CHAPTER 1

Java 2 Micro Edition Basics

The Mobile Information Device Profile (MIDP) is just one part of a larger initiative to make Java work on small computing devices, the Java 2 Micro Edition (J2ME). Before starting our exploration of the MIDP, we need to step back and understand what J2ME is and how it evolved. This chapter provides you with a broad overview of J2ME and the MIDP's place in it, including the Connected Limited Device Configuration on which it is based. For a more comprehensive look at J2ME, refer to Eric Giguère's *Java 2 Micro Edition*, also part of John Wiley & Sons' Professional Developer's Guide series.

A Very Brief History of Java

Java's first incarnation was Oak, a language developed at Sun Microsystems for programming consumer devices. You can still find information about Oak on Sun's Web site by searching for references to the Green Project. Oak was ahead of its time, however, and instead became the more general-purpose language we now know as Java.

The Architecture of Java

Architecturally, Java has not changed much since its first release. A Java compiler transforms the Java programming language into a set of Java *bytecodes*. Bytecodes are instructions for an abstract computing machine referred to as a *virtual machine*, or VM for short. A Java VM, sometimes referred to as a JVM, interprets the Java bytecodes in order to run a Java program. A Java VM is thus often called an *interpreter*, although Java code can also be compiled straight into native machine binary code. Whether interpreted or compiled, Java bytecode execution must follow the steps and semantics described in *The*

Java Virtual Machine Specification (JVMS), or else you cannot call it Java. (Java is a trademark of Sun Microsystems, and anyone wishing to implement a Java runtime environment and call it such must obtain permission from Sun and pass a comprehensive set of compatibility tests.) The syntax and semantics of the Java language itself are described in a separate document entitled *The Java Language Specification* (JLS).

One of Java's inherent strengths is its portability—the capability to take a Java program and execute it on various operating systems without having to recompile or otherwise retarget the program for each operating system. This portability is achieved in several ways. First, both the JLS and the JVMS ensure that the types, bytecodes, and encodings used in Java are defined independently of the underlying operating system. Second, the binary encoding of a Java class—how the bytecodes are packaged at a class level—is defined by the JVMS, also in a machine-independent fashion. Third, a core set of runtime classes (and an associated set of platform-specific native code) abstract the interface between a Java program and the underlying operating system.

Java is also known for its security infrastructure. From the verification of class files to ensure the integrity of the generated bytecode to the use of class loaders and security managers, Java makes it possible to securely and safely download and execute third-party code of untrusted origin. This capability to download code across a network was arguably Java's most important feature when it was first developed and is being re-emphasized today with initiatives such as J2ME and Jini.

Unless the operating system is written in Java, of course, there has to be a way for Java programs to access the features of the native operating system. Java programs can call native code in a controlled manner through a native code interface. The current form of the native code interface is referred to as the Java Native Interface, or JNI for short. User-developed native code is rarely found, however, and can hamper portability and security.

Early Java

The first official release of Java outside Sun was called Java 1.0.2. The 1.0.2 actually refers to the version of the *Java Development Kit* (JDK) that included everything needed to develop and run Java programs on the Windows and Solaris operating systems. Even today, you will still refer to Java by the JDK version, although Sun now refers to it as the *Java Software Development Kit* (JSDK) and has separated the runtime-only portions of

Reading the Specifications Online

Although they are available in hard copy as well, both *The Java Language Specification* and *The Java Virtual Machine Specification* can be read online from Sun's Web site. The JVMS is at `http://java.sun.com/docs/books/vmspec/index.html`**, and the JLS is at** `http://java.sun.com/docs/books/jls/index.html`**.**

70899_Ortiz_CH01I 10/30/2001 3:30 PM Page 3

the JDK into a separate *Java Run-Time Environment* (JRE). Although strictly speaking, the JRE did not actually appear until version 1.1 of the JDK was released, we will use the term generically to refer to the runtime part of any Java platform.

Java 1.0.2 was notable primarily for two things. First, it defined an *Abstract Windowing Toolkit* (AWT) for the creation of portable *graphical user interfaces* (GUIs). Second, it defined *applets*, a way in which Web browsers use an embedded Java VM to safely run applications downloaded on the fly from untrusted Web sites.

