMo, etc.) to develop new J2ME classes called configurations and profiles that describe useful APIs. J2ME software layers are represented in Figure 3.

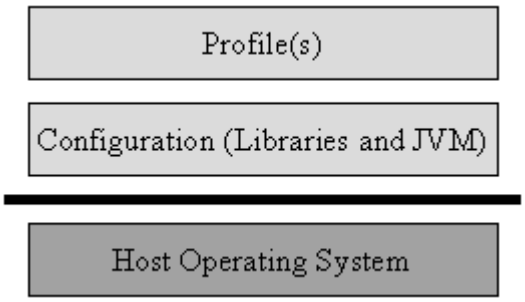


Figure 3. Software layers in a J2ME device [1].

A configuration is the most basic unit of a J2ME runtime environment. It consists of a Java virtual machine, set of core classes, and the native code required by the core classes. Each configuration is designed to run on a specific family of devices and defines minimum requirements for those devices. Device manufactures often produce a configuration tied to their platform (processor, memory, peripheral devices). J2ME has a capability to receive not just the application code, but also libraries that form part of the platform itself. This enables a J2ME environment to be dynamically configured. Configuration is performed by server software running on the network. There are currently two J2ME configurations available: Connected Limited Device Configuration (CLDC) and Connected Device Configuration (CDC) (Figure 4). CLDC is a small subset of J2SE classes designed for devices with constrained CPU and memory resources. Typically, these devices run on either a 16- or 32-bit CPU and have 512 KB or less memory available for the Java platform and applications. CLDC does not define any user interface classes because such things are the responsibility of profiles, not configurations. CDC is a superset of CLDC and it is aimed at higher-end devices with more robust resources. These devices run on a 32-bit CPU and have 2 MB or more memory available.

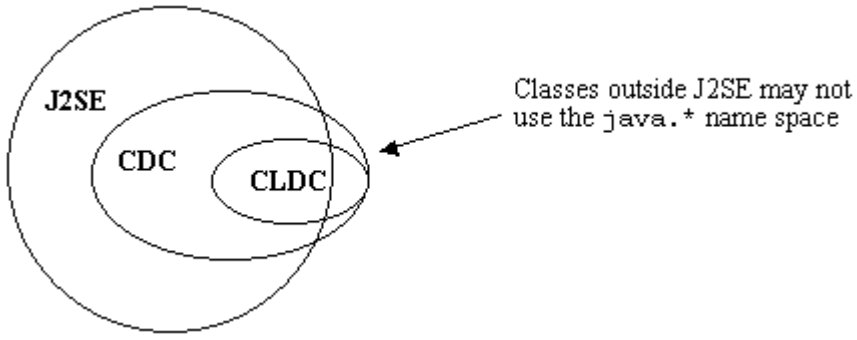


Figure 4. Relations between J2ME configurations and Java 2 Standard Edition [4].

A profile is a set of classes that extends the capabilities of a configuration in a specific way, for instance, to let developers build user interfaces or to provide network connectivity. Profiles are, at least in theory, independent of a particular hardware configuration. There are several profiles available for both the CLDC and the CDC. A profile targeted at the wireless market and utilizing CLDC can retain a very small footprint, consume little power, and provide as much capability as is needed for handheld devices. Devices that require more capable environments can receive profiles that provide additional functionality and are based on CDC. All profiles share a J2ME base as well as the ability to safely download code onto a device and configure the Java environment.

A J2ME implementation can also have vendor-specific classes, but these classes are not officially part of any profile or configuration. By using these classes, the portability will be sacrificed, but it may be the only way to gain access to specific features of the device.

### *2.2 CLDC and KVM*

The Connected, Limited Device Configuration (CLDC) technology defines targeted Java platforms which are resource-constrained devices having a memory budget in the range of 160 to 512 KB. The CLDC technology is composed of the K Virtual Machine (KVM) and core class libraries.

