Beginning Java™ ME Platform

ISBN-13 (pbk): 978-1-4302-1061-0

ISBN-10 (pbk): 1-4302-1061-3

ISBN-13 (electronic): 978-1-4302-1062-7

ISBN-10 (electronic): 1-4302-1062-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Christopher King
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Richard Dal Porto
Copy Editor: Nicole Abramowitz
Associate Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Patrick Cunningham
Proofreader: Liz Welch
Indexer: Brenda Miller
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 1

Mapping the Java Jungle

Although at its heart Java ME is really just an adaptation of the Java language, class libraries, and concepts to fit constrained devices, the business behind Java ME is in fact quite different. A firm grasp of the Java ME market, platform, and terminology will put you in good stead to developing successful products using Java ME.

In this chapter, I begin by introducing the market for Java ME. Next, I take you on a tour of the Java ME platform, showing you how Sun identified and defined the basic requirements for mobile platforms, and how manufacturers, carriers, and others have extended this basic platform. Finally, I discuss how the process of application development for Java ME is different, and I show you how important it is to know your audience, target devices, and distribution channel.

After reading this chapter, you will understand why Java ME differs from Java. You will see how device manufacturers, wireless operators, and consumers view Java ME, and how Java ME meets the needs of all of these parties. Armed with this knowledge, you'll be able to better manage a Java ME development project.

Introducing the Market for Java ME

A trio of forces dominates the Java ME market: *device manufacturers* looking to differentiate their products in the marketplace, *wireless operators* seeking to differentiate services and raise the average revenue per user (ARPU), and *consumers* personalizing their devices in new and novel ways.

Looking from the Device Manufacturers' Perspective

The interplay between device manufacturers and wireless operators is complex. Manufacturers are in constant competition with each other to differentiate their products, while at the same time, in many markets they are beholden to wireless operators to meet stringent requirements for features and functionality.

Device manufacturers can be broadly separated into two categories: *original equipment manufacturers* (OEMs) and *original design manufacturers* (ODMs). OEMs build

devices under their own label and sell devices to consumers (either directly or via the operator, or most often both), while ODMs design and build hardware on behalf of others. While both ODMs and OEMs must differentiate their product on the basis of price, quality, and features, for OEMs brand and marketing also become key concerns.

Many of today's wireless operators simply require a Java ME runtime on most of the phones that they provide to subscribers. As I discuss in the next section, "Looking from the Operators' Perspective," operators are seeking ways to raise revenue per subscriber, and data services are one way to do this. Today, data services consist of more than just wireless web services; many Java ME applications rely on the network for their content. Requiring handset manufacturers to include Java ME on their devices leaves an open door for developers to create new applications that provide operators with new sources of revenue.

Providing Java ME on devices is more than just an operator requirement for many manufacturers. Some manufacturers, including Research In Motion (RIM), offer Java ME runtimes that both meet Java ME standards as well as include additional classes in their implementation, enabling developers to build novel applications atop the phone's fundamental platform. More frequently, however, you'll find the baseline Java ME implementation on a device. In either case, Java ME enables device manufacturers to build and bundle applications for their products more quickly than with existing embedded toolkits.

This is especially true for the growing number of dedicated devices that connect via home or municipal wired and wireless networks where the use of Java ME may not be a mandate. Java ME provides a ready alternative to closed, proprietary platforms for writing application software for wireless Internet devices, set-top boxes, and other embedded systems. Even when the end platform is closed to third-party developers, selecting Java ME can help device manufacturers bring their product to market by providing a more powerful and well-understood platform than an internally defined or purely embedded alternative.

Whether chosen because of a customer requirement, as an opportunity for differentiation, or to speed product development, Java ME provides important advantages over other platforms. Unlike its larger cousins, Java Platform, Standard Edition (Java SE) and Java Platform, Enterprise Edition (Java EE), Java ME has been carefully tuned to run on small devices, important for meeting the cost and power constraints of most devices today. It's an open platform, encouraging contributions of technologies through the Java Community Process (JCP). Finally, Java ME brings with it the entire community of Java developers, providing a pool of talented engineers, designers, and project managers from which to draw.

Looking from the Operators' Perspective

Wireless operators today face challenges, too. While differentiation on the basis of quality and brand remain important, chief among challenges is the drive for higher ARPU. Revenue from voice activity has largely leveled off, making data services an obvious area in

which to drive growth. This is especially true in some countries, including the United States, where carriers have spent huge sums of money obtaining the rights to new parts of the wireless spectrum.

