

1

Introduction to J2ME

In order to understand how Java 2 Micro Edition (J2ME) lies within the wider Java landscape it is best to explore the overall Java architecture. J2ME has been developed primarily as a technology for the execution of applications on constrained devices. In this case, constrained devices are mobile phones, PDAs, TV set-top boxes, in-vehicle telemetry, residential gateways and other embedded devices.

J2ME as a whole can be described as the technology that caters for all these devices. Given that many of them have limited resources, it would be imprudent to expect all of these devices to be able to deliver all of the functionality of the few. The Java community therefore decided that these devices should be grouped to best reflect their purpose and capabilities. This would provide a lowest common denominator for each device group and arrange them into **configurations**. To further differentiate these devices and to accommodate vertical markets within each configuration, **profiles** were created, refining the Java APIs for each device type.

The following analyzes how J2ME is positioned within the Java architecture and how the J2ME configurations and profiles complement each other. It also describes the packages and classes within the commonly used environments, with special emphasis on MIDP 2.0.

1.1 Configurations and Profiles

1.1.1 Architecture

J2ME is the newest and smallest addition to the Java family. It is the smaller brother of J2SE (Standard Edition) and the server-based J2EE (Enterprise Edition). As mentioned, J2ME provides a development environment for a range of small, constrained devices. Even though J2ME is targeted at devices with limited capabilities, it has been derived from J2SE and shows all the characteristics of the Java language. We have already

introduced the concepts of configurations and profiles; the rest of this chapter will explain how and why these concepts have been derived and implemented.

Each combination of configuration and profile matches a group of products specifically optimized to match the memory, processing power and I/O capabilities of each device.

The full Java architecture can be seen in Figure 1.1. It shows how the technology has developed to offer a platform for a range of circumstances. Enterprise applications can be developed using the J2EE packages, taking full advantage of the power of large servers capable of transmitting large chunks of data across networks. The J2SE edition complements J2EE and provides the basis for desktop-type applications. Already we can see that these two versions of Java are defined with consideration of processor power, memory and communication ability: it would be inefficient for the virtual machine running on a desktop machine (J2SE) to also include large packages targeted towards an enterprise application (J2EE).

Further inspection of the Java architecture reveals that there are two groups of special interest to us, under the banner of J2ME. J2ME provides an environment for developers wishing to develop applications for smaller devices. This environment has been specialized to cater for machines with even less capacity.

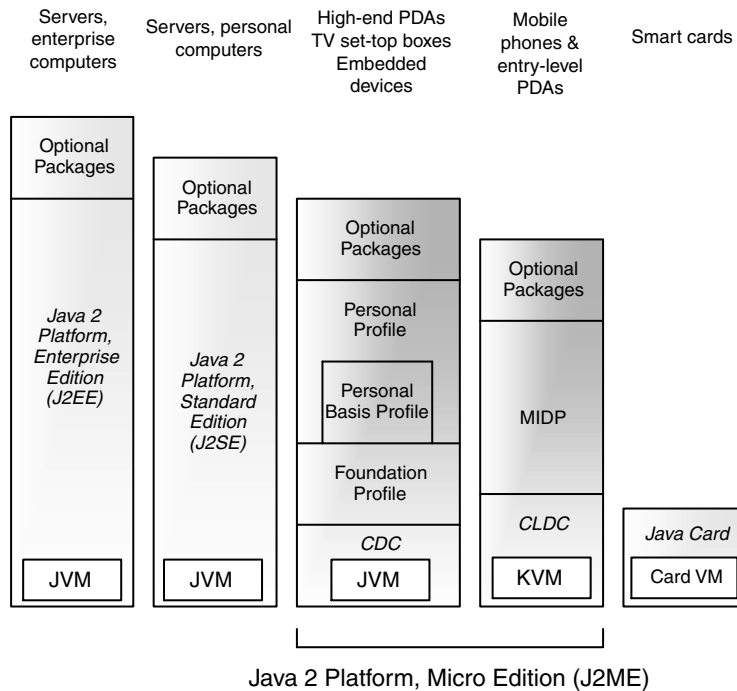


Figure 1.1 The Java landscape.