The K Virtual Machine started out as a research project in Sun Microsystems Laboratories. Initially called the Spotless VM [2], it was used to test limited memory Java Virtual Machine configurations [5]. The KVM was designed to overcome three key technical challenges: reducing the size of the virtual machine and class libraries themselves, reducing the memory utilized by the virtual machine during execution, and allowing for components of the virtual machine to be configured to suit particular devices. The 'K' in KVM stands for kilo. It was so named because its memory budget is measured in kilobytes. The KVM is written from scratch in C (around 25 000 lines of code), so it can be easily ported onto various platforms for which a C compiler is available. At the moment, the KVM has been ported to more than 25 devices by Sun Microsystems' Early Access licenses. The key features of the KVM are [6]:

- Reduced virtual machine size – KVM is currently only 50-80 KB in its standard configuration, depending on the target platform and compilation options.
- Reduced memory utilization – Only 128 KB of memory is needed to run simple Java applications on a device. More typical applications need a total memory budget of 256 KB, of which at least 32 KB is used as runtime heap space.
- Performance – KVM is able to run effectively on 16-bit processors clocked as low as 25 MHz, and scale up to much more powerful 32-bit

processors. KVM runs anywhere from 30 to 80 percent of the speed of JDK 1.1 VM without a Just-In-Time compiler (JIT).

- Portability – KVM has a highly portable architecture that reduces system dependencies to a minimum. Non-portable code elements (including functions that obtain information about the host computing system) have been minimized and limited to a single file. Even multi-threading and the garbage collector have been implemented in a completely platform-independent fashion.

On desktop implementations, the KVM can be run from the command line, just like J2SE. For devices that do not have such a user interface, KVM provides a reference implementation of a facility called the Java Application Manager (JAM). The JAM serves as an interface between the operating system and the virtual machine, and it is able to load applications (JAR files) via network. The JAM reads the contents of the JAR file and launches the KVM with the main class as a parameter. The Java Code Compact (JCC, also known as 'ROMizer') utility is supported by the KVM. This utility allows classes to be linked directly in the virtual machine, reducing the startup time of a virtual machine. KVM provides a simple, non-moving, single-space mark-and-sweep garbage collector. It is optimized for small heaps (32 – 512 KB). The collector is handle-free, because direct object references are always used rather than indirect. KVM does not support Java Native Interface (JNI). All native code calls from the virtual machine must be linked directly into the virtual machine at compile time. It is not recommended to use native code calls since the consequences of mistakes are frequently severe. KVM does not support AWT, Swing, or any other high-level graphics libraries. Graphics are currently implemented by calling platform-specific graphics functions.

### *2.3 CDC and CVM*

The Connected Device Configuration (CDC) is one of the key components in the J2ME environment. CDC includes the C Virtual Machine (CVM) and basic class libraries to support Java applications on consumer electronic and embedded devices. CVM is designed for devices needing the functionality of the Java 2 virtual machine feature set, but with a smaller footprint. CVM offers superior savings in memory usage and runs effectively on 32-bit processors. A typical target device for the CDC and the CVM has a 32-bit processor, network connectivity and from 2 to 16 MB of total memory (including both RAM and flash or ROM) for the storage of the virtual machine and libraries.

The CDC class library is the minimal set of APIs needed to support the CVM. The CDC has the following basic Java packages [7]:

- java.lang – Virtual machine system classes
- java.util – Underlying Java utilities
- java.net – UDP Datagram and File I/O
- java.io – Java File I/O

- java.text – Bare minimal support for I18n
- java.security – Minimal security and encryption for object serialization

The CVM is very portable, real time operating system –aware, and suitable for embedded devices. It allows devices to map directly to native threads, and it can run Java classes out of ROM. In addition, CVM has advanced features such as a precise memory system, generational garbage collector, and fast Java synchronization. The advanced memory system of CVM has short average garbage collection pause times, full separation of virtual machine from memory system and pluggable garbage collectors. CVM has been written in C with very little assembly code and it is highly portable having a well-documented porting layer that supports multiple porting options. CVM is able to perform synchronization operations with just a few machine instructions, and without consuming additional lock resources from the operating system. CVM supports dynamically loaded classes and "ROMable" classes that can be executed out of ROM. These both make the memory requirements for classes quite small. The reduction in memory footprint for classes is around 40 percent compared to Java 2 Standard Edition. CVM features reduced-footprint native stack usage, which is achieved by a static analysis of the virtual machine code. Due to clearly defined and well-documented interfaces, it is fairly easy to add new features to CVM.

### *2.4 J2ME Devices*

There are already several Java-enabled mobile phones and PDAs on the market. Table 1 lists some of them.