While arguably the wireless Web, Short Message Service (SMS), and Multimedia Messaging Service (MMS) all play a role in contributing to the bottom line for data services, mobile applications play a growing role as well. Java ME applications can contribute in two ways: by driving data use itself and by providing revenue when an operator acts as the channel for application distribution. Increasingly, just as you can buy a ring tone or wallpaper for your handset via the operator's wireless web portal, you can now buy Java ME applications as well. As you will learn later in this chapter (in "Marketing and Selling Your Application"), partnerships between developers and operators establish important channels for application sales, bringing revenue to both parties.

While the bottom line drives business decisions, marketing and brand image play an increasing role in Java ME's importance for operators. By opening their network to third-party developers such as yourself, operators bring your creativity to the table. Moreover, Java ME enables operators additional ways to partner with key brands around the globe, helping the operators differentiate themselves and giving brands far removed from the mobile computing market an opportunity to interact with consumers in new ways.

Looking from the Consumers' Perspective

Today's consumers demand more from their devices. Whether using a cell phone, set-top box, or dedicated appliance, consumers expect clear value. Reliability, ease of use, personalization, and network awareness are features becoming increasingly important even for traditionally isolated devices. Java ME fits the bill as a platform on which to base these devices, because it's small, highly portable, and powerful.

In the wireless telecommunications market, consumer demand for reliability, ease of use, personalization, and network awareness has already begun and continues to grow. As I write this, Sun estimates that more than two *billion* wireless terminals have shipped with a Java runtime since the initial launch of Java for handsets. These devices support communication, entertainment, multimedia, and other applications bringing customization and choice to mobile device users around the world.

Java is poised to repeat this success within the set-top box market after years of persistent effort. With Java on every Blu-ray player as well as countless set-top boxes for personal entertainment, the potential for new applications is almost boundless. This market will be more diverse than the mobile market, with room for both small players and large application development houses as well as the traditional entertainment content partners (many of whom were latecomers to the wireless telecommunications marketplace).

Through all of this, Java ME enables developers to provide subscribers with greater choice, freedom, and flexibility. The fundamental platform enables developers to create stand-alone applications as well as network-aware applications and games, while

enabling manufacturers to add more interfaces to tap the custom features of the underlying platform.

Some pundits have accused the Java ME marketplace of "wagging the dog" to some extent—that is, driving consumer demand for applications for the sake of technology itself. While this claim is not wholly unwarranted, it's also the nature of a fast-paced market in which new products are tested on the market, and only those with successful business cases and clear value to consumers will survive. Java ME accelerates this process by providing an open standard on which to base the development of new products and services.

Looking Inside the Java ME Platform

The Java ME platform isn't really one platform, but rather a collection of platforms and libraries that work on a host of mobile devices. Even more confusing, Java ME began as a mobile environment for cell phones and personal digital assistants (PDAs), but has since expanded to include devices with similar constraints, including industrial devices, set-top boxes, Internet appliances, and other constrained platforms.

Justifying the Need for a Mobile Edition of Java

At first, the need for a mobile edition of Java may not be apparent. After all, today's cell phones are more powerful than the PCs that ran the first commercially available versions of Java more than a decade ago. However, a key feature of Java ME is its size and performance footprint. This is especially important for the constraints common to mobile and embedded devices. The constraints that mobility puts on size, power consumption, and cost mean that less capable processors and less memory will be found in devices for the foreseeable future. These constraints apply to less-mobile devices such as set-top boxes, too; in designing a commodity consumer device, every penny counts, so frequently low-cost (slower) processors and less memory are available. Moreover, as green manufacturing increasingly comes into play, fixed consumer devices will be subject to the same sorts of power constraints as mobile devices.

But Java ME isn't just about footprint. It's also about a new way of looking at computing. Many of the differences between Java SE and Java ME are about functionality, not footprint. The Java Application Descriptor (JAD) file is one example; it describes an application, including its name, icon, publisher, and other information. Or take Java ME's security model, which includes the notion of privileges and permissions for interfaces. Using Java ME, running applications may require individual privileges to perform sensitive operations, such as sending an SMS message, requesting the position of the handset via the location-based interface, or exchanging data using Bluetooth. Privileges are specified in the JAD file of a Java ME application and may be granted or denied depending on the origin of the application. (I discuss this in more detail in the "Packaging and Executing CLDC/MIDP Applications" section in Chapter 3.)

