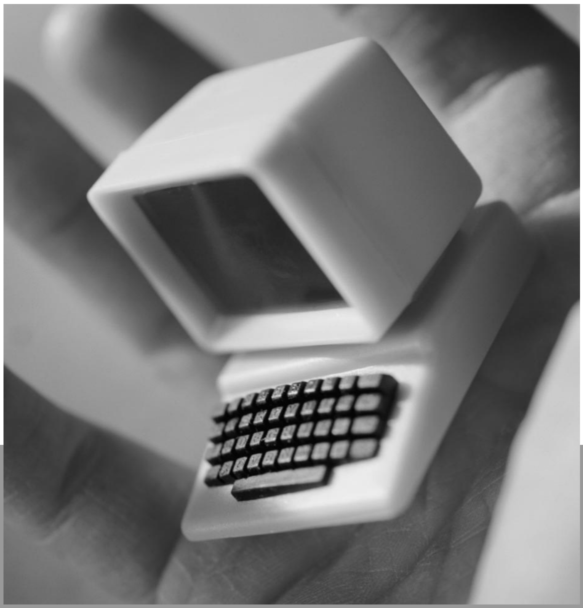
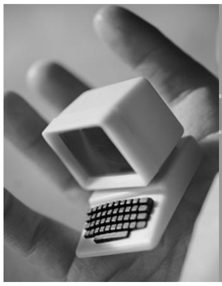
PART TWO

Java 2 Micro Edition (J2ME) Specifications





CHAPTER 4

Java 2 Micro Edition (J2ME)

At the JavaOne Conference in June 1999, Sun Microsystems announced a new edition of the Java 2 platform: the Java 2 Micro Edition (J2ME). The purpose of the Micro Edition is to enable Java applications to run on the small computing devices that we discussed in the first chapter. Although the J2ME announcement was interesting, what really caused a stir was the preliminary release of a new Java virtual machine (JVM) that could run simple Java programs on Palm devices. The initial frenzy, however, was soon tempered by the need to develop formal specifications for J2ME and to finish the work on the new virtual machine. It would take almost a year for the first production release of J2ME to become a reality, however.

In this chapter, we discuss what J2ME is and what it is not. We look closely at one of its key pieces, a new virtual machine optimized for small devices called the KVM. We also look at how other related pieces of Java technology—things such as EmbeddedJava and Personal-Java—fit into the J2ME puzzle.

Introducing the Micro Edition

Sun Microsystems' Web site describes J2ME this way:

> . . . Java 2 Platform, Micro Edition is a highly optimized Java runtime environment targeting a wide range of consumer products, including pagers, cellular phones, screenphones, digital set-top boxes and car navigation systems.

The key phrase from this description is "highly optimized Java runtime environment." We must emphasize that J2ME does not define a new kind of Java but instead adapts Java for consumer products that incorporate or are based on some kind of small computing device. A Java application written for the Micro Edition will also work with the Standard Edition and even the Enterprise Edition, assuming the APIs it uses are available in each edition. There are constraints, but the architecture of Java never changes. Writing Java code that runs unchanged in all three editions is possible. Cross-edition portability is not normally a requirement, however, because what you are really interested in is cross-device portability. In other words, will the application work correctly on a specific set or family of devices? Let's look at some of the key features of J2ME and see why the answer to this question is a simple but resounding yes.

A New Virtual Machine

The Java 2 Standard Edition (J2SE) platform currently supports two different virtual machines: the so-called classic virtual machine and the newer HotSpot virtual machine. Swapping out the classic virtual machine and replacing it with a HotSpot virtual machine gives J2SE programs an immediate and measurable performance boost without making any other changes to the runtime environment. If a new virtual machine can be designed from the ground up in order to boost performance, why not design a virtual machine to run in a constrained environment? That is exactly what J2ME does with the KVM, which is short for Kuaui VM (an early name).

The KVM is a completely new implementation of a Java virtual machine, an implementation optimized for use on small devices. The KVM accepts the same set of bytecodes (with a few minor exceptions) and the same class-file format that the classic virtual machine does. We will discuss the KVM in greater detail shortly.

You should understand that the Micro Edition is more than just the KVM. In fact, the classic virtual machine can still be used with J2ME. Thus, J2ME supports two different virtual machines:

- The classic virtual machine for 32-bit architectures and large amounts of memory
- The KVM for 16-bit or 32-bit architectures with limited amounts of memory

Possibly, future versions of J2ME will support other virtual machines as well.

New and Changed Classes

As you might imagine, one of the keys to getting Java to run on a small device is to reduce the size of the runtime classes installed with the runtime environment. The Micro Edition performs this task by removing unnecessary classes to form a new set of core classes. A similar pruning occurs within the classes themselves (unnecessary or duplicate methods are removed). What is left is a true subset of the J2SE runtime classes.

