J2ME: Step by step

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Tutorial tips

Should I take this tutorial?

This tutorial examines in detail the Java 2 Platform, Micro Edition (J2ME) and targets the intermediate developer who has a solid background in Java programming and the concepts of object-oriented design and development.

You'll start off by examining the background of J2ME and exploring the J2ME configurations and profiles. You'll then step through the process of setting up your development environment for developing J2ME applications.

You'll be introduced to topics such as the K virtual machine (KVM) and KJava API used in conjunction with the Connected Limited Device Configuration (CLDC) API, and the Mobile Information Device Profile (MIDP), which uses CLDC. You will then build a simple application that will allow you to see what you can do with J2ME. You'll use CLDC and KJava to develop a basic drawing application and also a small MIDP application.

Requirements

To work through this tutorial, you will need the following:

* The *Java 2 SDK* (formerly known as a JDK); J2SE 1.3 SDK is recommended. In particular, you will use the following tools from the Java 2 SDK:
  - java *
  - javac *
  - jar *
  - javadoc (optional)

* The *Connected Limited Device Configuration (CLDC)* reference implementation
* The K virtual machine (KVM), which is included with the CLDC reference implementation
* The *Mobile Information Device Profile (MIDP)*
* The *Palm OS Emulator (POSE)* , which you can use to test your KJava applications before deploying to a "real" Palm OS device.
* A Palm handheld device

Getting help

For technical questions about J2ME, visit the *Java Developer Connection* .

For questions about the content of this tutorial, contact the authors, Shari Jones, at *shari_jones@mindspring.com* , or Steven Gould, at *73774.2356@compuserve.com* .

Shari Jones is a freelance journalist and a technical writer. She is a former consultant and has more than ten years experience writing technical articles and documentation covering all areas of the high-tech industry.

Steven Gould is an Executive Consultant with CGI Information Systems. Based in Dallas, he

is a systems architect and senior developer, focusing primarily on Java and C++ development under Windows and various UNIX platforms. A Sun-certified Java developer, Steven has been using Java since the JDK 1.0 beta release.

Section 2. J2ME overview

Introduction

This section will get you started using J2ME. We'll begin by defining J2ME, then we'll discuss its general architecture and learn about the devices J2ME targets. As part of the architecture discussion, we will provide an overview about profiles and configurations. (We'll address the details of both profiles and configurations in later sections.) We also will cover briefly some considerations for packaging and deploying J2ME applications.

What is J2ME?

Sun Microsystems defines J2ME as "a highly optimized Java run-time environment targeting a wide range of consumer products, including pagers, cellular phones, screen-phones, digital set-top boxes and car navigation systems."

Announced in June 1999 at the JavaOne Developer Conference, J2ME brings the cross-platform functionality of the Java language to smaller devices, allowing mobile wireless devices to share applications. With J2ME, Sun has adapted the Java platform for consumer products that incorporate or are based on small computing devices.

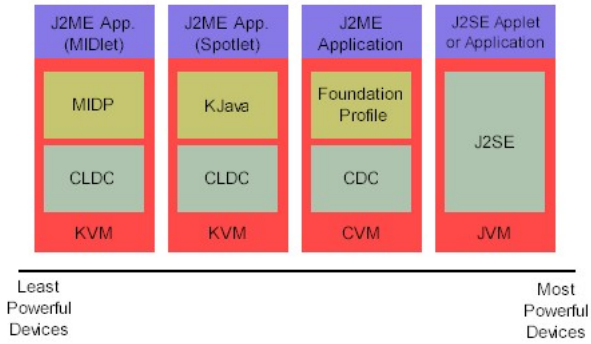
General J2ME architecture

J2ME uses configurations and profiles to customize the Java Runtime Environment (JRE). As a complete JRE, J2ME is comprised of a configuration, which determines the JVM used, and a profile, which defines the application by adding domain-specific classes.