It's important to realize that the ultimate goal for Java ME is to provide an extensible yet highly portable, minimum-footprint, Java implementation that can run on a wide variety of network devices with constant or intermittent network connectivity. The platform emphasis is on application-level, not system-level, programming, and the application programming interfaces (APIs) that are supported reflect this distinction. Extensibility is another distinction, especially for one flavor of Java ME: the Connected Limited Device Configuration, which I discuss in the "Introducing the Connected Limited Device Configuration" section later in this chapter. Extensibility is a key differentiator between the platform, other Java platforms, and other computing platforms as a whole.

Making Java Work on Mobile Devices

Chapter 2 looks at the changes made to Java and its base classes for Java ME in close detail, but it's worth summarizing these changes now:

- The Java runtime must have the ability to reject invalid Java class files to ensure system security and integrity.
- The Java runtime controls an application's access to specific parts of the system (such as the file system, network access, and so forth).
- Applications run within a sandbox that prevents unauthorized access to other applications and libraries.
- The environment must support the ability to download new applications, but cannot use this mechanism to modify or override protected system classes in any way (including changing the order in which classes are looked up).
- The platform libraries may lack specific interfaces for performance and memory use reasons; for example, one configuration of Java ME doesn't support object finalization, nor does it have support for the Java Abstract Window Toolkit (AWT) or Swing user-interface libraries.
- The platform may or may not have support for floating-point mathematical operations, depending on the version.

For brevity, I've painted this list with a broad brush; not all of these changes apply to all flavors of Java in the Java ME family, as you'll learn in the next section and Chapter 2.

Because of the immense variety in devices supported by Java ME, there are actually different implementations of Java for different devices. Specifically, how Java ME functionality is defined for a specific device is based on three concepts:

- *Configuration*: Defines the basic set of libraries and Java virtual machine (VM) for a broad range of devices
- *Profile*: Defines a set of APIs for a narrower range of devices
- *Package*: A set of optional APIs that pertain to a specific technology, such as multimedia or Bluetooth access

Figure 1-1 shows how these abstractions stack to define the software characteristics of a device.

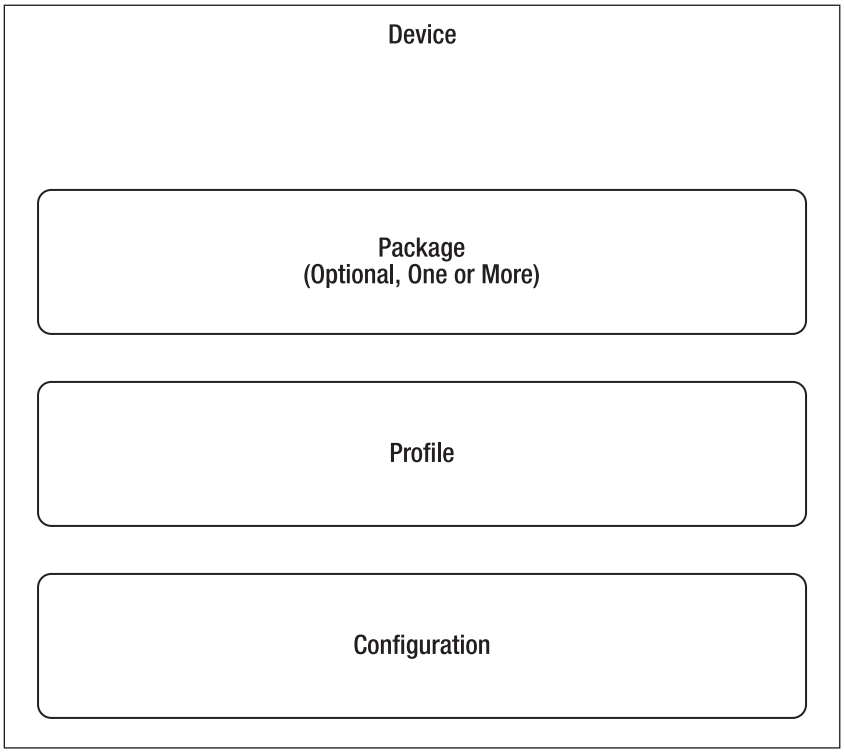

Figure 1-1. *The relationship between configurations, profiles, and packages in Java ME*

Each instance of a configuration, profile, or package *should* have as its basis one or more Java Specification Requests (JSRs) that document the purpose and interface for the Java extension in question. Appendix A summarizes the JSRs relevant to Java ME that define functionality discussed in this book.

Today, there are two configurations within Java ME: the Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC). There are a handful of profiles that sit atop either of these configurations, and many, many packages. (In the next section, I explore these configurations in greater detail.)