While 1.0.2 was a good first attempt, it had a number of deficiencies, such as limited control over the user interface, a lack of internationalization and localization capabilities, and a restrictive security model. The next major release, Java 1.1, addressed a number of these issues and added many new features. The new features included a listener-based event model, object serialization, *remote method invocation* (RMI), *just-in-time* (JIT) compiling, and inner classes. There were also some optional pieces, like a new user interface toolkit called Swing (a set of enhanced AWT components) and a set of collections classes with more advanced data structures than those found in the `java.util` package.

Java 2

As work progressed within Sun Microsystems on Java 1.2, a decision was made to rebrand Java and to make major changes in the way Java was packaged and licensed. Java 1.2 became simply Java 2, although the JDK and JRE versions remained at 1.2. More importantly, however, the Java platform was split into three editions:

- *Java 2 Standard Edition* (J2SE) is for conventional desktop application development. Swing has been folded into the core Java classes, and a number of new classes have been added to enhance application development even more than what Java 1.1 offered.
- *Java 2 Enterprise Edition* (J2EE) is a superset of J2SE that is geared toward enterprise programming with an emphasis on server-side development using *Enterprise JavaBeans* (EJBs), web applications (servlets and JavaServer Pages), CORBA, and *Extensible Markup Language* (XML).
- *Java 2 Micro Edition* (J2ME) is a subset of J2SE that is geared toward embedded and handheld devices that cannot support a full J2SE implementation.

Although there is a certain amount of overlap, each edition targets a different kind of application developer. The sheer number of classes available to J2EE programmers—and the complexities of using those classes—stands in stark contrast to the much smaller set of classes available to J2ME programmers. On the other hand, J2ME programmers have severe memory and resource constraints to handle. Splitting Java 2 into three editions makes it possible for Java to evolve in different directions while staying true to the spirit of the language.

The Java Community Process

Although Sun is the ultimate authority for the Java platform, much of its work in defining and extending the platform is done through the auspices of the *Java Community*

70899_Ortiz_CH01I 10/30/2001 3:30 PM Page 4

Process (JCP). The JCP enables corporations and individuals to participate in the definition and revision of different parts of the Java platform. The process is fairly simple: A specification request (known as a JSR) is submitted with a specific proposal to extend the Java platform. If the JSR is accepted for development, an *Expert Group* (EG) is formed to define a formal specification for the JSR. The Expert Group consists of JCP members who have expertise in the area covered by the JSR and who volunteer their time and effort to develop the proposal with the interests of the larger Java community in mind. When ready, the specification is published for review by other JCP members and by the general public. The specification is revised based on reviewer comments before being voted on and accepted as a formal Java standard.

All J2ME standards are defined by using the Java Community Process. For more information about the JCP, see the JCP Web site at `www.jcp.org`. From there, you can get a list of all the JSRs that have been defined or are in the process of being defined, including the ones mentioned in this book.

Java 2 Micro Edition

J2ME enables Java applications to run on small, resource-constrained computing devices. It does not define a new language; rather, it adapts existing Java technology for handheld and embedded devices. Compatibility with J2SE is maintained wherever feasible. In fact, J2ME removes the parts of J2SE that are not applicable to constrained devices, such as AWT and other features. In this section, we briefly describe the key components of J2ME: configurations and profiles.

Configurations

A *configuration* defines the basic J2ME runtime environment. This environment includes the virtual machine, which can be more limited than the VM used by J2SE, and a set of core classes derived primarily from J2SE. The key point is that each configuration is geared toward a specific family of devices with similar capabilities.

Currently, two configurations are defined: the *Connected Device Configuration* (CDC) and the *Connected Limited Device Configuration* (CLDC). Both target connected devices—devices with network connectivity—whether it is a high-speed fixed link or a slow-speed wireless link. The CLDC targets the really small devices: cellular telephones, *personal digital assistants* (PDAs), and interactive pagers. As a group, these devices have important power, memory, and network bandwidth restrictions that directly affect the kind of Java applications that they can support. The CDC, on the other hand, targets devices that are less restricted, such as set-top boxes (devices that provide network-based computing features through a television) and car navigation systems. The line between the CDC and the CLDC is not distinct, because some high-end cellular telephones and PDAs can meet the requirements of the CDC—forcing the device manufacturer (the most likely provider of a Java runtime environment) to decide which configuration to support.