The *configuration* defines the basic run-time environment as a set of core classes and a specific JVM that run on specific types of devices. We'll discuss configurations in detail in the section on J2ME configurations on page 10.

The *profile* defines the application; specifically, it adds domain-specific classes to the J2ME configuration to define certain uses for devices. We'll cover profiles in depth in the section on J2ME profiles on page 12.

The following graphic depicts the relationship between the different virtual machines, configurations, and profiles. It also draws a parallel with the J2SE API and its Java virtual machine. While the J2SE virtual machine is generally referred to as a JVM, the J2ME virtual machines, KVM and CVM, are subsets of JVM. Both KVM and CVM can be thought of as a kind of Java virtual machine -- it's just that they are shrunken versions of the J2SE JVM and are specific to J2ME.



Configurations overview

The configuration defines the basic run-time environment as a set of core classes and a specific JVM that run on specific types of devices. Currently, two configurations exist for J2ME, though others may be defined in the future:

* **Connected Limited Device Configuration (CLDC)** is used specifically with the KVM for 16-bit or 32-bit devices with limited amounts of memory. This is the configuration (and the virtual machine) used for developing small J2ME applications. Its size limitations make CLDC more interesting and challenging (from a development point of view) than CDC. CLDC is also the configuration that we will use for developing our drawing tool application. An example of a small wireless device running small applications is a Palm hand-held computer. We will cover CLDC in depth in the section on J2ME configurations on page 10.
* **Connected Device Configuration (CDC)** is used with the C virtual machine (CVM) and is used for 32-bit architectures requiring more than 2 MB of memory. An example of such a device is a Net TV box. CDC is outside scope of this tutorial, although we will cover it briefly later on in the section on CDC API on page 34.

Profiles overview

The profile defines the type of devices supported by your application. Specifically, it adds domain-specific classes to the J2ME configuration to define certain uses for devices. Profiles are built on top of configurations. Two profiles have been defined for J2ME and are built on CLDC: KJava and Mobile Information Device Profile (MIDP). These profiles are geared toward smaller devices.

A skeleton profile on which you create your own profile, the Foundation Profile, is available for CDC. However, for this tutorial, we will focus only on profiles built on top of CLDC for smaller devices.

We will discuss both of these profiles in later sections and will build some sample applications using KJava and MIDP.

Devices J2ME targets

Target devices for J2ME applications developed using CLDC generally have the following characteristics:

* 160 to 512 kilobytes of total memory available for the Java platform
* Limited power, often battery powered
* Network connectivity, often with a wireless, inconsistent connection and with limited bandwidth
* User interfaces with varying degrees of sophistication; sometimes with no interface at all

Some devices supported by CLDC include wireless phones, pagers, mainstream personal digital assistants (PDAs), and small retail payment terminals.

According to Sun Microsystems, target devices for CDC generally have the following characteristics:

* Powered by a 32-bit processor
* Two megabytes or more of total memory available for the Java platform
* Devices that require the full functionality of the Java 2 "Blue Book" virtual machine
* Network connectivity, often with a wireless, inconsistent connection and with limited bandwidth
* User interfaces with varying degrees of sophistication; sometimes with no interface

Some devices supported by CDC include residential gateways, smartphones and communicators, PDAs, organizers, home appliances, point-of-sale terminals, and car navigation systems.

J2ME versus J2SE versus J2EE

This graphic depicts the devices that support J2ME applications and illustrates where J2ME fits into the Java platform:

Section 3. Developing J2ME applications

Introduction

In this section, we will go over some considerations you need to keep in mind when developing applications for smaller devices. We'll take a look at the way the compiler is invoked when using J2SE to compile J2ME applications. Finally, we'll explore packaging and deployment and the role preverification plays in this process.

Design considerations for small devices

Developing applications for small devices requires you to keep certain strategies in mind during the design phase. It is best to strategically design an application for a small device before you begin coding. Correcting the code because you failed to consider all of the "gotchas" before developing the application can be a painful process.