The most commonly known tuple of configuration, profile, and package is based on the CLDC package, including the Mobile Information Device Profile (MIDP) accompanied by optional packages, powering billions of the world's mobile phones today. In fact, this configuration is *so* common that many think of it as the only version of Java ME. Figure 1-2 shows a typical mobile-phone configuration with a few common optional packages. I discuss the MIDP in depth in the "Introducing the Mobile Information Device Profile" section later in this chapter.

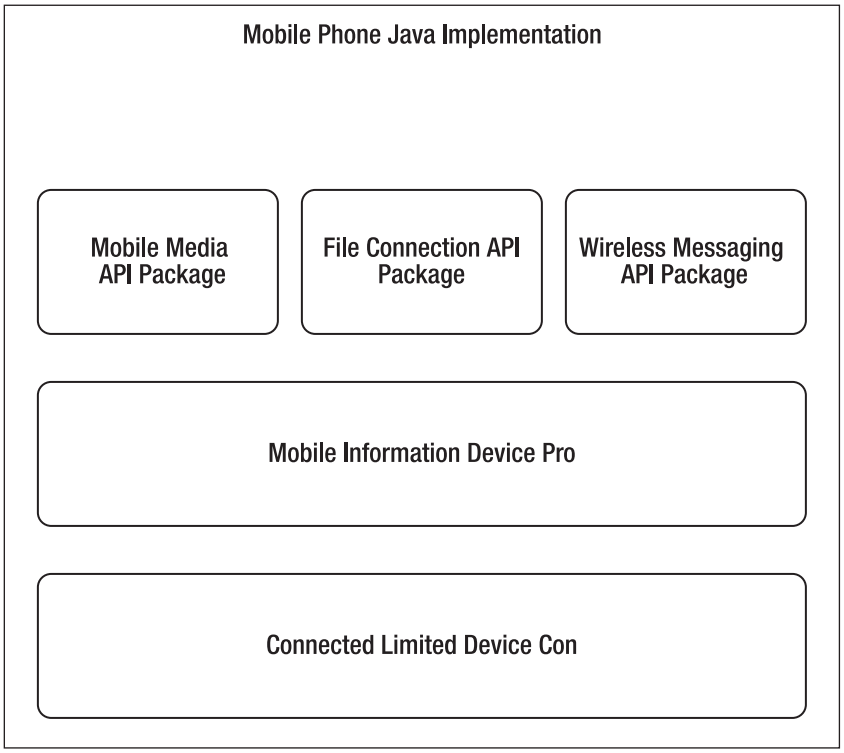

Figure 1-2. *The relationship between configurations, profiles, and a few of the packages in a typical cell phone*

Another, increasingly common tuple is based on the CDC package and powers set-top boxes and other devices. Such a digital media configuration is for higher-end connected devices, and it incorporates the Foundation Profile as well as the Personal Profile, and perhaps additional optional packages. Unlike the CLDC-MIDP configuration, this configuration defines a subset of the Java AWT, bringing a well-understood graphics library to mobile devices. Figure 1-3 shows a typical configuration based on the CDC for today's consumer electronics devices.