70899_Ortiz_CH01I 10/30/2001 3:30 PM Page 5

Note that the CDC is a superset of the CLDC: The CDC includes all of the classes defined by the CLDC, including any new ones that are not part of J2SE. The CDC includes many more core J2SE classes than the CLDC, however, which makes the CDC a more familiar and comfortable environment for experienced Java programmers. For the purposes of this book, though, you will have to learn to live within the restrictions imposed by the CLDC.

Perhaps the biggest difference between the CDC and the CLDC is that the former requires a full-featured Java virtual machine that is compliant with the one in J2SE. In other words, the CDC VM must support all the advanced features of a J2SE VM, including low-level debugging and native programming interfaces. Sun has released a new Java VM, the *Compact VM* (CVM), for this purpose—it is more portable and efficient than the standard VM.

Profiles

A *profile* extends a configuration, adding domain-specific classes to the core set of classes. In other words, profiles provide classes that are geared toward specific uses of devices and that provide functionality missing from the base configuration—things such as user interface classes, persistence mechanisms, and so on. Profiles are the double-edged sword of J2ME: While they provide important and necessary functionality, not every device will support every profile. In Japan, for example, NTT DoCoMo has released a number of Java-enabled cellular telephones based on the CLDC but with their own proprietary profile. Applications written for these devices will not work on cellular telephones that support the MIDP.

A number of profiles are defined or are in development. Besides the MIDP, which is based on the CLDC (we will discuss this topic in detail in the next chapter), the following profiles are or will be available:

- A *Personal Digital Assistant Profile* (PDAP) that extends the CLDC to take advantage of the extended capabilities of PDAs when compared to the simpler devices targeted by the MIDP.
- A *Foundation Profile* that adds additional J2SE classes to the CDC but no user interface classes. It acts as a foundation for building other profiles.

What about Java Card and EmbeddedJava?

J2ME is not Sun's first foray into the handheld and embedded device space. Although PersonalJava is being folded into J2ME as a CDC profile, what happens to Java Card and EmbeddedJava? Nothing; they remain as they are. Java Card adapts Java for use on smart cards, a very specialized environment that is not suitable for general-purpose programming. EmbeddedJava is more about licensing than defining a portable Java subset: EmbeddedJava licensees can pretty much choose which features of Java they want to support in their devices. The catch is that they cannot expose those features for use by a third party—only their own developers can write the applications that run on the device.

- A *Personal Profile* that redefines PersonalJava as a J2ME profile. The Personal Profile extends the Foundation Profile.
- An *RMI Profile* that adds RMI support to the CDC.

Multiple profiles can exist within the same configuration. Profiles can also build on each other—for example, the Personal Profile is an extension of the Foundation Profile. Expect more profiles to be developed as J2ME evolves.

The Connected Limited Device Configuration

To understand the MIDP, you must first understand the CLDC, the most minimalist of the J2ME implementations. The CLDC is defined by JSR-30 in the Java Community Process. For more information on the CLDC, refer to Sun's Web site at `http://java.sun.com/products/cldc`.

Requirements

The CLDC does not require many resources. It is meant to run on devices with 128K or more of non-volatile (persistent) memory and 32K or more of volatile memory. CLDC devices are required to have some kind of network connection (hence the term *connected device*), although it might be an intermittent, slow-speed connection. The configuration is for *limited* devices (devices that have severe limits on their computational power and battery life).

The CLDC defines a number of requirements for the Java environment. The first requirement is for full support of the Java language, except for a few differences. These differences are as follows:

- **No floating point support.** Floating point types or constants are not supported, and neither are the core J2SE classes that deal specifically with floating point values—classes such as `java.lang.Float` and `java.lang.Double`. Methods taking or returning floating point values are removed from all classes.
- **No object finalization.** To simplify the garbage collector's task, the `finalize` method is removed from `java.lang.Object`. The garbage collector will simply reclaim any unreferenced object. This action prevents unreferenced objects from "resurrecting" themselves and causing extra bookkeeping work for the garbage collector.
- **Runtime errors are handled in an implementation-dependent fashion.** Runtime errors are exceptions that are subclasses of `java.lang.Error` thrown by the virtual machine itself. The CLDC only defines three of these error classes: `java.lang.Error`, `java.lang.OutOfMemoryError`, and `java.lang.VirtualMachineError`. Any other error condition is handled by the VM in an

70899_Ortiz_CH01I 10/30/2001 3:30 PM Page 7

implementation-dependent manner, which usually means terminating the application.