1.1.2 Configurations

So far we have examined the bigger Java picture and looked at how J2ME fits within that. We have also established that J2ME provides an environment for the development and execution of applications for constrained devices. These devices cover a broad range of functionality and use: we may want to program devices that provide telemetry data from a vehicle, or create data applications for a TV set-top box; but we might instead want to develop applications for mobile phones.

These three examples show immediately why we might want to split J2ME into configurations. While an application sitting in a motor vehicle transmitting data back to a server has much in common with a gaming application transmitting high scores to a server, one thing that becomes apparent is the differential in power source available to both. One device is able to draw on the car battery, whereas a mobile phone has to rely on a rechargeable battery. The requirements in the cost and size of the hardware are also different. This provides particular constraints on the capabilities of the processor and therefore the virtual machine within the device. While all these devices have common attributes, not all of them are the same. It is therefore necessary to provide a set of base classes appropriate to each grouping of devices.

A configuration consists of a combination of a virtual machine and a minimal set of class libraries designed to provide the base functionality for a distinct set of devices with similar characteristics, such as network connectivity, processor power and memory. There are two such current configurations, defined as follows:

- **Connected Device Configuration (CDC)**
This configuration is designed for devices with more memory, faster processors and greater network bandwidth. It is appropriate, at least in the near term, for home automation and automotive entertainment, navigation, and telemetry systems. A programming model closer to J2SE simplifies porting existing desktop clients for enterprise systems to mobile devices that support CDC.
- **Connected Limited Device Configuration (CLDC)**
This configuration is intended for devices with intermittent network connections, small processors and limited memory. Expected targets included two-way pagers, mobile phones and entry-level PDAs. However, in practice, the functionality delivered by CLDC and the associated profiles and optional packages is very close to that of CDC. As a consequence it is used today on most high-end mobile phones, or smartphones, which are replacing PDAs in the marketplace.

1.1.3 Profiles

Whereas a configuration provides the lowest common denominator for a group of devices, the profile adds an additional layer on top of the

configuration providing APIs for a specific class of device. This creates the ability for each configuration to be adapted and targeted towards vertical markets. That is to say, while some devices may appear to have similar functionality, they do in fact have different requirements in terms of the available APIs and interfaces to their own hardware. Some mobile phones, for example, offer more memory, CPU speed or I/O interfaces than others and therefore might want to offer more in terms of an interface between the programmer and the hardware.

Currently, four Java Community Process profiles exist across the two J2ME configurations, but only one of those is a CLDC profile. However, an additional profile called 'DoJa', defined by NTT DoCoMo, operates on the J2ME CLDC APIs and is used on i-mode devices. With only one JCP profile currently defined, a developer new to J2ME might ask themselves: why is a profile required at all?

Using the example of two-way pagers as a possible type of CLDC device, it becomes easier to understand the need for another profile. We can see there are similarities between two-way pagers and mobile phones. Both usually connect intermittently over a wireless network, both can communicate via text type messaging and, possibly, both may store a certain level of information, such as phone numbers. They will both also have a screen of some description. However, the user interface (UI) signals the beginning of the diversity between the two types of device. The method by which data input is captured and indeed displayed will be very different. Each device should have a UI in tune with its own capabilities. While both types of device are CLDC, each will require a separate profile so that the most appropriate APIs are available to the developer.

Mobile Information Device Profile (MIDP)

This profile offers the core functionality required by mobile applications, such as the user interface, network connectivity, local data storage and, importantly, application lifecycle management. As well as the reference implementation for mobile phones and pagers, there is a second implementation that caters for the Palm OS. It is known as MIDP for Palm OS and it provides for the different user interface on such devices.

Information Module Profile (IMP)

This profile is based upon the MIDP 1.0 profile. IMP combined with CLDC provides a Java application environment targeted at resource-constrained and embedded networked devices. These devices do not have rich graphical user interfaces, but their relationship to MIDP 1.0 means that developer skills can be easily transferred to IMP.

Foundation Profile

This profile is the first of three, tiered CDC profiles. It provides a network-capable implementation without a user interface. It can be combined

with the Personal Profile and Personal Basis Profile when devices require a UI.

Personal Profile

This profile is aimed at devices that require full GUI or Internet applet support, such as high-end PDAs or communicator-type devices. It provides a full Abstract Window Toolkit (AWT) library and offers web fidelity. It is capable of running web-based applets designed for the desktop environment.