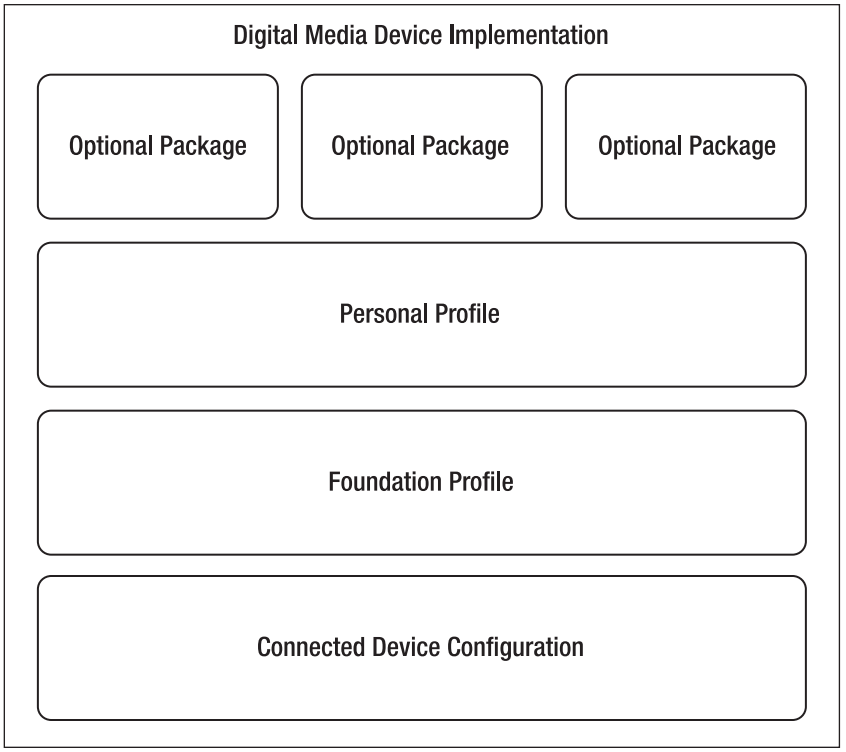

Figure 1-3. *The relationship between configurations, profiles, and packages in a typical set-top box or other digial media hardware*

Understanding Configurations

The fundamental building block of the Java ME architecture is the configuration. A configuration defines the Java Virtual Machine (JVM), basic language support, and fundamental classes for the widest possible set of devices, such as all mobile wireless terminals or all set top box–like devices. A specific device must meet the requirements of at least one configuration, such as the CLDC, in order to qualify as a Java ME device.

Introducing the Connected Limited Device Configuration

The CLDC is explicitly designed for memory-constrained devices that may be always or intermittently connected to the network. It defines a Java environment based on the Java K virtual machine (KVM), a modified version of the JVM that has been specially tuned to operate on low-power devices with a 16- or 32-bit processor and as little as 192KB of RAM. The most common deployment of the CLDC, as I've already noted, is in the billions of Java-enabled cell phones that have shipped in the last several years.

A device with the CLDC is quite primitive; it's missing many of the classes you'd expect to find that permit application development (such as user-interface widgets, network connectivity, and the like). Instead, the implementation leaves those details to profiles atop the CLDC, and focuses instead on reducing memory and CPU footprint to the absolute minimum necessary for the Java environment. One area obviously affected is the nature of the libraries included in the CLDC; many Java SE classes are simply not available in the CLDC, and of those that are available, not all methods may be available. This is especially true for collections, where only three classes and one interface are available. (For more details on exactly what *is* supported, see Chapter 2.)

While the JVM historically absorbs the entire burden of Java bytecode verification, this is not the case for the CLDC. Because bytecode verification is expensive both in terms of processor and memory, the responsibility of bytecode verification is shared between the developer and the KVM running on the mobile device. As part of the build process, you run a tool called preverify that inlines subroutines in each class file (removing certain bytecodes in the process) and adds information about variables on the stack and their types. At runtime, the KVM uses this additional information to complete the bytecode verification prior to execution.

■ **Note** Two-pass preverification obviously brings with it potential security issues, because a malicious developer could inject code that appears to be preverified but does not meet all of the standards required by the CLDC. To address this, CLDC applications are typically downloaded from trusted sources; moreover, the profile used with the CLDC—the MIDP—adds code signing, so that a Java implementation can verify the originator of the code being executed and provide an appropriate level of trust.

But security changes don't end there. The sandbox model, familiar to applet developers from the early days of Java, plays a much greater role in the CLDC, where applications are sandboxed from each other as well as the host operating system. Each application runs in an environment where the only facilities it can access are those classes in its own distribution and the local classes provided by the device's configuration, profile, and optional packages. The download, installation, and management of other Java applications are also inaccessible, preventing one application from affecting another. Finally, there is no facility for creating native methods, closing potential holes between the sandbox and the native platform.

Another key feature of the CLDC is the Generic Connection Framework (GCF), which defines a hierarchy of interfaces that generalize port connections. The resulting hierarchy provides a single set of classes you use to connect to any network or remote port, including Transmission Control Protocol over IP (TCP/IP), User Datagram Protocol (UDP), and serial ports, just to name a few. The GCF is defined in the `javax.microedition.io` package.

Due to processor and power constraints, CLDC 1.0 did not support floating-point mathematics; CLDC 1.0 devices could only perform integer math. This changed with the advent of CLDC 1.1, as CLDC 1.1 devices must support floating-point operations. However, as a developer, you should be aware that floating-point mathematics remains computationally expensive. Be careful when choosing to use them, as they can cause performance issues within your applications.

Introducing the Connected Device Configuration

The CDC is for devices that are more capable than those used by the CLDC, such as high-end PDAs, set-top boxes, and other Internet appliances. As such, the goals of the CDC are slightly different than those of the CLDC. Instead of targeting the largest possible number of low-cost hardware, the CDC focuses on leveraging developer and technology skills from the existing Java SE platform while respecting the needs of resource-constrained devices.

