Chapter 1

Introduction to the Java 2 Micro Edition (J2ME) Platform

- Defining a Java Platform for Pervasive Devices
- Configurations and Profiles
- Device Application Management Systems

Sun Microsystems has defined three Java platforms, each of which addresses the needs of different computing environments:

- Java 2 Standard Edition (J2SE)
- Java 2 Enterprise Edition (J2EE)
- Java 2 Micro Edition (J2ME)

The inception of the J2ME platform arose from the need to define a computing platform that could accommodate consumer electronics and embedded devices. These devices are sometimes referred to collectively as *pervasive* devices.

The creators of the J2ME platform delineated pervasive devices into two distinct categories:

- **Personal, mobile information devices** that are capable of intermittent networked communications—mobile phones, two-way pagers, personal digital assistants (PDAs), and organizers
- **Shared-connection information devices** connected by fixed, uninterrupted network connection—set-top boxes, Internet TVs, Internet-enabled screen phones, high-end communicators, and car entertainment/navigation systems

The first category describes devices that have a special purpose or are limited in function; they are not general-purpose computing machines. The second category describes devices that generally have greater capability for user interface (UI) facilities. Of course, devices with superior UI facilities typically have more computing power. Practically speaking, computing power is the primary attribute that distinguishes these two categories of devices. Nevertheless, this delineation is somewhat fuzzy, because technology continues to enable more and more power to be placed in smaller and smaller devices.

Like computing power, connectivity—the availability of media such as wireless networks—also affects the kinds of functionality and services that pervasive devices can support. The challenge—and the primary goal—for J2ME is to specify a platform that can support a reasonable set of services for a broad spectrum of devices that have a wide range of different capabilities.

The creators of J2ME identify modular design as the key mechanism that enables support for multiple types of devices. The J2ME designers use configurations and profiles to make J2ME modular.

Defining a Java Platform for Pervasive Devices

Configurations and profiles are the main elements that comprise J2ME's modular design. These two elements enable support for the plethora of devices that J2ME supports.

A J2ME *configuration* defines a minimum Java platform for a family of devices. Members of a given family all have similar requirements for memory and processing power. A configuration is really a specification that identifies the system-level facilities available, such as a set of Java language features, the characteristics and features of the virtual machine present, and the minimum Java libraries that are supported. Software developers can expect a certain level of system support to be available for a family of devices that uses a particular configuration.

A configuration also specifies a minimum set of features for a category of devices. Device manufacturers implement profiles to provide a real platform for a family of devices that have the capabilities that a given configuration specifies.

The other J2ME building block, the *profile*, specifies the application-level interface for a particular class of devices. A profile implementation consists of a set of Java class libraries that provide this application-level interface. Thus, a profile theoretically could specify all kinds of functionality and services.

This is not the intention of its creators, however. The creators of J2ME intend that a profile should address the needs of a specific device category or vertical market pertaining to that device category. The idea is not to place a plethora of unrelated application level features in a profile. Rather, the main goal is to guarantee interoperability—which doesn't necessarily imply compatibility between different manufacturers' implementations—between all devices of the same category or vertical market family to define a standard platform for Java application development.

For example, a profile might support a network communication facility for the popular Short Message Service (SMS) standard widely used by mobile phones. Because the SMS standard is a ubiquitous feature of mobile telephony, it makes sense to define this service in a profile that targets mobile phones, rather than to build it into a configuration.

A profile is implemented on top of a configuration, one step closer to the implementation of real-world applications. Typically, a profile includes libraries that are more specific to the characteristics of the category of devices they represent than are the libraries that comprise configurations. Applications are then built on top of the configuration and profile; they can use only the class libraries provided by these two lower-level specifications. Profiles can be built on top of one another. A J2ME platform implementation, however, can contain only one configuration. Figure 1.1 shows the conceptual layers that comprise the J2ME platform.

So far, these notions of configurations, profiles, and platform definitions is somewhat abstract. The next section gives you a more concrete description of the characteristics of actual environments.