Specific implementations of the Micro Edition are also free to ROMize the core classes. In other words, any classes that are provided as part of the basic runtime environment can be stored by using the virtual machine's internal format instead of the normal class-file format. The virtual machine must still have the capacity to read user-defined classes in the normal class-file format, however.

A stripped-down runtime is not particularly useful if there is no way to interact with the user or with other external devices. J2ME also augments the runtime environment by defining new classes that are suitable for smaller devices. Some of these classes replace similar classes in J2SE while others provide new functionality not found in the other editions.

Configurations and Profiles

Realizing that the one-size-fits-all principle used with the Standard Edition does not work on small devices, the Micro Edition uses configurations and profiles to customize the Java runtime environment.

A configuration defines the basic J2ME runtime environment as a virtual machine and a set of core classes that run on a family of devices that have similar capabilities. Two configurations are currently defined: the Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC), both of which we will discuss in the next chapter. Each configuration provides a minimal set of features that all devices in the configuration must support. The CDC uses the classic virtual machine while the CLDC uses the KVM (as shown in Figure 4.1).

A profile adds domain-specific classes to a particular J2ME configuration. Unlike configurations, which are device-oriented, profiles are more application-oriented. They provide classes that are geared toward specific kinds of applications (or more accurately, specific uses of devices). Examples are user interface classes, persistence mechanisms, messaging infrastructures, and so on. Figure 4.2 shows how profiles and configurations work together in order to provide a complete Java runtime environment. Several profiles have been defined or are in the process of being defined, and we will discuss them in Chapter 6, "Profiles."

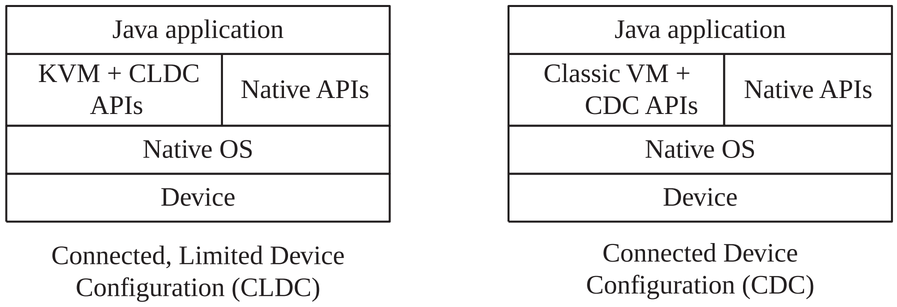


Figure 4.1 The two J2ME configurations.

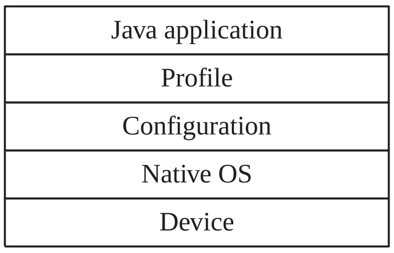


Figure 4.2 Profiles.

Building Blocks

As this book was being written, a revision of J2ME was being developed under the Java Community Process that would replace configurations with building blocks. A building block subsets classes from J2SE or J2EE for use with J2ME. Individual profiles would include appropriate building blocks in order to share more functionality with the J2SE runtime classes. The proposal, JSR-68, states that over time, the concept of a configuration will be replaced with building blocks but that profiles will still exist. Because building blocks are not yet part of an approved specification, this book uses the term *configuration*.

Profiles are the double-edged sword of J2ME. They define important and necessary functionalities, but they also limit your application's portability to the other editions of Java or even to other profiles. Such restrictions are not unexpected or unreasonable, however, because the need to run a single application on all possible platforms is quite rare. Some would argue that it is not even possible to do right. What J2ME enables you do to, however, is use Java for all of the pieces of your application—from the database server (more and more databases support Java directly as a programming language) through the middle tier and down to the smaller devices. Although no Java developer really wants another set of application programming interfaces (APIs) to learn (it is hard enough keeping track of all of the new classes that are added to the Standard Edition), profiles keep the developer using a familiar language and familiar development tools. This is a key feature of all three editions of Java.

The KVM

Many perceive the KVM to be the heart of the Micro Edition. Strictly speaking, of course, this situation is not the case. The KVM is one of two virtual machines provided by Sun Microsystems for use with J2ME. Also, while most implementations of J2ME will use or port the Sun virtual machines, vendors are also free to implement their own virtual machines as long as they adhere to the J2ME specifications and pass Sun's compatibility tests. RIM's BlackBerry system, which we discuss in

Chapter 11 ("Java for BlackBerry Wireless Handhelds"), uses its own implementation, for example.