Unlike the CLDC, the CDC virtual machine meets the same requirements as the JVM that powers Java SE. In fact, if you add to the CDC the profile and packages usually found on a media-capable device, you'll find little that distinguishes the environment from a Java SE platform except the *extra* APIs that additional packages may provide. This is the strength of Java, and especially the more robust CDC: you can leverage your Java skills across the entire product family line. In addition, the CDC includes all of the Java language APIs required of the CLDC, including the GCF.

Packages containing classes defined by the CDC include `java.lang`, `java.io`, `java.net`, `java.security`, `java.text`, and `java.util`. Devices running the CDC must also support CLDC APIs and packages; this enables the fullest possible support for all Java applications. While such devices are rare on the market today, it's an obvious future direction for Java, as the majority of devices continue to become more and more powerful.

Understanding Profiles

Profiles collect essential APIs for building the most fundamental of applications across a family of devices. The most well known profile by far is the MIDP, which powers mobile phones and sits atop the CLDC. Equally important is the Foundation Profile, which is analogous to the MIDP for providing application support classes. Unlike CLDC-based devices, however, CDC-based devices typically also include either the Personal Basis Profile or the Personal Profile, or both, to provide user-interface support for applications.

Introducing the Mobile Information Device Profile

The MIDP is the foundation of today's mobile-phone Java revolution (Part 2 of this book is dedicated to the MIDP). Defining a number of new interfaces, it enables you to develop *MIDlets*, which are applications that run atop the CLDC. The MIDP defines these other interfaces, as well:

- The application life cycle via the MIDlet
- Networking
- Persistent storage
- Sound
- Timers
- The user interface (including the portable display and input as well as support for games)

However, the MIDP defines more than just interfaces and the classes that support those interfaces. It also describes how applications are installed on a device. While the actual implementation may differ from device to device, the general requirement is that from the device you're able to browse applications, select one for download using HTTP/1.1, and have the device install the MIDP and present it in its application manager. Applications are accompanied by an application descriptor (see the "Packaging and Executing CLDC/MIDP Applications" section in Chapter 3) that includes information about the application, such as the application vendor, application name, and application size.

The MIDP defines the notion of *permissions*, which indicate that a MIDlet can access a restricted API. The only permission defined by the MIDP pertains to network connections, but packages are free to introduce other permissions. A permission is a name using the same prefix and class or interface name as the name of the restricted API. For example, to use a network socket, a MIDlet needs the permission `javax.microedition.io.Connector.socket`. This permission accompanies the application in the descriptor file, and a MIDlet can test for the presence of a privilege using the MIDlet method `checkPermission`, which returns `1` if the permission is granted, `0` if it is not, and `-1` if the result is indeterminate, such as a requirement that the user be asked to grant permission manually.

The MIDP also defines the notion of a *trusted application*, which is permitted to use a restricted API, such as the file connection package, to access the file system. An application gains trust by virtue of the domain from which it came; typically the application carries this information through a cryptographic signature applied by a certification authority or carrier. As a result, even when bearing privilege, applications must be prepared for security exceptions, which may occur when an application is untrusted and attempts to access a restricted API.

Introducing the Foundation Profile

The Foundation Profile, targeted for CDC-enabled devices, runs on devices with less than 256KB of ROM (of course, a Foundation Profile device can have more ROM, but that's the minimum supported by the profile), a minimum of 512KB of RAM, and a persistent network connection. In many ways, the Foundation Profile is far less ambitious than the MIDP. In conjunction with the CDC, it provides network and input/output (I/O) support, but no classes for application development; those are relegated instead to the Personal Basis Profile and the Personal Profile.

Classes augmented by the Foundation Profile include those in the `java.lang`, `java.io`, `java.net`, `java.security`, `java.text`, and `java.util` packages. These classes are based on classes found in the Java SE 1.4 class hierarchy, as well as additional `javax.microedition.io` classes that provide HTTP and HTTP-over-Transport Layer Security (TLS) network operations.

Introducing the Personal Basis Profile

Most CDC-based devices have at least some user-interface requirements. The most highly embedded devices may use only the Foundation Profile with a custom package atop that to provide support for a custom-made liquid-crystal or light-emitting diode (LED) display, but by far the most common are devices with raster displays that need rich graphical user interfaces (GUIs). To accommodate this in a standard way, two profiles are available. The smaller of the two, the Personal Basis Profile, actually provides *two* class hierarchies for applications: the applet model, and a new hierarchy for media devices that defines the Xlet programming model. Xlets are similar to applets, except they have a life cycle that supports being paused and resumed, which is important for media devices in which multiple applications and media streams may interrupt an application's execution at any time.