Configurations and Profiles

A configuration specifies three basic elements:

- a set of Java programming language features
- a set of Java virtual machine features
- a set of supported Java libraries and application programming interfaces (APIs)

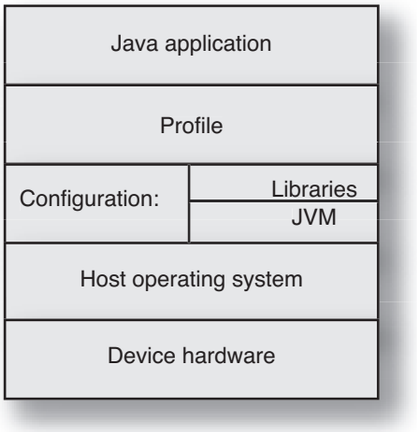

Figure 1.1 The J2ME platform consists of a set of layers that support a basic runtime environment with core Java libraries and a Virtual Machine (VM), a set of system-level application programming interfaces (APIs) in a configuration, and a set of application-level APIs in a profile.

The creators of J2ME have defined only two configurations to avoid a fragmented landscape of incompatible platforms. The two configurations that exist currently represent the two categories of pervasive devices you saw earlier in this chapter, namely:

- **personal, intermittently connected mobile devices—**supported by the Connected, Limited Device Configuration (CLDC)
- **constantly connected network devices—**supported by the Connected Device Configuration (CDC)

Theoretically, a configuration could specify the very same support as the J2SE platform libraries. This is unlikely in the real world because, as you now know, J2ME is targeted at devices that are far less powerful than desktop computers.

Configuration specifications require that all Java classes adapted from J2SE be the same as or a proper subset of the original J2SE class. That is, a class cannot add methods not found in the J2SE version. Configurations can include additional classes in their specifications, however; configurations themselves are not necessarily proper subsets of J2SE. Both configurations that have been defined to date add classes not present in J2SE in order to address device attributes and constraints.

The Connected Device Configuration (CDC)

The Connected Device Configuration (CDC) intends to capture just the essential capabilities of each kind of device in the category of devices it targets, namely, devices with 2 MB or more of total memory, including both RAM and ROM.

As you saw in Figure 1.1, a configuration specifies both the set of Java VM features that are supported and a set of class libraries. The CDC specifies the use of the full Java 2 platform VM, which, in this context, is called the Compact Virtual Machine (CVM).

The CVM. Although the CVM supports the same features as the J2SE VM, it is designed for consumer and embedded devices. This means that the standard J2SE VM has been reengineered to suit the constraints of limited-resource devices. The features of the resulting offspring CVM are:

- advanced memory system
- small average garbage collection pause times
- full separation of VM from memory system
- modularized garbage collectors
- generational garbage collection

In particular, the CVM has been engineered to offer the following features:

- portability
- fast synchronization
- execution of Java classes out of read-only memory (ROM)
- native thread support
- small class footprint
- provision of interfaces to and support for real-time operating system (RTOS) services
- mapping Java threads directly to native threads
- support for all Java 2, v1.3 VM features and libraries: security, weak references, Java Native Interface (JNI), Remote Method Invocation (RMI), Java Virtual Machine Debugging Interface (JVMDI)

CDC Class Libraries. The CDC specifies a minimal set of class libraries and APIs. It supports the following standard Java packages:

- `java.lang`—Java VM system classes
- `java.util`—underlying Java utilities

- `java.net`—Universal Datagram Protocol (UDP) datagram and input/output (I/O)
- `java.io`—Java file I/O
- `java.text`—very minimal support for internationalization (I18N—see chapter 9)
- `java.security`—minimal fine-grain security and encryption for object serialization

As you can see, these APIs do not include the full set of Java 2 software development kit (SDK) packages. In some cases, these packages and classes are subsets of the Java 2 SDK packages and classes. Resource constraints dictate removal of the remainder of the J2SE classes and APIs. Also, all deprecated J2SE APIs are removed. Table 1.1 lists the full set of packages supported by the CDC.

Table 1.1 CDC Packages

CDC Package Name	Description
`java.io`	Standard IO classes and interfaces
`java.lang`	VM classes
`java.lang.ref`	Reference classes
`java.lang.reflect`	Reflection classes and interfaces
`java.math`	Math package
`java.net`	Networking classes and interfaces
`java.security`	Security classes and interfaces
`java.security.cert`	Security certificate classes
`java.text`	Text package
`java.util`	Standard utility classes
`java.util.jar`	Java Archive (JAR) utility classes
`java.util.zip`	ZIP utility classes
`javax.microedition.io`	CDC generic connection framework classes and interfaces

The Foundation Profile. A configuration, together with a profile, creates a J2ME runtime environment. The system-level features and services supported by a configuration are more or less hidden from the application developer. In reality, the application developer is prohibited from accessing them directly. If this were not the case, the application would not be considered J2ME compliant.