Here are some design strategies to consider:

* **Keep it simple.** Remove unnecessary features, possibly making those features a separate, secondary application.
* **Smaller is better.** This consideration should be a "no brainer" for all developers. Smaller applications use less memory on the device and require shorter installation times. Consider packaging your Java applications as compressed Java Archive (jar) files.
* **Minimize run-time memory use.** To minimize the amount of memory used at run time, use scalar types in place of object types. Also, do not depend on the garbage collector. You should manage the memory efficiently yourself by setting object references to null when you are finished with them. Another way to reduce run-time memory is to use lazy instantiation, only allocating objects on an as-needed basis. Other ways of reducing overall and peak memory use on small devices are to release resources quickly, reuse objects, and avoid exceptions.

Design considerations for mobile devices

The same rule of thumb applies for mobile device development that we mentioned for small device development: design and then code. Let's examine some design recommendations to consider when developing applications for mobile devices:

* **Let the server do most of the work.** Move the computationally intensive tasks to the server, and let the server run them for you. Let the mobile device handle the interface and minimal amounts of computations, and let the server do the intensive work. Of course, the mobile device for which you are developing plays a factor in how easy and how often the device can connect to a server.
* **Choose the language carefully.** J2ME still is in its infancy and may not be the best option. Other object-oriented languages, like C++, could be a better choice, depending on your needs.

Performance considerations

Code for performance. Here are some ways to code with the aim to achieve the best performance:

* **Use local variables.** It is quicker to access local variables than to access class members.
* **Avoid string concatenation.** String concatenation decreases performance and can increase the application's peak memory usage.
* **Use threads and avoid synchronization.** Any operation that takes more than 1/10 of a second to run requires a separate thread. Avoiding synchronization can increase performance as well.
* **Separate the model using the model-view-controller (MVC).** MVC separates the logic from the code that controls the presentation.

Compiling considerations

As with any other Java application, you compile the application before packaging and deploying it. With J2ME, however, you use the J2SE compiler and need to invoke it with the appropriate options.

In particular, you need to use the `-bootclasspath` option to instruct the compiler to use the J2ME classes, not the J2SE classes. Do not place the configuration classes in the compiler's CLASSPATH. This approach will lead to run-time errors, because the compiler automatically searches the J2SE core classes first, regardless of what is in the CLASSPATH. In other words, the compiler will not catch any references to classes or methods that are missing from a particular J2ME configuration, resulting in run-time errors when you try to run your application.

Packaging and deployment considerations

Because J2ME is designed for small devices with limited memory, much of the usual Java preverification has been removed from the virtual machine to allow for a smaller footprint. As a result, it is necessary to preverify your J2ME application *before* deployment. An additional check is made at run time to make sure that the class has not changed since preverification.

Exactly how to perform the preverification, or checking the classes for correctness, depends on the toolkit. CLDC comes with a command-line utility called preverify, which does the actual verification and inserts extra information into the class files. MIDP uses the wireless toolkit, which comes with a GUI tool, though this too can be run from the command line.

Deployment depends on the platform to which you are deploying. The application must be packaged and deployed in a format suitable for the type of J2ME device, as defined by the profile.

Section 4. J2ME configurations

What are the J2ME configurations?

As you learned earlier, the configuration defines the basic run-time environment as a set of core classes and a specific JVM that run on specific types of devices. You also learned that the two types of configurations for J2ME are CLDC and CDC.

Sun provides J2ME configurations that are suitable for different market segments -- CLDC for small devices and CDC for larger devices. A J2ME environment can be configured dynamically to provide the environment needed to run an application, regardless of whether or not all Java technology-based libraries necessary to run the application are present on the device. The core platform receives both application code and libraries. Configuration is performed by server software running on the network.

In the next few panels, you will learn more about CLDC and CDC and which profiles they are associated with.

Connected Limited Device Configuration (CLDC)

CLDC was created by the Java Community Process, which has standardized this "portable, minimum-footprint Java building block for small, resource-constrained devices," as defined on Sun Microsystems' Web site.