Personal Basis Profile

This profile is a subset of the Personal Profile and provides a network-based environment for network-connected devices that support a limited GUI or require specialized graphical interfaces. Devices include set-top boxes and in-vehicle systems.

The Personal Basis Profile and Personal Profile have replaced PersonalJava technology and provide a clear migration path for PersonalJava applications to J2ME. Although Personal Information Management and Telephony APIs are not mandatory in this profile, replacements are being specified for J2ME use. Both the Personal Basis Profile and Personal Profile are layered on top of the CDC and Foundation Profile.

1.2 CLDC and MIDP

1.2.1 CLDC

A developer wishing to create applications for mobile devices may be tempted to ignore the full specification of CLDC. A developer may initially be interested in getting acquainted with MIDP as a standalone technology. It is, however, important to understand the underlying technology that forms MIDP.

The CLDC, as specified by Java Specification Request (JSR) 30 (<http://jcp.org/en/jsr/detail?id=30>), is the smaller of the two configurations and sets out to define a standard for devices with the following capabilities:

- 160 KB to 512 KB of total memory budget available for the Java platform
- 16-bit or 32-bit processor
- low power consumption, often operating on battery power
- intermittent network connection, possibly wireless and limited to a bandwidth of 9600 bps or less.

The 160 KB memory budget is derived from the minimum hardware requirements, as follows:

- at least 128 KB of non-volatile memory available for the Java Virtual Machine and CLDC libraries
- 32 KB of volatile memory for the Java runtime object memory.

CLDC itself defines the minimum required Java technology in terms of libraries and components for small-connected devices. Specifically, this addresses the Java language itself, the virtual machine definition, core libraries, I/O capabilities, networking and security.

Interestingly, from an early stage, one of the focuses for the CLDC definition was to recognize that much of the content for these devices would come from third-party developers. Another was that the idea of being able to create applications portable across a range of devices should be adhered to. This would provide an easier path to revenue generation and therefore proliferate content for more devices. The nature of Java means that a programmer can create applications that use the device's features without having to actually understand the working of the device. The developer only needs to comprehend the interface to the device. CLDC does not guarantee portability and it does not implement any optional features. Variants of devices within CLDC should be specified through profiles, rather than the configuration. It must be said that true application portability can only be obtained if a few principles are applied during the application design stage. We shall be looking at these issues later in this book.

1.2.1.1 K-Virtual Machine

Sun's original VM for CLDC was known as the KVM (which stood for Kauai Virtual Machine, sometimes also known as the Kilo Virtual Machine). The CLDC VM is, apart from a few differences which we shall outline shortly, compliant with the Java Virtual Machine Specification and the Java Language Specification.

The libraries available are typically split into two categories: those defined by CLDC and those defined by a profile and its optional packages such as MMAPAPI and WMA. Figure 1.2 demonstrates at a high level how these components fit together.

So that the CLDC virtual machine can run within a small footprint and also to take into account additional security requirements for CLDC devices, CLDC differs from CDC in the following respects:

- no floating point support (although it has been added for CLDC 1.1) – this means that `float` and `double` numbers cannot be used and alternative means of storing these values have to be found, for example, "string math"
- no finalization – the `Object.finalize()` method does not exist (`Object.finalize()` is used to carry out any tidying up that may

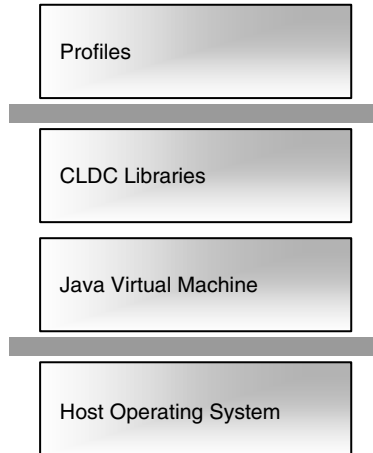


Figure 1.2 High-level architecture.

be needed when an object is collected by the garbage-collector. However, there is little, if any, practical need for this method.)

- limited error handling – only three error classes exist: `java.lang.Error`, `java.lang.OutOfMemoryError` and `java.lang.VirtualMachineError`
- no Java Native Interface (JNI) – this is due to security concerns and the overhead exerted by JNI on the device memory
- no user-defined class loaders – the built-in class loader cannot be overridden, for security reasons
- no reflection
- no thread groups and daemon threads – although threading is available, thread groups cannot be created (however, `Thread` arrays can be created if a similar effect is required)
- no weak references, although these will be added to CLDC 1.1.