From the programmer's perspective, a profile is required to do "useful" work. A profile defines the layer that contains the APIs that the programmer usually

manipulates. The J2ME creators initially defined one CDC profile, the *Foundation Profile,* which is based on the J2SE v1.3 release. It was designed by standard committee through the Java Community Process, by an expert group of companies in the consumer electronics industry. The Foundation Profile contains the J2SE packages listed in Table 1.2.

The list of packages above looks exactly like the list that comprises the CDC. In fact, they are the same. To say that the Foundation Profile contains these packages really means that they are available to the Foundation Profile. The intention is that the Foundation Profile be used with the CDC. The delineation between the profile and the configuration is a conceptual one, not a physical one.

Notice that the whole `java.awt` Abstract Window Toolkit (AWT) and `javax.swing` Swing package hierarchies that define the J2SE graphical user interface (GUI) APIs are absent from the supported packages. If an application needs a GUI, an additional profile would be required. Profiles can be built on top of one another. An implementation of the J2ME platform, however, can contain only one configuration.

The lack of GUI support in the Foundation Profile has less impact for the family of shared, constantly connected network devices such as TV set-top boxes than it does for personal, mobile devices, which are served by the second J2ME configuration, the CLDC.

In general, the decision to include or omit features and libraries from a configuration or profile is based on their footprints, static and dynamic resource requirements, and security requirements.

Table 1.2 **Foundation Profile Packages**

Foundation Profile Package Name	Description
`java.lang`	Rounds out full `java.lang.*` J2SE package support for the Java language (`Compiler, UnknownError`)
`java.util`	Adds full zip support and other J2SE utilities (`java.util.Timer`)
`java.net`	Adds TCP/IP Socket and HTTP connections
`java.io`	Rounds out full `java.io.*` J2SE package support for Java language input/output (`Reader` and `Writer` classes)
`java.text`	Rounds out full `java.text.*` J2SE package support for internationalization (I18N): `Annotation, Collator, Iterator`
`java.security`	Adds code signing and certificates

Personal Profile. The Personal Profile specification was created through the Java Community Process, resulting in JSR-62. The Personal Profile provides an environment with full AWT support. The intention of its creators is to provide a platform suitable for Web applets. It also provides a J2ME migration path for Personal Java applications.

Personal Profile version 1.0 requires an implementation of the Foundation Profile version 1.0. It is a superset of the Personal Basis Profile version 1.0. Personal Profile is a subset of the J2SE version 1.3.1 platform, however, which makes Personal Profile applications upward compatible with J2SE version 1.3.1.

Table 1.3 lists the packages that comprise Personal Profile version 1.0.

Table 1.3 **Personal Profile Packages**

Personal Profile Package Name	Description
`java.applet`	Classes needed to create applets and those used by applets
`java.awt`	Classes for creating AWT UI programs
`java.awt.datatransfer`	Classes and interfaces for transferring data within and between applications
`java.awt.event`	Classes and interfaces for AWT event handling
`java.awt.font`	Classes and interface for font manipulation
`java.awt.im`	Classes and interfaces for defining input method editors
`java.awt.im.spi`	Interfaces that aid in the development of input method editors for any Java runtime environment
`java.awt.image`	Classes for creating and modifying images
`java.beans`	Classes that support JavaBean development
`javax.microedition.xlet`	Interfaces used by J2ME Personal Profile applications and application managers for communication

RMI Profile. The RMI Profile is a profile designed for platforms that support the CDC configuration. It has been defined by JSR-66 by various companies participating through the Java Community Process.

The RMI Profile requires an implementation of the Foundation Profile and is built on top of it. RMI Profile implementations must support the following features:

- full RMI call semantics
- marshaled object support

- RMI wire protocol
- export of remote objects through the `UnicastRemoteObject` API
- distributed garbage collection and garbage collector interfaces for both client and server side
- the activator interface and the client side activation protocol
- RMI registry interfaces and export of a registry remote object

The RMI profile supports a subset of the J2SE v1.3 RMI API. The following interfaces and features are part of the J2SE v1.3 RMI specification and public API, but support for these interfaces and functionality is omitted from the RMI profile specification because of limitations on device processing power, network performance, and throughput:

- RMI through firewalls and proxies
- RMI multiplexing protocol
- implementation model for an "activatable" remote object
- deprecated methods, classes, and interfaces
- support for the RMI v1.1 skeleton/stub protocol
- stub and skeleton compiler

Support for the following J2SE RMI v1.3 properties is omitted:

- `java.rmi.server.disableHttp`
- `java.rmi.activation.port`
- `java.rmi.loader.packagePrefix`
- `java.rmi.registry.packagePrefix`
- `java.rmi.server.packagePrefix`

Connected, Limited Device Configuration (CLDC)

The second of the two J2ME configurations, the Connected, Limited Device Configuration (CLDC), supports personal, mobile devices, which constitute a significantly less powerful class of devices than the one that the CDC supports. The CLDC specification identifies devices in this category as having the following characteristics:

- 160 to 512 KB total memory available for the Java platform
- 16-bit or 32-bit processor
- low power consumption, often battery powered

- intermittent network connectivity (often wireless) with potentially limited bandwidth

The goal of the CLDC is to define a standard Java platform for these devices. Because of the wide variety of system software on various personal devices, the CLDC makes minimum assumptions about the environment in which it exists. For example, one OS might support multiple concurrent processes, another might or might not support a file system, and so forth.

The CLDC is different from, yet also a subset of the CDC. The two configurations are independent of each other, however, so they should not be used together to define a platform. Figure 1.2 shows the relationship between the two configurations and the J2SE platform.

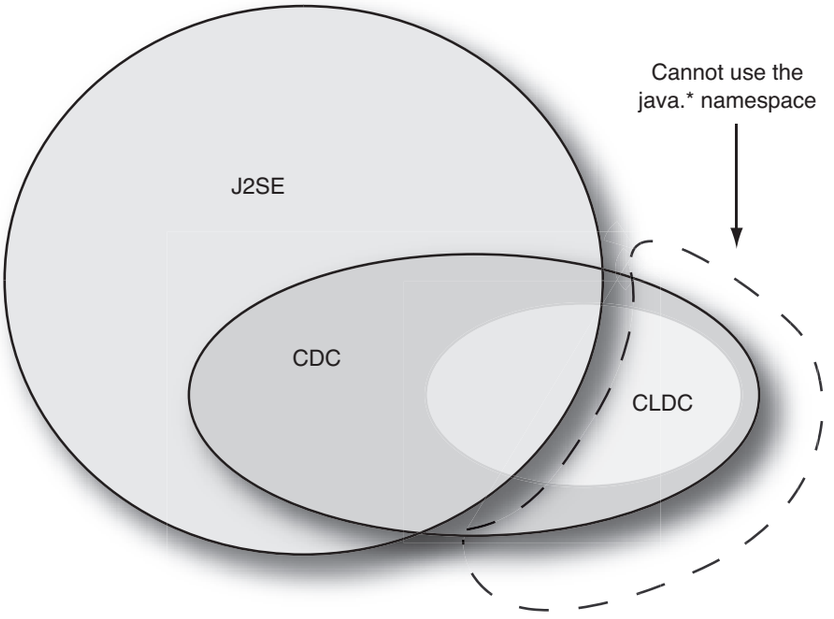

Figure 1.2 The CLDC is a proper subset of the CDC. Neither the CLDC nor the CDC is a proper subset of the J2SE platform, however, because both of these configurations add new classes necessary to deliver services on their respective families of devices.

Like the CDC, the CLDC specifies the level of support of the Java programming language required, the required functional support of a compliant Java VM, and the set of class libraries required.

Java Language Support. The CLDC specification omits support for the following features of the Java language:

- floating point calculations
- object finalization
- the `java.lang.Error` class hierarchy in its entirety

Of course, these features involve the VM as well and are discussed in chapter 5 of the CLDC specification ("Adherence to Java Virtual Machine Specification"). I address them here, however, because they have a language-level presence that affects programmers.

The lack of floating point support is the main language-level difference between a Java virtual machine that supports CLDC and a standard J2SE VM that is visible to programmers. This means that programs intended to run on the CLDC cannot use floating point literals, types, or values. You can't use the `float` built-in type, and the `java.lang.Float` class has been removed from CLDC libraries. This feature is not present because of the lack of floating-point hardware or software on most mobile devices.

Object finalization is also absent. This means that the `Object.finalize()` method has been removed from the CLDC libraries.