The J2ME CLDC configuration provides for a virtual machine and set of core libraries to be used within an industry-defined profile. As mentioned in Section 2, a profile defines the applications for particular devices by supplying domain-specific classes on top of the base J2ME configuration. The K virtual machine (KVM), CLDC's reference implementation of a virtual machine, and its KJava profile run on top of CLDC.

CLDC outlines the most basic set of libraries and Java virtual machine features required for each implementation of J2ME on highly constrained devices. CLDC targets devices with slow network connections, limited power (often battery operated), 128 KB or more of non-volatile memory, and 32 KB or more of volatile memory. Volatile memory is non-persistent and has no write protection, meaning if the device is turned off, the contents of volatile memory are lost. With non-volatile memory, contents are persistent and write protected. CLDC devices use non-volatile memory to store the run-time libraries and KVM, or another virtual machine created for a particular device. Volatile memory is used for allocating run-time memory.

CLDC requirements

CLDC defines the following requirements:

* Full Java language support (except for floating pointer support, finalization, and error handling)
* Full JVM support
* Security for CLDC
* Limited internationalization support
* Inherited classes -- all classes not specific to CLDC must be subsets of J2SE 1.3

classes
* Classes specific to CLDC are in `javax.microedition` package and subpackages

In addition to the `javax.microedition` package, the CLDC API consists of subsets of the J2SE `java.io`, `java.lang`, and `java.util` packages. We will cover details in the section on CLDC API on page 18and will use the CLDC API to develop our drawing application.

Connected Device Configuration (CDC)

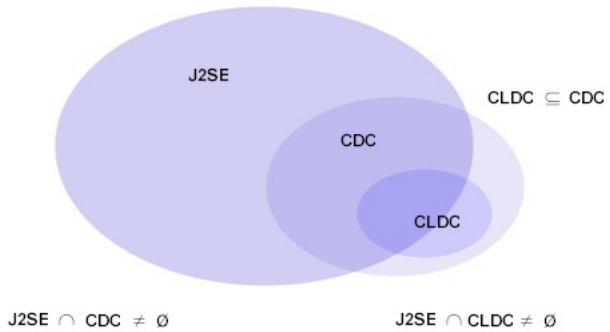
Connected Device Configuration (CDC) has been defined as a stripped-down version of Java 2 Standard Edition (J2SE) with the CLDC classes added to it. Therefore, CDC was built upon CLDC, and as such, applications developed for CLDC devices also run on CDC devices.

CDC, also developed by the Java Community Process, provides a standardized, portable, full-featured Java 2 virtual machine building block for consumer electronic and embedded devices, such as smartphones, two-way pagers, PDAs, home appliances, point-of-sale terminals, and car navigation systems. These devices run a 32-bit microprocessor and have more than 2 MB of memory, which is needed to store the C virtual machine and libraries. While the K virtual machine supports CLDC, the C virtual machine (CVM) supports CDC. CDC is associated with the Foundation Profile, which is beyond the scope of this tutorial.

We'll look into CDC in more detail in the section on CDC API on page 34.

CLDC versus CDC

The following graphic depicts the relationship between CDC and CLDC. It also illustrates their relationship to the full J2SE API. As we saw earlier in this section, CDC is a subset of J2SE plus some extra classes. We also saw that CLDC is a subset of CDC.



Section 5. J2ME profiles

What is a J2ME profile?

As we mentioned earlier in this tutorial, a profile defines the type of device supported. The Mobile Information Device Profile (MIDP), for example, defines classes for cellular phones. It adds domain-specific classes to the J2ME configuration to define uses for similar devices. Two profiles have been defined for J2ME and are built upon CLDC: KJava and MIDP. Both KJava and MIDP are associated with CLDC and smaller devices.

Profiles are built on top of configurations. Because profiles are specific to the size of the device (amount of memory) on which an application runs, certain profiles are associated with certain configurations.