The Personal Basis Profile also defines a subset of the AWT for GUI development. Unlike the traditional AWT, the Personal Basis Profile defines a *lightweight* control facility, in which user-interface components draw themselves rather than have peer controls derived from the native platform. (The Java Swing implementation takes the same approach of having the Java environment draw its own controls.) Consequently, the Personal Basis Profile only includes support for `java.awt.Window` and `java.awt.Frame` (which hook to the native platform's windowing manager and contain lightweight components), and `java.awt.Component` and `java.awt.Container` (which are lightweight components used to create all of the other components in the hierarchy). Note that the Personal Basis profile does not define traditional AWT controls, including buttons, lists, and other items, because these would have connections to peer components from the native platform. Instead, you can create your own components or import a package on a specific platform that provides the components you need.

Finally, the Personal Basis Profile also includes classes that support communication between Xlets, using a subset of Java's Remote Method Invocation (RMI) API. It's important to remember, though, that while parts of the `java.rmi` package are included in the Personal

Basis Profile, the profile does *not* support RMI; just enough of the RMI implementation is included to facilitate communication between two Xlets running on the same device.

Introducing the Personal Profile

The Personal Profile is a superset of the Personal Basis Profile that provides support for the entire AWT, as well as limited JavaBean support. Some readers may remember PersonalJava, the predecessor to Java ME that was targeted for higher-end Internet appliances and set-top boxes; the Personal Profile atop the CDC is the forward migration path for applications running on PersonalJava.

In fact, the Personal Profile is almost the same as Java SE 1.4.2, with these differences:

- Support for RMI is available through an optional package (`java.rmi`).
- Support for SQL is available through an optional package (`java.sql`).
- Support for Java Swing is available through an optional package (`java.swing`).
- Support for Object Management Group (OMG) interfaces, including Common Object Request Broker Architecture (CORBA), is available through an optional package (`org.omg`).
- There is no support at present for the Java Accessibility API (`javax.accessibility`).
- There is no support at present for the Java Naming and Directory Interface (JNDI) in `java.naming`.
- There is no support for the Java Sound API (`java.sound`).
- Support for JavaBeans is limited to runtime support; there is no support for bean editors running directly on a CDC environment.
- The applet API `getAccessibleContext` is not supported.
- Applet and AWT methods deprecated from Java SE have been removed from all supported classes.

Understanding Packages

A package, as its name implies, is an object or group of objects in a common container. As important as platforms and profiles are to the modularity of the Java ME platform, packages are arguably the key to Java ME's continued success, as they permit Sun and third-party vendors to extend the Java ME platform for specific families of devices.

There are countless packages for Java ME; many of the now-standard interfaces that are part of successful MIDP-based devices are in fact packages. In this book, you will learn how to use several packages, including

- The GCF, documented in JSR 30
- The `FileConnect` interface, which provides local file access on MIDP and is documented in JSR 75
- The Java Bluetooth API, documented in JSR 82
- The Wireless Messaging API, documented in JSR 120
- The Web Services API, documented in JSR 172
- The Java Advanced Graphics and User Interface (AGUI) API, for CDC devices, documented in JSR 209
- The Java Mobile Service Architecture (MSA), documented in JSR 248

Planning Your Approach to Java ME Development

Java ME's strength is rooted in the ubiquity of Java today. However, with this ubiquity comes challenges. The multitude of APIs and the diversity of distribution channels make planning the technical and business aspects of your application equally important.

Selecting Appropriate Device Targets

As you've seen, the triad of configurations, profiles, and packages means managing a *lot* of different APIs. For many kinds of applications, this may not be a serious problem—most productivity and network applications need just a network layer, some GUI elements, and a persistent store, which is available under just about any combination of configuration and profile you might encounter.

That's not always the case, however. Your application might depend on functionality present in only a specific package, perhaps, either by design (say, a Bluetooth-derived application for proximity detection) or product differentiation. There's always the temptation of using an optional package to speed time to market, too, only to find later that it's not available on the next target for your product.