The `java.lang.Error` exception hierarchy has also been removed from the CLDC libraries and is therefore not available to applications. The primary reason that error handling is absent is memory constraints on mobile devices. This typically doesn't create any disadvantages for applications development; after all, applications are not supposed to recover from error conditions. And the resource cost of implementing error handling is expensive, beyond the capabilities of today's mobile devices. Moreover, error recovery is device-specific on embedded devices like mobile phones. In consequence, it doesn't make sense to stipulate the recovery mechanism that devices should use. This mechanism may well be outside the scope of an embedded VM.

Java Virtual Machine and Library Support. The CLDC specifies requirements for a Java virtual machine. It defines a VM that is highly portable and designed for resource-constrained small devices. Support for several features that exist in a standard J2SE VM have been omitted from the CLDC specification. The following list describes the features that are not supported in a CLDC-compliant

VM. The features in this list have been omitted because of either changes to libraries or security concerns:

- Java Native Interface (JNI)
- user-defined class loaders
- reflection
- thread groups and thread daemons
- finalization (no `Object.finalize()` method in CLDC libraries)
- weak references
- errors (a small subset of J2SE errors is supported)
- class file verification

Among these unsupported features, class file verification deserves further mention. The VM in the CLDC specification still performs this process, but it uses a two-step process and a different algorithm that requires fewer computation resources than the standard J2SE verifier. In addition, there is a new preverification tool, which you will learn about in chapter 2.

The VM that comes with the CLDC reference implementation is called the Kilobyte Virtual Machine (KVM), so named because it uses only a few KB of runtime memory. It is a reference implementation that adheres to the CLDC specification's description of a compliant VM. The KVM is not a full-featured J2SE VM.

The specification of the features that a VM supports includes a specification of the libraries that it supports. The CLDC specification details the libraries that an implementation must support.

As you know, a configuration is the basis for one or more profiles. The CLDC is a configuration on top of which one or more profiles are to be built in the same way that the Foundation Profile is built on top of the CDC. The intention is that the APIs in the CLDC profile support application development for the mass market of personal devices. The CLDC therefore targets third-party application developers. This is somewhat different than the CDC, which targets OEM developers.

Table 1.4 lists the packages that comprise the CLDC. Notice that it is quite a bit smaller than the list of packages contained in the CDC, shown earlier in Table 1.1.

The first three packages use the `java.` prefix in their name because each one contains a subset of the standard J2SE platform classes. The last one, however, must use the `javax.` prefix because it defines a new "standard extension" that is not part of the core Java platform.

Table 1.4 CLDC Packages

CLDC Package Name	Description
`java.io`	Standard Java IO classes and packages; subset of the J2SE package
`java.lang`	VM classes and interfaces; subset of the J2SE package
`java.util`	Standard utility classes and interfaces; subset of the J2SE package
`javax.microedition.io`	CLDC generic connection framework classes and interfaces

Mobile Information Device Profile. Because the category served by the CLDC encompasses so many different types of personal devices, potentially many different profiles are necessary to support them all. The most popular and well known of these is the Mobile Information Device Profile (MIDP), sometimes called the MID Profile. The MIDP layers atop the CLDC and defines a set of user interface (UI) APIs designed for contemporary wireless devices.

Following in the tradition of Java parlance, MIDP applications are called MIDlets. A MIDlet is a Java application that uses the MIDP profile and the CLDC configuration. This book concentrates on teaching you how to write MIDlets, because the vast majority of J2ME programmers will encounter the CLDC/MIDP platform far more often than other J2ME platforms. And, from a practical standpoint, the MIDP is the only profile currently available.

Another profile, the PDA Profile, is currently in its definition stage. PDAs also belong to the general category of mobile information devices. The PDA profile might never be implemented, however, because it's questionable whether it offers enough differences and enhancements to the MIDP specification to warrant its development. The PDA Profile also poses portability challenges for developers.

The MIDP specification, like the CDC's Foundation Profile, was produced by an expert group, in this case, the Mobile Information Device Profile Expert Group, which is an international forum that includes representatives from several companies in the mobile device arena. The MIDP targets mobile information devices (MIDs), such as mobile phones, two-way pagers, and so forth, which have roughly the following characteristics:

- screen size of approximately (at least) 96x54 pixels
- display depth of 1 bit

- one- or two-handed keyboard, touchscreen input device
- 128 KB nonvolatile memory for MIDP components
- 8 KB nonvolatile memory for application-persistent data
- 32 KB volatile runtime memory for Java heap
- two-way wireless connectivity

Because the range of MID capabilities is so broad, the MIDP established a goal to address the least common denominator of device capabilities. The MIDP, therefore, specifies the following APIs:

- application (MIDP application semantics and control)
- user interface
- persistent storage
- networking
- timers

Table 1.5 lists the packages that comprise the MIDP.