A skeleton profile upon which you can create your own profile, the Foundation Profile, is available for CDC. However, for this tutorial and this section, we will focus only on the KJava and MIDP profiles built on top of CLDC.

Profile 1: KJava

KJava is Sun's proprietary profile and contains the KJava API. The KJava profile is built on top of the CLDC configuration. The KJava virtual machine, KVM, accepts the same byte codes and class file format as the classic J2SE virtual machine.

KJava contains a Sun-specific API that runs on the Palm OS. The KJava API has a great deal in common with the J2SE Abstract Windowing Toolkit (AWT). However, because it is not a standard J2ME package, its main package is `com.sun.kjava`. We'll learn more about the KJava API later in this tutorial when we develop some sample applications.

Profile 2: MIDP

MIDP is geared toward mobile devices such as cellular phones and pagers. The MIDP, like KJava, is built upon CLDC and provides a standard run-time environment that allows new applications and services to be deployed dynamically on end-user devices.

MIDP is a common, industry-standard profile for mobile devices that is not dependent on a specific vendor. It is a complete and supported foundation for mobile application development.

MIDP contains the following packages, the first three of which are core CLDC packages, plus three MIDP-specific packages. We will discuss each of these packages later on in the tutorial:

```
*   java.lang
*   java.io
*   java.util
*   javax.microedition.io
*   javax.microedition.lcdui
*   javax.microedition.midlet
```

* `javax.microedition.rms`

Move over KJava, here comes MIDP

As we mentioned earlier in this section, KJava is a proprietary Sun API. It was not intended as a complete fully featured profile, but rather as a demonstration of how a profile could work with CLDC. According to the CLDC Release Notes included with the CLDC download:

The GUI classes provided in package `com.sun.kjava` are not part of the Connected Limited Device Configuration (CLDC). Official GUI classes for Java 2 Platform, Micro Edition will be defined separately through the Java Community Process and included in J2ME profiles.

In spite of this fact, the KJava profile has been widely used by early adopters.

At the 2001 JavaOne Developer Conference, Sun announced Early Access availability of MIDP for Palm OS (initial releases of the MID Profile had focused mostly on wireless phones). The specification of MIDP for Palm OS was defined by the Java Community Process (JCP) and is, therefore, vendor independent. One of the current restrictions -- which will disappear over time -- is that MIDP for Palm OS requires the latest Palm OS, version 3.5.

At the time of this writing, the specification of MIDP for Palm OS is still under development by the JCP, and the details are subject to change. Although no date for its release has been set, when it becomes available it will eliminate the need for KJava.

Section 6. Setting up your development environment

Introduction

In this section, we will see how to download and install the necessary software required for developing J2ME applications. We begin by downloading and installing CLDC under Windows or UNIX. The current CLDC 1.0 release contains a CLDC implementation for Win32, Solaris, and Linux platforms. We'll then install the KVM on your Palm hand-held device and look at compiling the Palm database development tools.

Next, we'll show you how to download and install the Palm OS Emulator (POSE) and how to transfer a ROM image of a Palm device to your PC for use with the emulator.

Finally, we'll look at downloading and installing the J2ME Wireless Toolkit, which is used for developing J2ME applications for MIDP devices.

Downloading and installing CLDC on Win32 or UNIX

To install CLDC and the Sun's KVM software on a Windows or UNIX platform:

1. *Download the CLDC* . Two packages are downloaded and must be installed: j2me_cldc-1_0_2-fcs-winunix.zip and j2me_cldc-1_0_2-fcs-kjava_overlay.zip.
2. Unzip the first package.
   - Under Windows you may want to unzip the contents into the root directory, c:\
   - Under UNIX, unzip the contents into your preferred directory. This may be your home directory, or if you are installing for all users of the machine install wherever you normally install shared application files (for example, this is usually something like `/usr/local` or `/opt`.

   A new folder, j2me_cldc, is created beneath the installation directory, and it contains the following subdirectories: jam, docs, build, tools, api, kvm, samples, and bin.