1.2.1.2 Core Libraries

A number of classes have been inherited from J2SE. To maintain the relationship between J2ME configurations and J2SE, it was decided that each class has to have the same name and that each package name must be identical or a subset of the corresponding J2SE class. The semantics of the class must remain the same; methods included in the subset shall not be changed. This means that classes may not be added to a package if they do not exist in J2SE.

The following outlines the classes that are available in CLDC 1.0 (a full listing of these packages can be found in Appendix 1):

- system classes – J2SE includes several classes that are closely tied into the Java virtual machine; for example, the `javac` compiler requires certain functions from the `String` and `StringBuffer` classes
- data type classes – `Boolean`, `Byte`, `Short`, `Integer`, `Long` and `Character` are supported under CLDC; `Double` and `Float` are not supported
- collection classes – `Vector`, `Stack` and `Hashtable` are available, together with interfaces such as `Enumeration`
- input/output classes – `Reader`, `Writer`, `InputStreamReader` and `OutputStreamWriter` are required in order to support internationalization
- calendar and time classes – a small subset of the `java.util` classes `Calendar`, `Date` and `TimeZone` are included; only one time zone is supported by default, although device manufacturers may implement additional ones
- additional utility classes – the `java.util` classes `Random` and `Math` have been included to provide a pseudo-random number generator and methods such as `min`, `max` and `abs`, respectively
- exception classes – as the CLDC classes are compatible with J2SE libraries, CLDC classes throw the same exceptions as J2SE classes; there is, therefore, a fairly comprehensive list of exception classes (see Appendix 1)
- error classes – in contrast to the exception classes, the error handling capabilities of CLDC are limited to the three error classes seen previously
- internationalization – CLDC provides support for the translation of Unicode characters to and from byte streams; just as J2SE uses readers and writers, J2ME uses the following constructors:

```
new InputStreamReader(InputStream is);
new InputStreamReader(InputStream is, String name);
new OutputStreamReader(OutputStream os);
new OutputStreamReader(OutputStream os, String name);
```

The constructors that define a string parameter can name the encoding scheme. If it is not named, the default encoding (stored in the system property `microedition.encoding`) is used. Additional converters may be used by certain implementations. An `UnsupportedEncodingException` will be thrown if the specified converter is not present. CLDC does not support localization such as time and currency formatting. If necessary, these can be added to an application's logic.

- property support – `java.util.Properties` provides support for the limited set of properties available in CLDC.

The properties are obtained by making a call to `System.getProperty(String, key)`. This method returns some limited property information about the device itself, such as the configuration version, platform name, character encoding and supported profiles. It also returns the values of the properties defined by each optional package supported by the device.

1.2.1.3 Networking and I/O

Networking on CLDC devices has been streamlined so that the programmer does not have to fully understand the underlying device capabilities. The Generic Connection Framework (GCF) has been created, streamlining the implementation of networking within applications. This also helps provide a smaller footprint.

Networking and I/O are implemented using the same interface. All connections are created using a single static method in a system class called `Connector`. There are six basic interface types addressed by this framework, although the actual implementation of any of these protocols is governed by the profile rather than by CLDC:

- basic serial input
- basic serial output
- datagram communication
- connection-orientated, i.e. TCP/IP
- notification mechanism for client–server communications
- basic web server connections.

Creating the connections is rather simple and, regardless of the type of connection, the format is the same. Here is a list of some common examples:

- HTTP:
`Connector.open("www.foo.com");`
- Sockets:
`Connector.open("socket://192.168.0.1:9000");`
- Datagrams:
`Connector.open("datagram://192.168.0.1");`

This minimizes the differences between one protocol and another and uses a text string (the parameter to the `open()` method) to categorize the type of connection required. This approach means abstractions within application modules remain the same when communication changes from one form to another. Essentially, the binding of the protocols is

carried out at runtime. At implementation level, the `open()` parameter (up to the first `:/`) instructs the system to obtain the desired protocol from the location where the protocol implementations are stored. This late binding allows an application to dynamically adapt to use different protocols at runtime.