If the KVM is not the heart of J2ME, however, it is certainly an important part of it. As mentioned, when Sun first announced J2ME at the JavaOne Conference, an early version of the KVM was made available to attendees (preloaded on a Palm V that they could buy as a show special). The KVM soon found its way onto the Internet, causing considerable excitement in both the Palm and Java development communities. Finally, there was a way to run Java on a mainstream personal digital assistant (PDA) with promises of more device support to come. It makes sense, then, for us to spend some time looking at the KVM before exploring J2ME configurations or profiles in any detail.

The Spotless System

Development of the KVM started a number of months before the JavaOne Conference. In fact, the KVM originated as a project in Sun Laboratories, the research arm of Sun Microsystems. This progenitor of the KVM was known as the Spotless project, and its goal was to write a fully functional Java runtime environment—the Spotless System—for the Palm series of connected organizers.

Complete details about the Spotless System are found in a Sun Labs technical report published in February 1999, which is available online at www.sun.com/research/spotless. We will summarize the main points here, but we encourage you to read the full report if the history of the KVM interests you.

The primary motivation for the development of the Spotless System was to create a small but complete Java execution engine that was suitable for use on small devices. The emphasis was on size, not speed, and on the portability of the execution engine source code.

The Spotless System was not the first attempt at getting Java to run on small devices. Sun had three formal initiatives already: Java Card, EmbeddedJava, and PersonalJava—that addressed the problem. We will briefly summarize these initiatives later in this chapter, so for now, all we need to know is that the Spotless team found them all unsuitable for a number of reasons. Thus, they developed a new runtime environment.

As we saw in the previous chapter, writing a Java virtual machine is not difficult, nor is the resulting virtual machine particularly large. As both we and the Spotless team discovered, the size of the Java runtime environment derives mostly from its runtime libraries. The Spotless team conducted extensive and detailed analyses of the Java 1.1 core run time and concluded that it would be simpler to build a new, smaller set of classes from scratch rather than trying to adapt the existing library. The reasons are simple. There are too many classes and too many interdependencies between classes. Untwining the classes and reworking their innards is more complex than writing new classes based on the runtime library specification.

By starting from scratch, then, the Spotless team was able to carefully choose which classes to include in its own run time and even which methods to include. The team also chose to merge certain classes and to eliminate various subclasses, all in order to reduce the total number of classes and their cumulative effect on the runtime size. This latter decision would prove to be particularly controversial, as we will see shortly.

The result was a complete runtime environment that could run in the constrained setting of a Palm device (and in theory, other similar environments). At some point, a decision was made to transform the research project into the product that we now know as the KVM. The Spotless project still continues as a separate research initiative, but it and the KVM are evolving along different paths.

Early KVM Controversy

The first public release of the KVM was at the 1999 JavaOne Conference. After the initial excitement subsided, those who took a close look at the KVM and tried to port their Java programs to it soon ran across a number of important road blocks.

The most obvious limitation was program size. The static size of a program—basically, the sum of the class files—was limited to 64K. Runtime memory constraints were even more onerous. On a typical Palm device that was available at the time, there was about 20K of runtime memory available to an application after the virtual machine started. These limitations were not inherent to the KVM but were mostly due to the segmented memory architecture of Palm OS, which we will describe in more detail in Chapter 8, “Java for Palm Connected Organizers.” These restrictions have relaxed since then, but runtime memory constraints are still limiting when compared to a typical desktop computer.

A second road block was the decision not to implement the AWT. AWT is a set of APIs that Java programs use in order to generate Graphical User Interfaces (GUIs). Even Swing-based user interfaces eventually use AWT, so AWT is extremely important for client programs. They felt, however, that AWT was simply too large to work well in the restricted environment of the KVM, so the KVM team avoided it altogether. They understood, of course, that some kind of user interface library was necessary, so they implemented their own user interface objects that were simple and Palm-specific. Defining a cross-device user interface was left for later and more formal specifications. In other words, users had to be prepared to rewrite their user interface code whenever a formal specification (in this case, the Mobile Information Device Profile that we will discuss in Chapter 6) was defined and implementations of it were available.

kAWT: AWT for the KVM

Early adopters of the KVM were quite disappointed with the limited set of user interface components that it provided and proceeded to build their own set of components called kAWT (modeled after the AWT components with which they were familiar). kAWT is discussed in Chapter 8 and is included on the CD-ROM for this book.

The roadblock that caused the most controversy in the early days, however, was the Spotless team's decision to move and/or rename methods in the core classes. What it meant was that you could not take the logic sections of a program—the parts not directly involved with the user interface—and run them unchanged on J2SE. This situation somehow seemed more offensive than just dropping methods from those classes, because it broke the promise of Java compatibility.

The KVM Today

The KVM has been officially released as the reference virtual machine for the first J2ME configuration, which we will discuss in the next chapter. Developers can download the KVM from Sun Microsystems' Web site (as part of the configuration) and compile it for the Palm, Windows, or Solaris platforms (or port it to any other platform).