3. Unzip the second package to the j2me_cldc directory just created by your CLDC installation. For example, c:\j2me_cldc under Windows, or something like /usr/local/j2me_cldc or /opt/j2me_cldc under UNIX. If prompted, overwrite any files that already exist.
4. Add the j2me_cldc/bin directory to your PATH to save typing the full path every time you want to run the CLDC programs kvm and preverify.

Installing CLDC and the KVM on your PDA

Use the HotSync feature of the PDA device to install the kvm.prc and kvmutil.prc files on your Palm OS handheld. To install these files from Windows:

1. Place the PDA in its cradle.
2. From the Palm Desktop, click the Install icon.
3. Click the Browse button to select the c:\j2me_cldc\bin directory.
4. Select the kvm.prc and kvmutil.prc files. Note that there are various other prc files in the same directory -- these contain some sample applications which you also may want to

install and try out.
5. Press the HotSync button on your PDA's cradle to install the selected prc files.
6. On your PDA, locate either of the files just installed. Tap on the icon for either of the two files on your PDA to load and run the application. You now can set the maximum heap size and screen output options.

Compiling the Palm database tools

By installing the CLDC packages, you have set up the development environment and now have access to the documentation and classes, which you will find under j2me_cldc/docs and j2me_cldc/bin/api/classes, respectively.

The j2me_cldc/tools directory, one of the other directories installed with the two packages, stores the utilities used to generate .prc files. These utilities allow you to install your J2ME applications on your PDA. To use the tools in the j2me_cldc/tools directory, you must first compile the class files:

1. Go to or change to the j2me_cldc/tools/palm directory.
2. Create a subdirectory called classes
3. Compile the .java source files by entering the following command on one line:
```
javac -d classes src/palm/database/*.java
```
The classes are now compiled and can be found in the classes subdirectory.
4. Copy the src/palm/database/Wrapper.prc and src/palm/database/DefaultTiny.bmp files to the classes directory.

You have now compiled the Palm database tools class files into the j2me_cldc/tools/palm/classes directory (and its subdirectories). You may want to add the full path to the classes subdirectory to your Java CLASSPATH. Alternatively, you can add it to the CLASSPATH specified on the `java` command line when you use the Palm database tools.

Installing the Palm OS Emulator (POSE)

The Palm OS Emulator (POSE) application software emulates different models of PDAs. An emulator allows you to develop, test, and debug PDA applications before downloading them to your PDA. POSE is free and available at the Palm OS Emulator Web site (see Resources on page 35).

A binary version of this is only available for Windows. Although POSE also is available to run under UNIX, you must download the source files from the Palm OS Web site and compile them yourself for your specific UNIX platform.

To download and install POSE under Windows:

1. Download the latest POSE zip file from *Palm OS Web site* .
2. Extract the contents of the zip file into its own directory.
3. The emulator.exe file now exists in the POSE installation directory. When launched, emulator.exe runs the Palm OS Emulator.

Uploading the ROM image

To use the emulator, you need a copy of the ROM (a "ROM image") for the target PDA. The ROM provides the exact operating system that is to be emulated by the POSE emulator. ROM images are available from some manufacturers, but you should be able to download a ROM image from your own PDA.

To view a ROM image of your hand-held computer on your desktop or workstation, you can use POSE to download the ROM image from your PDA.

Note: Handspring users must use a regular serial cable and COM port instead of a USB cable.

Here's how:

1. Start the Palm Desktop Software that came with your Palm device and click on the Install icon.
2. Browse to the POSE directory and select the ROM Transfer.prc file.
3. Place the Palm device in its cradle.
4. Press the HotSync button on the cradle to install the file. The ROM Transfer icon should display on your Palm device when the process is complete.
5. To transfer the ROM image, you must exit the HotSync manager, making certain it is not running on your PC. Leave your Palm device in the cradle.
6. Tap the ROM transfer icon on your Palm device.
7. On your PC, change to the POSE directory and run the emulator.exe program. The Emulator window displays.
8. On your PC, select the Download button from the menu. On your Palm device, tap the Begin Transfer button. The ROM image could take a few minutes to transfer.
9. When the transfer is complete, you are prompted to select a directory in which to save the file. Save it as palm.rom in the POSE directory.
10. From the Palm OS Emulator window, select New. The New Session Emulator window displays.
11. Select the appropriate settings from the New Session Emulator window, and use the Browse button to select the ROM file, palm.rom, just transferred.
12. Finally, click the OK button.