1.2.1.4 Security

Implementing a full J2SE-style security policy requires a large amount of memory that is not available to typical CLDC devices. CLDC therefore implements a simpler domain-based security model, which specifies:

- Java classes are properly verified and guaranteed to be valid Java applications; the classes are pre-verified at build time, which means that the CLDC implementation has much less to do to verify a JAR file
- only a limited, predefined set of Java APIs is available to the application programmer: those defined by CLDC, the profiles and optional packages
- the downloading and management of applications on the device takes place at the native code level inside the virtual machine; no user-definable class loaders are provided
- the set of native functions accessible to the virtual machine is closed, meaning that the programmer cannot download new libraries containing native functionality; native functions other than those associated with the Java libraries provided by the configuration or profile cannot be accessed
- the programmer cannot override the system classes provided in the packages `java.*`, `javax.microedition.*` and other profile or system-specific packages; this is governed by a class lookup which is performed during class verification and provides the reason for the pre-verification stage of MIDlet (the basic MIDP application structure) packaging.

Further security measures may, of course, be implemented by the profile, as shall be seen in Section 1.2.2.

1.2.2 MIDP

The Mobile Information Device Profile (MIDP) combined with CLDC provides a more focused platform for mobile information devices, such as mobile phones and entry-level PDAs. MIDP provides the vertical integration required to make the Java runtime environment applicable to these devices by providing direction for the base environment provided by CLDC.

The MIDP specification has been revised under JSR 118 (Symbian is one of the contributors to the JSR 118 expert group). MIDP 2.0 extends

the original definition in a number of ways and provides a platform which enables developers to create highly graphical, audio-capable, networked applications for mobile devices. A maintenance release, MIDP 2.1, is being specified.

Supported by many integrated development environments, MIDP has become a widely-accepted platform and has been deployed on many mobile devices around the world. If developers take the approach that they can "write once and tweak everywhere", they can leverage the underlying technology to distribute enterprise, utility and entertainment applications to a wide and varied audience.

The introduction of over-the-air provisioning has standardized the method by which applications may be deployed to end-users. Users can browse web or WAP sites to locate applications and the Application Manager System (AMS) checks for versioning and compatibility with the host device and manages local installation. MIDP is also optimized to provide a graphical user interface for mobile devices, regardless of input method and screen size.

1.2.2.1 MIDP Packages

The MIDP 2.0 specification offers developers seven packages with which they may create applications. The packages are derived from CLDC as well as providing additional classes, which can be found under `javax.microedition.*`. This follows the rule that all packages and classes inherited from J2SE must follow the same naming conventions. All new classes not inherited from J2SE must be given a new naming convention, hence the creation of the `javax.microedition` package nomenclature.

Inherited classes

These classes are inherited from J2SE via CLDC:

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

MIDP 2.0 classes

These classes extend the CLDC environment and provide user interface, gaming, MIDlet application framework, persistent storage, multimedia, network and security classes. Details of these classes can be found in Appendix 2:

- `javax.microedition.io` provides networking support based upon the Generic Connection Framework defined in CLDC
- `javax.microedition.lcdui` provides a standard set of user interface classes

- `javax.microedition.lcdui.game` is new to MIDP 2.0 and provides a game development framework aimed at speeding up the game development process
- `javax.microedition.media` is new to MIDP 2.0 and provides basic audio functionality such as playback and simple tone generation
- `javax.microedition.media.control` is new to MIDP 2.0 and defines the specific `Control` types that can be used with a `media Player`
- `javax.microedition.midlet` provides the MIDlet framework
- `javax.microedition.rms` provides persistent storage for applications, even when the MIDlet is not running; a “best effort” is also made by the device implementation to retain data during power loss
- `javax.microedition.pki` is new to MIDP 2.0 and provides end-to-end security for MIDlets by the introduction of registered domains; trusted MIDlets can be installed and given extra access to the device.

1.2.2.2 Core Functionality

Mobile User Interface (LCDUI)

MIDP provides a set of standard components to aid the creation of portable, intuitive user interfaces. These classes reduce the development time and also reduce the size of the final application.

The standard classes include screen objects, which hold objects such as choice groups, lists, pop-up alerts and progress bars. Forms can be created to capture user input via text entry components, read-only fields and custom items. All screen and form objects are device-aware and provide support for native display, input and navigation techniques. MIDP 2.0 also sees the introduction of the `CustomItem` class, which allows developers to define their own form items.