The second requirement is for full virtual machine support, except for these few differences:

- **No floating point support.** CLDC devices might have no native support for floating point operations. As such, the VM does not support floating point constants or any of the bytecodes that involve floating point types.
- **No finalization and no weak references.** These are left out to simplify the garbage-collection algorithms.
- **No support for JNI or reflection or any low-level interfaces that depend on them.** In particular, there is no support for object serialization in the CLDC. Note that a VM *can* have a native interface, a debugging interface, or a profiling interface, but it is not required and it does not have to be a standard J2SE interface.
- **No thread groups or dameon thread.** Threads are supported, but thread groups or dameon threads are not. The VM can choose to implement threads by relying on the operating system or by performing its own context switching.
- **No application-defined class loaders.** An application cannot influence how classes are loaded. Only the runtime system can define and provide class loaders.
- **Implementation-defined error handling.** As mentioned, any runtime errors that are not specifically defined by the CLDC are handled in an implementation-specific manner.
- **Class verification is done differently.** The standard class verification process is too computationally expensive, so an alternate process was defined. The alternate process moves most of the verification work to a separate *preverification* step that occurs on a desktop or server computer and not on the device. The preverified class files are then processed on the device using a second, much simpler kind of verification that merely validates the results of the preverification step.

The third requirement is that any classes that are drawn or "inherited" from J2SE must be subsets of the J2SE 1.3 classes. Methods can be omitted, but no new public methods or data members can be added. Upward compatibility is of paramount importance.

The fourth requirement is that classes defined by the CLDC and its profiles are in the `javax.microedition` package or its subpackages, which makes it easy to identify the classes that are specific to the CLDC.

The final requirement is for minimal internationalization support. The CLDC provides basic support for converting byte streams to Unicode and back by using at least one character encoding. The CLDC does not address localization issues, such as how to display dates, times, currencies, and other locale-specific behaviors.

70899_Ortiz_CH01I 10/30/2001 3:30 PM Page 8

Supported J2SE Classes

The CLDC includes classes and interfaces drawn from these three J2SE packages:

- java.lang
- java.io
- java.util

As you can see, most J2SE classes are excluded, including those from useful packages such as java.awt, java.net, and java.sql. Even the packages that are included are missing classes, and many of those classes are missing methods. A listing of supported non-exception classes is found in Table 1.1. For a complete list of what is actually included in each class and which exceptions are available, refer to the class reference in Appendix A.

Table 1.1 Non-Exception J2SE 1.3 Classes Included in the CLDC

PACKAGE	CLASS
java.lang	Boolean
	Byte
	Character
	Class
	Integer
	Long
	Math
	Object
	Runnable
	Runtime
	String
	StringBuffer
	System
	Thread
	Throwable
java.io	ByteArrayInputStream
	ByteArrayOutputStream
	DataInput
	DataInputStream
	DataOutput
	DataOutputStream
	InputStream

70899_Ortiz_CH01I 10/30/2001 3:30 PM Page 9

Table 1.1 Continued

PACKAGE	CLASS
	InputStreamReader
	OutputStream
	OutputStreamWriter
	PrintStream
	Reader
	Writer
java.util	Calendar
	Date
	Enumeration
	Hashtable
	Random
	Stack
	Time
	Vector

The Generic Connection Framework

Apart from the classes discussed, the only other classes defined by the CLDC are the classes that make up the Generic Connection Framework, or GCF for short. The GCF abstracts the concepts of files, sockets, HTTP requests and other input/output mechanisms into a simpler set of classes than those defined by J2SE. In other words, the GCF is meant to provide the same functionality as classes from the `java.io` and `java.net` packages without requiring specific capabilities from a device. Note that the CLDC does, in fact, include some classes from the `java.io` package, but only the classes that do not depend on the capabilities of the underlying operating system. The GCF does not replace these basic input/output classes and depends on and uses classes such as `java.io.InputStream` and `java.io.OutputStream`. One way to look at the GCF is as a framework for building communications drivers, much like JDBC in J2SE is a framework for building database drivers.