Table 1. Examples of mobile phones and PDAs supporting the J2ME [8].

| Brand/Model | Telecommunication | Comments |
|---|---|---|
| Nokia Communicator 9210 | GSM900, GSM1800 | - |
| DoCoMo FOMA N2001 (made by NEC) | 3G (WCDMA) | Only in Japan. |
| DoCoMo F503i (made by Fujitsu) | i-mode (PDC) | Only in Japan. |
| Motorola i85s | iDEN | Only in USA, Canada, Brasil, Israel and Middle East. |
| Palm V (PDA) | - | Runs on PalmOS |

*"In 2002, Nokia will deliver more than 50 million Java-enabled phones; more than 100 million delivers by the end of 2003"*

*Pekka Ala-Pietilä*
*Nokia President, JavaOne, 2001*

# 3 JAVA CARD

## *3.1 Overview*

Java Card technology provides architecture for open application development for smart cards, using the Java programming language. A smart card is a plastic card that looks like a credit card but has an embedded microprocessor. The microprocessor is capable of doing computer-like things such as running programs, processing input and output, and storing data. But unlike most computers, a smart card does not incorporate a display or a keyboard, and it has no power supply. A typical smart card has an 8-bit processor and only a few kilobytes of ROM and RAM. Smart cards represent one of the smallest computing platforms in use today.

Smart cards are far too resource-limited devices to support the full functionality of the Java platform. Therefore, the Java Card platform supports only carefully chosen, customized subset of the features of the Java platform. This subset includes features that are well suited for writing applications for smart cards and other small devices while preserving the object-oriented capabilities of the Java language. Table 2 lists some notable supported and unsupported Java language features. Since Java Card technology does not support all the features in Java language, the Java Card virtual machine can only support the features that are required by the language subset. The Java Card virtual machine is discussed in detail in the next chapter.

Table 2. Supported and unsupported Java features in the Java Card platform [9].

| Supported Java Features | Unsupported Java Features |
|---|---|
| • Small primitive data types: boolean, byte, short<br>• One-dimensional arrays<br>• Java packages, classes, interfaces and exceptions<br>• Java object-oriented features: inheritance, virtual methods, overloading and dynamic object creation, access scope, and binding rules<br>• The int keyword and 32-bit integer data type support are optional. | • Large primitive data types: long, double, float<br>• Characters and string<br>• Multidimensional arrays<br>• Dynamic class loading<br>• Security manager<br>• Garbage collection and finalization<br>• Threads<br>• Object serialization<br>• Object cloning |

The Java Card technology has several advantages. First of all, Java Card is platform independent meaning that applets that comply with the Java Card API specification will run on smart cards developed using the Java Card Application Environment (JCAE). Secondly, applications can be installed after the smart card has been issued. This provides smart card deliverers an ability to dynamically respond to their customers' changing needs without having to issue a new card. Java Card also allows multiple applications to run on a single smart card and the Java Card API is compatible with international standards such as ISO7816.

### *3.2 Java Card Virtual Machine*

A primary difference between the Java Card virtual machine (JCVM) and the "standard" Java virtual machine is that the JCVM is implemented as two separate parts, as shown in Figure 5. The on-card part of the JCVM includes the Java Card bytecode interpreter. The Java Card converter runs on a PC or a workstation. The converter is the off-card part of the JCVM. These two part together implement all the virtual machine functions.

Development of a Java Card applet begins as with any other Java program. A developer writes one or more Java classes and compiles the source code with a Java compiler. The applet can be tested on a simulator before installing it to the target environment (Java Card technology enabled device). Once the applet works correctly on a simulator, the class files comprising the applet are converted to a CAP file (converted applet) by using a Java Card converter. During the conversion from class files into a CAP file, the converter performs tasks that a Java virtual machine would perform at class-loading time. The idea behind the preprocessing is to keep the JCVM on a smart card as small as possible. The converter performs the following tasks [9]:

- Checks for Java Card language violations
- Initializes static variables in the classes
- Resolves symbolic references
- Optimizes bytecode
- Allocates storage and creates virtual machine data structures to represent classes.

An installation program on the smart card receives the contents of the CAP file and prepares the applet to be run by JCVM. JCVM itself need not load or manipulate the CAP file; it need only execute the applet code found in the CAP file. The division of functionality between the JCVM and the installation program keeps them both small. The interpreter provides runtime support of the Java language model and thus allows hardware independence of applet code. The interpreter executes bytecode instructions, controls memory allocation and object creation, and plays an important role in ensuring runtime security.