Multimedia and Game Functionality

MIDP provides an ideal opportunity for developers to create game and other entertainment content for mobile devices. A set of low-level APIs allows the developer to take control of the screen at pixel level. Graphics can be animated and user input can be captured. The Game API adds game-specific control over animation with its framework implementation managing sprites, collision detection, layers and tiled layers. Built-in multimedia support is also provided with the Mobile Media API (MMAPI), an optional MIDP package that adds video and other multimedia functionality. MIDP also has a subset of the MMAPI which provides support for simple tone generation and playback of WAV files.

The Game API has been added as part of MIDP 2.0 and further consolidates the case for Java being a game development platform for mobile devices. Coupled with over-the-air provisioning, this offers a

strong business case for generating revenue streams from users obtaining entertaining applications whilst on the move. The provision of this game development framework leaves the designer more time to work on game-play, rather than having to repurpose home-made animation classes to suit another application. This also reduces application size and optimizes animation routines by permitting extensive use of native code, hardware acceleration and device-specific image data formats, as required.

The Game API provides a manager for sprites and layers, as well as providing an implementation for creating complex tiled layers. The layer manager keeps an index of all screen objects registered with it and renders them on screen as required when calls are made to its `paint()` method.

The Media API has been created for MIDP 2.0 as a subset of the larger Mobile Media API (MMAPI), developed under the Java Community Process JSR 135. When the MMAPI was developed it was recognized that smaller constrained devices, such as mobile phones, would not be able to accommodate its full complement. Wisely, it was recognized that not all mobile devices would, for example, have cameras so making this compulsory would be ineffective. The MIDP 2.0 Media API therefore sets out to provide upwards audio compatibility with MMAPI. The Media API provides the ability to perform simple tone generation, audio playback of WAV files, and general media controls such as start, stop and volume control.

Extensive Connectivity

Developers can enable their applications to communicate over a network as required (see Section 2.1.3.2). Interfaces are available for communication over `http`, `https`, datagrams, sockets and serial ports. MIDP also supports the SMS capabilities of GSM and CDMA networks through the optional Wireless Messaging API (WMA). WMA 2.0 even supports MMS capabilities. A specific device may not provide support for all of these protocols.

Communication with third parties can also be created using an event-based networking model. MIDP supports a server push model based upon a push registry which keeps track of registered third party inbound communications from the network. When information arrives, the device can start the registered MIDlet (this may depend on user approval). This enables developers to create turn-based games, for example, or to create enterprise applications which receive alert-based data such as financial or field sales information, and integrate that information directly with an application.

Over-the-Air Provisioning

Although MIDP 1.0 did not officially encapsulate an over-the-air provisioning (OTA) definition, it did recommend a practice that was adopted as an addendum to the original specification and has now been made a part of the MIDP 2.0 specification. This means that deployment and updating

of applications over-the-air now falls within the MIDP specification. It has therefore been standardized and defines how applications are discovered, installed and removed on MIDP devices. The most useful consequence of this is that status reports can now be produced. This greatly enhances the revenue model for MIDP applications because applications can be tracked as they are installed, updated or removed.

Persistent Storage

MIDP also implements a simple record-based database management system. The data will remain present across multiple invocations of a MIDlet. The platform is responsible for making its best effort to maintain the integrity of the data throughout normal use of the device, including rebooting and battery changes. However, when the associated MIDlet suite is removed, so are the record stores. MIDP 2.0 now allows explicit sharing of data across MIDlet suites, assuming the serving data store has given permission for this sharing.

End-to-End Security

With greater network connectivity and the nature of common application installation methods, a robust security model has been specified. HTTPS leverages existing standards such as SSL and WTLS and enables the transmission of encrypted data. Security domains are used to identify trusted and untrusted MIDlets. By default, all applications are untrusted and are prevented from accessing any privileged functionality. Access can be gained by signing the MIDlet to specific domains defined on the device using the X.509 PKI standard.

This allows mobile phone operators and manufacturers to improve the user experience by limiting the capabilities of unknown applications. Developers see application credibility and user confidence increased by having their applications reviewed, and deemed trusted, by operators or manufacturers in order to access advanced capabilities. Depending on the security policy of the device, a user may also choose to allow unknown applications full or temporary access to advanced capabilities.