With the GCF, all communication is abstracted through a set of well-defined interfaces. Instead of creating a specific class of communication objects like `java.io.File` or `java.net.Socket`, the application asks the GCF to create a connection that uses a specific protocol. The protocol can be a formal protocol, such as *Hypertext Transfer Protocol* (HTTP), or a reference to a low-level storage or communication facility like a filesystem or a wireless packet transceiver. The protocol is passed in as part of a *Universal Resource Identifier* (URI) that specifies other important information that is relevant to the protocol, such as the name of a host to connect to or the name of a file on the

70899_Ortiz_CH01I 10/30/2001 3:30 PM Page 10

filesystem. The GCF then determines whether the implementation supports that protocol and returns an appropriate interface if it does. The application then uses this interface to interact with the implementation in sending or receiving data.

The classes defined by the GCF are listed in Table 1.2. All the classes are defined as part of the `javax.microedition.io` package. An application uses one of the static `Connector.open` methods to obtain an object that implements the `Connection` interface or one of its subinterfaces. The application then uses the methods defined by the interface to read and/or write data. Most of the interfaces work on stream-based data and therefore expose input or output streams.

All told, there are six subinterfaces of `Connection` defined. The `Input-Conection` and `OutputConnection` interfaces are for one-way stream connections. The `StreamConnection` interface is for two-way stream connections—it extends both `InputConnection` and `OutputConnection`. The `Content-Connection` interface extends `StreamConnection` with methods for determining information about the content itself such as its type and length. The `DatagramConnection` interface is for sending and receiving packet data. Finally, the `StreamConnectionNotifier` interface is for implementing server-side connections where the application must wait for a client to connect to it. The interface hierarchy is shown in Figure 1.1.

Note that while the CLDC defines the Generic Connection Framework, *it does not mandate support for any particular protocol*. This concept has confused more than one novice J2ME programmer, because there is a reference implementation of the CLDC available from Sun Microsystems that includes support for a number of communication protocols. Those protocols are there strictly as examples, though. Protocol support is defined at the profile level or as device-specific extensions.

We will look at the GCF in more detail in Chapter 6, "Network Communication," when we discuss networking and the MIDP.

Table 1.2 Generic Connection Framework

PACKAGE	CLASSES/INTERFACES
javax.microedition.io	Connection
	ConnectionNotFoundException
	Connector
	ContentConnection
	Datagram
	DatagramConnection
	InputConnection
	OutputConnection
	StreamConnection
	StreamConnectionNotifier

70899_Ortiz_CH01I 10/30/2001 3:30 PM Page 11

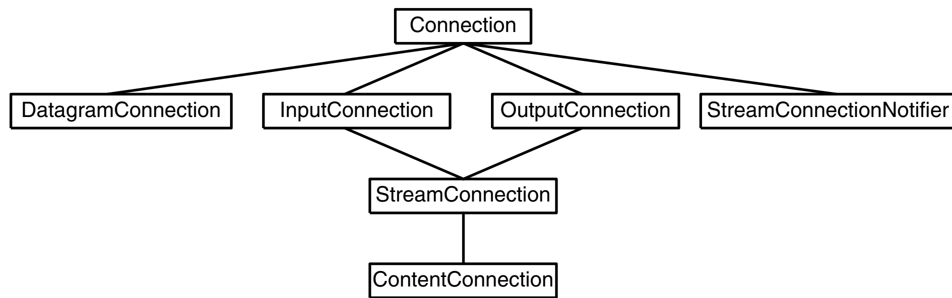

Figure 1.1 The GCF connection hierarchy.

Summary

In this chapter, we have taken a short history lesson on Java and learned about Java 2 Micro Edition in general and the Connected Limited Device Configuration more specifically. We are now ready to take our first look at the Mobile Information Device Profile.