Although many changes were made to the KVM, the basic motivation for its existence has not changed: to provide a complete Java runtime environment for small devices. In other words, the KVM is a true Java virtual machine as defined by *The Java Virtual Machine Specification* (except for some specific, documented deviations that are necessary for proper functioning on small devices):

Long integer and floating point datatypes are optional. This makes sense, because operations involving these types often have to be simulated in software on a small device. Floating point operations are especially expensive without a dedicated coprocessor.

No object finalization. With no finalization, the garbage collector's job is made much simpler and less time-consuming.

No JNI support. The KVM has a native interface, of course, but this interface is not like JNI, which is more complex because it is a portable interface. In fact, the native methods in the KVM are compiled into the virtual machine and are not user installable.

Off-device class verification. Because class verification is an expensive and time-consuming operation, the KVM team came up with a way to move most of the work off the device and onto the desktop or server computer, where the class files are compiled. This step is referred to as preverification. All the device does is run a few simple checks on a preverified class file in order to ensure that it was verified and is still valid.

Multidimensional arrays are optional. Few applications use multidimensional arrays, so removing the necessary support saves a bit of space.

No user-defined class loaders. The only class loader that is available to applications running on the KVM is the system class loader provided by the KVM itself.

Features that are optional are defined in separate modules within the KVM source code. A particular J2ME configuration defines the exact set of features that it expects from a virtual machine, and that determines which of the optional modules will be included when a version of the KVM is compiled for a particular platform.

Note that specific use of the KVM is not necessarily required, because the KVM is a reference implementation of a Java virtual machine. Vendors who are shipping a J2ME configuration only need to provide a virtual machine that meets the requirements of the configuration. Starting with the KVM makes it easier, but there is nothing preventing a vendor from writing his or her own virtual machine from scratch.

Related Technologies

If you are not a newcomer to Java, you probably realize that J2ME is not Sun's first attempt at making Java work on small devices. Java Card, EmbeddedJava, and PersonalJava already existed by the time that KVM made its first public appearance. How do these technologies relate to J2ME?

Of the three, PersonalJava is the closest in spirit to J2ME. PersonalJava defines a subset of the core Java APIs (an optimized version of the Java runtime library) that can be combined with a fully-compliant (classic) virtual machine in order to produce a smaller Java runtime environment that is suitable for use on consumer devices. PersonalJava also defines some new APIs—of which the most important is the Truffle Graphical Toolkit, which defines user interfaces with a customizable look-and-feel (such as the Touchable look-and-feel for touch-screen devices). For more information about PersonalJava, refer to http://java.sun.com/products/personaljava.

The biggest difference between PersonalJava and the Micro Edition is that the latter supports a much broader set of devices, mostly by allowing specific changes to the Java virtual machine. As such, PersonalJava is really a subset of J2ME and is in fact being redefined as a J2ME profile, the Personal Profile. We will discuss profiles in Chapter 6.

If PersonalJava is being folded into J2ME, what about EmbeddedJava? At first glance, PersonalJava and EmbeddedJava seem similar. In fact, EmbeddedJava seems even more flexible than PersonalJava, because every class, method, and field of the Java runtime library is optional. Unlike PersonalJava or even J2ME, however, EmbeddedJava is specifically meant to be an unexposed, embedded runtime environment. In other words, only the implementer of the embedded system can use and write applications for the EmbeddedJava environment. EmbeddedJava defines a closed system—a black box whose contents and implementation are unknown and inaccessible to third-party developers. EmbeddedJava does not affect and is not affected by the Micro Edition.

For more information about EmbeddedJava, refer to http://java.sun.com/products/embeddedjava.

The remaining technology similar to J2ME is Java Card, an architecture for running Java programs on smart cards. A smart card is a set of electronic circuits—some memory with or without a central processing unit (CPU)—packaged as a thin device that looks like a standard credit card. Smart cards are constrained in the amount of data that they can store—too constrained even for the J2ME. To deal with these constraints, a significant number of changes had to be made to Java. Not only does Java Card define its own set of APIs, but a Java Card virtual machine even has a specification separate from *The Java Virtual Machine Specification*. As you can see, then, J2ME and Java Card have different goals. For more information about Java Card, refer to http://java.sun.com/products/javacard.

To summarize, then, both Java Card and EmbeddedJava can be viewed as complementary technologies to J2ME, while PersonalJava is being folded into J2ME. For a starting point on all of these technologies, refer to the Java Consumer and Embedded Technologies page at http://java.sun.com/products/OV_embeddedProduct.html.

Chapter Summary

In this chapter, we introduced J2ME, a new platform for Java programming. We outlined what goes into J2ME and how it compares to existing Java technologies. We also spent a bit of time on the history of the KVM, which is a key part of J2ME. We can now move on to examining J2ME configurations and profiles.