1.3 CDC and Personal Profile

1.3.1 CDC

The Connected Device Configuration (CDC) has been developed under the Java Community Process, by JSR 36. Symbian was a member of the expert group that developed it. The configuration has been designed for devices with more memory, faster processors and greater network bandwidth than those using CLDC. Examples of such devices include TV set-top boxes, residential gateways, in-vehicle telemetry and high-end PDAs.

With this in mind, it is easier to understand that CDC was designed with the aim of being based upon the J2SE 1.3 APIs while providing support for resource-constrained devices. This leaves a route open for existing J2SE developers to leverage their skills and also provides a path for the creation of secure enterprise-type applications for constrained devices.

CDC offers more facilities than CLDC. It provides a full Java 2 virtual machine including floating point and core library features, such as custom class loading, thread support and security. Like CLDC, it is a subset of the full J2SE implementation; the classes have been optimized to create a smaller memory footprint and some J2SE libraries have modified interfaces. An example of this is that the `javax.microedition.io` package provides the generic connection interface for input/output and networking.

Target devices are expected to have the following minimum specification:

- 32-bit CPU
- 2 MB RAM
- 2 MB ROM.

The Java environment for these devices is completed with the addition of one of three profiles which sit on top of the CDC classes to form the complete implementation. The CDC profiles, which are layered, are as follows:

- the Foundation Profile (JSR 46) is the most basic CDC profile; it provides the basic application support classes such as network and I/O support but does not provide a graphical user interface
- the Personal Basis Profile (JSR 129) provides all of the Foundation Profile APIs and a structure for building lightweight component toolkits and support for the Xlet application model
- the Personal Profile (JSR 62) provides full AWT, applet and limited bean support as well as the Foundation and Personal Basis Profiles; it represents a migration path for PersonalJava technology.

We shall have a close look at the Personal Profile in Section 1.3.2.

1.3.1.1 Core Libraries

The following core packages are available within the CDC configuration:

- `java.io` provides the system input and output through data streams, serialization and the file system

- `java.lang` provides the classes that are fundamental to the design of the Java language, for example, `Object`, which is the root of the class hierarchy
- `java.lang.ref` provides the reference-object classes, which support a limited degree of interaction with the garbage collector
- `java.lang.reflect` provides the classes and interfaces for obtaining reflective information about classes and objects
- `java.math` provides classes for performing arbitrary-precision integer (`BigInteger`) and decimal (`BigDecimal`) arithmetic
- `java.net` provides the classes for implementing networking applications
- `java.security` provides the classes and interfaces for the security framework
- `java.security.cert` provides the classes and interfaces for parsing and managing certificates
- `java.text` provides classes and interfaces for handling text, dates, numbers and messages in a manner independent of natural languages
- `java.util` provides the classes which contain the collections framework, legacy collection classes, event model, date and time facilities, internationalization and miscellaneous utility classes such as the string tokenizer and random number generator
- `java.util.jar` provides classes for reading and writing the JAR file format, which is based upon the standard ZIP file format with an optional manifest file
- `java.util.zip` provides classes for reading and writing the standard ZIP and GZIP file formats
- `javax.microedition.io` provides the classes for generic connections.

1.3.1.2 *Optional Packages*

The optional packages give device manufacturers the ability to support additional technologies if they so wish:

- RMI provides a subset of the J2SE RMI for Java-based network devices; it exposes distributed application protocols (through Java interfaces, classes and method invocations) and shields the application developer from the details of network communications

- JDBC provides a subset of the JDBC 3.0 API, which can be used to access flat files and tabular data sources such as spreadsheets; it also provides cross-DBMS connectivity to a range of SQL databases.

1.3.2 Personal Profile

The Personal Profile provides a further way of specifying the subset of APIs for a CDC-based device. Its definition is based upon the Java Community Process JSR 62, for which Symbian was a member of the expert advisory group.

As we have seen earlier, profiles provide a more specialized environment for devices common to a particular configuration. The Personal Profile is aimed at devices that require full GUI or internet applet support, such as communicators or game consoles. It is the successor to PersonalJava, which was developed prior to the formalization of J2ME, and therefore provides a clear migration path for PersonalJava applications to the J2ME platform.