Table 1.5 MIDP Packages

MIDP Package Name	Description
`javax.microedition.lcdui`	UI classes and interfaces
`javax.microedition.rms`	Record management system (RMS) supporting persistent device storage
`javax.microedition.midlet`	MIDP application definition support class types
`javax.microedition.io`	MIDP generic connection framework classes and interfaces
`java.io`	Standard Java IO classes and interfaces
`java.lang`	VM classes and interfaces
`java.util`	Standard utility classes and interfaces

You'll learn more about the programming details of the APIs in Table 1.5 in chapters 3 through 9.

A MIDP implementation must consist of the packages and classes specified in the MIDP specification. Additionally, it can have implementation-dependent classes for accessing native system software and hardware.

Figure 1.3 juxtaposes the CDC and CLDC platform stacks. There is nothing inherent in either the CDC or CLDC that prohibits a manufacturer from porting either platform to a given family of devices. Nevertheless, the platform stacks—specifically, the

configuration and profile features—have been specified to address practical limitations of the different families of hardware devices.

Device Application Management Systems

All J2ME applications—MIDlets and others—are real Java applications that run under the control of a Java VM. But what controls the Java VM, for instance on a mobile phone? There's no command shell from which you can invoke your favorite Java applications like you do on your workstation. Starting, stopping, and managing the execution of J2ME applications is controlled by *application management software* (AMS) that resides on the device. In fact, the AMS controls the entire application lifecycle, from installation, upgrade and version management, to removal of application software.

The device manufacturer typically provides the AMS software. This is the most logical scenario because AMS software must work in conjunction with the device's native system software, which, presumably, the manufacturer knows best. Nevertheless, third parties can also develop AMS systems for specific devices. AMS software could be written, for example, in Java or in some native language such as C.

Understanding the issues surrounding application management is important for the J2ME developer. Chapter 10 discusses application management. You must be

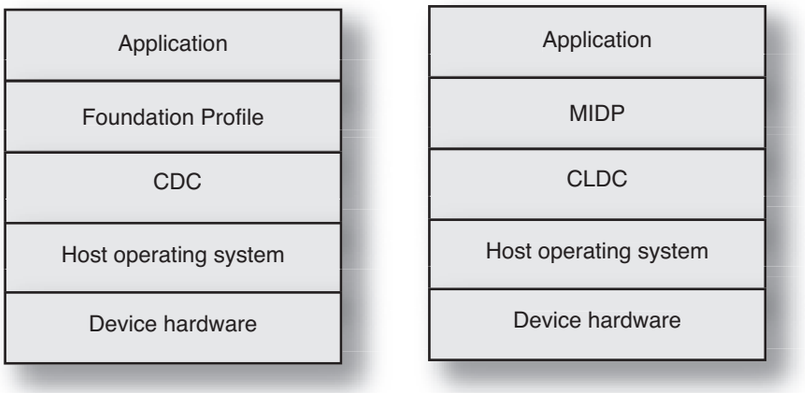

Figure 1.3 The CDC targets fixed-connection, shared, stationary devices. The CLDC targets personal, mobile, limited-connection devices.

aware of the ramifications of your choices regarding packaging, licensing, charging for use, and so forth, and how these decisions will affect the usability and viability of your software.

Chapter Summary

The J2ME platform addresses two classes of pervasive computing devices. The first class consists of stationary devices with fixed network connections such as TV set-top boxes. The second consists of personal, mobile devices with intermittent network connectivity, such as PDAs, mobile phones, and so on.

Different combinations of J2ME configurations and profiles support these two classes of devices. The CDC configuration and Foundation Profile support the former class of devices, and the CLDC configuration and MIDP profile support the latter.

A configuration attempts to provide interfaces for system-level services. A profile attempts to provide standard interfaces for application-level services. The configuration enables the profile, providing the necessary medium and mechanisms.

Devices must have some AMS to "bootstrap" the process of provisioning J2ME applications on devices. The device manufacturer usually provides the AMS.