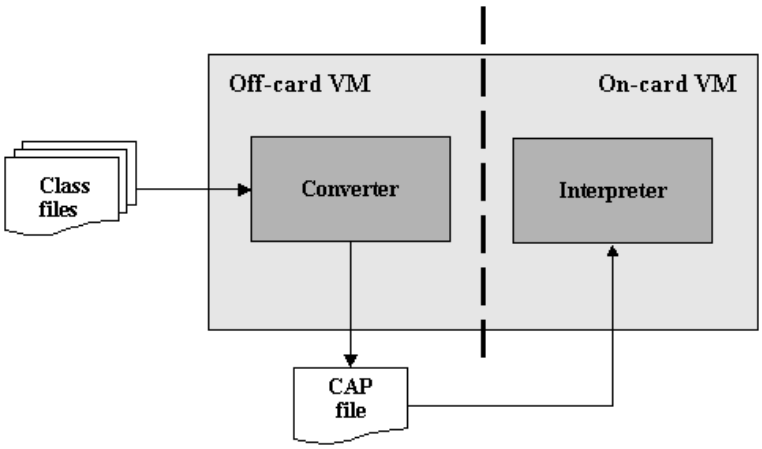


Figure 5. Java Card virtual machine [8].

There are a number of limitations in the JCVM. The limitations are spoken here as maximums from the developer's perspective. The converter is responsible for checking that the following maximum values are not violated [10]:

- A package can contain at most 255 public classes and interfaces.
- A class can implement at most 15 interfaces, including interfaces implemented by superclasses.
- An interface can inherit from at most 15 superinterfaces.
- A class can have at most 256 public or protected static fields or methods.
- A class can implement a maximum of 128 public or protected instance methods.
- Class instances can contain a maximum of 255 fields.
- Arrays can hold a maximum of 32767 fields.
- The maximum number of local variables that can be used in a method is 255 (an integer data type is counted as occupying two local variables).
- A method can have at most 32767 bytecodes.
- Switch statements are limited to a maximum of 65536 cases.

## 4 CONCLUSIONS

Small-scale Java virtual machines take Java back to its embedded-system roots and offer developers an opportunity to write Java applications for completely new platforms. Developing a Java application for a resource-limited device is not just copying a J2SE application to a new platform. The J2SE virtual machine is simply far too large to fit into these devices, and therefore smaller versions of virtual machines are needed. Both KVM and CVM can be thought of as shrunken versions of the J2SE virtual machine. Java Card VM is different from these two being an extremely small virtual machine targeted to smart cards. Small-scale Java virtual machines are still relatively immature technologies but they have a great potential in providing totally new application development platforms. The device manufactures are already taking advantage of these new Java platforms, and Java-enabled mobile phones and PDAs, for example, are already in use in Asia and Europe. Customized Java applications allow users to tailor these devices to meet their specific needs.

## References

[1] RIGGS, R., TAIVALSAARI, A., VANDENBRICK, M., Programming Wireless Devices with the Java 2 Platform, Micro Edition, Addison-Wesley, 2001.
[2] BUSH, B., SIMON, D., TAIVALSAARI, A., The Spotless System: Implementing a Java System for the Palm Connected Organizer. Sun Microsystems, 1999.
[3] Java 2 Platform, Micro Edition – Datasheet. Sun Microsystems, 2001.
[4] Java 2 Platform Micro Edition Technology for Creating Mobile Devices – White Paper. Sun Microsystems, 2000.
[5] BENATAR, D., PETSCHULAT, S., TicTacToe on the KVM. Java Report, February 2001.
[6] The K Virtual Machine. http://java.sun.com/products/cldc/kvm, Sun Microsystems, 2001. (October 17, 2001)
[7] Connected Device Configuration (CDC) and the Foundation Profile – Technical White Paper. Sun Microsystems, 2001.
[8] JavaMobiles.com. http://www.javamobiles.com (November 21, 2001)
[9] ZHIQUN, C., Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison-Wesley, 2000.
[10] Java Card 2.1.1 Virtual Machine Specification. Sun Microsystems, 2000.