Consequently, if portability is important, you should base your application on as few Java ME packages as you possibly can. Obviously, this doesn't mean creating your own control framework from scratch or implementing your own web services framework from scratch if you don't have to. But it does mean understanding what APIs are

available on the devices you're targeting, and performing regular surveys of the market. I find it best to keep track of what APIs I plan to use as I design and implement my application, and correlate them against the JSRs that define those APIs. Then, when it's time to bring my application to market on a different device, I can check to see which packages are offered by the new hardware and scope my porting effort accordingly. Sites like the Wireless Universal Resource File (WURFL) device description repository at `http://wurfl.sourceforge.net` and the support database for J2ME Polish (a library originally targeted at Java ME's predecessor, J2ME, that simplifies cross-device development) at `http://devices.j2mepolish.org/` are invaluable in planning your product launch or porting efforts. A little research as you design your application can pay big dividends when rolling it out to consumers.

Marketing and Selling Your Application

There are a myriad of channels for Java ME applications today, each with their own set of business challenges. Many readers see wireless operators as the logical channel for application distribution, given the popularity of the MIDP today. Still others target web distribution to hardware directly or bundle applications with hardware at manufacturing time. A small percentage of you may be working directly with platform or hardware manufacturers, and so your channel to the consumer (and the notion of a consumer itself!) is quite different.

You can distribute your application to consumers in a number of ways. Certainly direct distribution is a possibility, by publishing a link to your application (see Chapter 3 for how to package your application for the different configurations). This may sound simple, but it poses an obvious business question: How will you get paid for your application? Free distribution, advertising, and per-download or subscriptions via credit card or PayPal fulfillment are all possibilities.

Because of the need for privilege by most applications and revenue by most developers, a typical deployment for a mobile application involves both third-party certification and business negotiations. The process of third-party certification typically involves a business program such as Java Verified, which tests your application, and assuming it passes testing, cryptographically signs your application for distribution. Usually accompanying this signed endorsement is access to additional privileges; for example, most MIDP implementations won't permit HTTP transactions without prior user approval unless a member of the Java Verified testing authority has signed the application. Once signed, you can distribute your application through aggregators, who broker the carrier relationship on behalf of you and many other developers, reimbursing you for sales (typically via a premium-SMS push or direct billing). Another distribution path is to negotiate with one or more wireless operators to distribute your application. This involves crafting the business relationship—how much will consumers pay for your application, and how will it be obtained?—as well as additional testing, which usually results in a second cryptographic signature for your application.

With this signature may come additional privileges, such as access to location-based APIs. Negotiating operator partnerships can be an expensive and time-consuming task, but without the help of an intermediate aggregator, you need to perform these negotiations for each operator's network on which you want to deliver your application.

This issue of privileges and cryptographic signatures isn't just a business issue, but can be a functional issue as well. This may very well affect the functional requirements for your application; for example, Acme Wireless (a fictitious operator) might only permit applications access to location-based interfaces for applications *it* signs and distributes. If your application needs positioning data (say, to locate the user to recommend restaurants or services), that requirement won't be just a technical requirement but a business requirement to establish the necessary operator relationship to ensure the required privilege. This admittedly adds risk to your business model, because it's difficult to ascertain in advance whether or not specific operators will carry your application, and obtaining this information in advance can involve significant business investment. It does, however, give you certain assurances if you can bridge the gap and obtain operator signing and distribution for your application, because this gives significant placement and marketing clout for your application. While all of these comments apply primarily to Java ME MIDP applications, due to the large number of MIDP applications and MIDP-capable devices on the market, I suspect that the landscape for Java CDC applications will be similar, given the rampant success of the MIDP distribution model.

Of course, if you're a developer involved in planning your business strategy, don't forget other avenues for distribution, too. Direct-to-manufacturer deals, while difficult to obtain, are potentially lucrative sources of long-term revenue. Moreover, providing services for Java consulting remains an important mainstay for many developers. Thinking creatively, a host of business models can support your application's development and distribution.

Wrapping Up

Java ME meets needs for operators, device manufacturers, and consumers. By providing a rich set of platforms across devices from low-end mobile phones to high-end set-top boxes, and from embedded devices to high-end PDAs, Java ME enables software developers like you to leverage your skills to bring new applications to market.

The Java ME platform is divided up into configurations, platforms, and profiles. Two configurations are presently available: the CLDC, which is targeted for constrained devices, and the CDC, which is targeted for more robust devices. Atop these configurations are one or more profiles, such as MIDP for the CLDC, or the Foundation, Personal Basis, and Personal Profiles for the CDC. These provide additional APIs in a standard way that let you write your application for a family of devices. Finally, packages allow Sun and third parties to extend the APIs on a device or suite of devices in a standard way, giving access to new functionality in portable ways.