The Personal Profile builds upon the Foundation Profile and the Personal Basis Profile by adding graphical user interface classes to the environment. It inherits networking and Xlet capabilities from the other two profiles. It has been designed to provide full graphical support and the ability to run web-based applets designed for the desktop to mobile device applications with web fidelity. The following outlines the core packages included in the Personal Profile and from where they are derived.

Added by the Foundation Profile

The following packages provide full J2SE 1.3.1 support for basic class library packages:

- `java.io`
- `java.lang`
- `java.lang.ref`
- `java.net`
- `java.security`
- `java.security.acl`
- `java.security.cert`
- `java.security.interfaces`
- `java.security.spec`
- `java.text`
- `java.util`

- `java.util.jar`
- `java.util.zip`

The following package provides compatibility for the CLDC 1.0 generic connection framework:

- `javax.microedition.io`

Added by the Personal Basis Profile

The following packages provide support for lightweight components and some 2D Java graphics:

- `java.awt`
- `java.awt.color`
- `java.awt.event`
- `java.awt.image`

The following package provides bean support by an external bean editor (IDE) running on a J2SE-based JRE:

- `java.beans`

The following packages provide limited RMI support for Xlets and are not intended for general-purpose use:

- `java.rmi`
- `java.rmi.registry`

The following packages provide Xlet support:

- `javax.microedition.xlet`
- `javax.microedition.clet.ixc`

Added by the Personal Profile

The following package provides support for applets:

- `java.applet`

The following packages provide support for heavyweight components and 2D graphics:

- `java.awt`
- `java.awt.datatransfer`

1.4 J2ME on Symbian OS

Java on Symbian OS has a long history dating back to Symbian OS Version 5 (released in 1999). This initial Java offering was based on Sun's JDK 1.1.4 platform. For the next major release, Symbian decided to take advantage of the reduced memory footprint offered by PersonalJava (compared to the burgeoning JDK) and used the PersonalJava 1.1.1 specification as the basis for the Java implementation. This release, Symbian OS Version 6.0, became available in 2000.

PersonalJava was the forerunner of J2ME and the first attempt by Sun to provide a Java environment for the more resource-constrained embedded device. It is the direct antecedent of the CDC-based Personal Profile.

In 1999, acknowledging that "one size doesn't fit all", Sun announced the splitting of Java into three versions:

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

Symbian immediately became involved in shaping the Micro Edition via the expert groups of the Java Community Process. Soon it was clear that J2ME MIDP was gaining momentum in the wireless space as phone manufacturers endorsed the idea of a lightweight Java environment suitable for mass-market phones. Symbian recognized the strength of the MIDP movement by including J2ME CLDC/MIDP 1.0 as its standard Java offering in Symbian OS Version 7.0, released in 2002, as well as back-porting the technology to earlier versions. Currently, all Symbian OS phones available in Western markets support at least MIDP 1.0.

Although MIDP 1.0 generated considerable enthusiasm amongst the wireless Java community, it was also realized that MIDP 1.0 on its own was limited in its capabilities to access the functionality offered by a typical smartphone from within a MIDlet. Consequently, soon after the release of MIDP 1.0, the wireless Java community started work on enhancing the capabilities of MIDP. This has manifested in MIDP 2.0 (JSR 118), released in its final form in November 2002, and a range of extension API JSRs, all forming part of the Java Community Process.

These developments provide a substantial increase in the functionality available to MIDlets. As a consequence, the latest release of Symbian OS (Version 7.0s) and UIQ 2.1 move to a single Java technology stream based on J2ME CLDC and MIDP 2.0 (plus additional optional J2ME APIs).

J2ME MIDP is now established as the ubiquitous Java platform in the mobile phone arena and, as such, Symbian will continue to evolve and enhance its CLDC/MIDP offering. For more insight into future developments of J2ME on Symbian OS, including Symbian's position with regard to CDC-based technologies, the reader is referred to Chapter 8.

1.5 Summary

This chapter has introduced the J2ME architecture in order to indicate the position of MIDP 2.0 within that structure. We have examined the various configurations and profiles and shown why they are necessary in providing a structure for the various needs and requirements of a J2ME device now and in the future. We have outlined the packages and classes of CLDC 1.0 and MIDP 2.0 to show their core functionality and have also shown how J2ME and Symbian sit together.

In Chapter 2 we are going to examine MIDP 2.0 in more depth, start programming a simple MIDP 2.0 application and look at the some of the various tools on offer.