If the transfer was successful, you should see an image of your Palm device loaded on your PC or workstation monitor.

Downloading and installing the J2ME Wireless Toolkit

The J2ME Wireless Toolkit provides a complete development environment to write and test MIDP applications. The download includes tools, documentation, an emulation environment, examples, and a module to integrate with Forte for Java.

Currently, the J2ME Wireless Toolkit is available for Windows 98 Second Edition, Windows NT 4.0, and Windows 2000 only. Windows 95 is not supported. Solaris and Linux versions of the J2ME Wireless Toolkit are under consideration at the time of this writing.

Follow these steps to install the J2ME Wireless Toolkit under Windows:

1. *Download the J2ME Wireless Toolkit* .
2. Run the j2me_wireless_toolkit-1_0_1-fcs.exe program to install the Wireless Toolkit.

When you are prompted for the installation directory, be sure that the fully qualified path to the install directory does not contain any spaces. This will help you avoid possible problems later on when using the toolkit.

If you plan to use Forte for Java for development, select **Integrated setup** in the **Setup Type** dialog.

Section 7. CLDC API

Introduction

So far we have seen how CLDC fits into the bigger picture of J2ME and have set up our environment for development. In this section, we delve a little deeper into the CLDC API. The CLDC API is really just a subset of the J2SE `java.lang`, `java.io`, and `java.util` packages plus a new package -- `javax.microedition`. We will look at each of these packages in turn, highlighting the more important classes within each.

Although each of these classes exists in J2SE, the CLDC implementation of each class does not necessarily implement all methods supported by J2SE. You should check the CLDC API documentation to be sure which methods are supported. A copy of the documentation is in the j2me_cldc/docs directory of your J2ME CLDC installation. It is available in both PDF and javadoc formats.

---

java.lang

The CLDC `java.lang` package is a subset of the J2SE `java.lang` package. Perhaps the most notable omissions compared to J2SE are floating point operations and, in particular, floating point (`Float`) and double precision (`Double`) classes. These omissions have implications to all other classes if floating point is used.

When compared with J2SE v1.3 API, there are several other classes that are omitted from the CLDC API, including `ClassLoader`, `Compiler`, `InheritableThreadLocal`, `Number`, `Package`, `Process`, `RuntimePermission`, `SecurityManager`, `StrictMath`, `ThreadGroup`, `ThreadLocal`, and `Void`.

We describe the main classes available in the CLDC `java.lang` package in the tables on the next few panels. The use of each of these classes should already be familiar to any Java developer.

In addition to these core classes, you will also see that the `Runnable` interface is supported by CLDC, as are the `Exception`, `Error` and related classes.

---

java.lang core run-time classes

The core run-time classes for the `java.lang` package are:

* `Class` -- Represents classes and interfaces in a running Java application.
* `Object` -- As in J2SE, `Object` is the base class of all Java objects.
* `Runtime` -- Provides a way for a Java application to interact with the run-time environment in which it is running.
* `System` -- Provides several static helper methods, as with J2SE.
* `Thread` -- Defines a thread of execution for a Java program.
* `Throwable` -- The superclass of all errors and exceptions in the Java language.

---

java.lang core data type classes

The core data type classes in the `java.lang` packages are:

* `Boolean` -- Wraps the `boolean` primitive data type.
* `Byte` -- Wraps the `byte` primitive data type.
* `Character` -- Wraps the `char` primitive data type.
* `Integer` -- Wraps the `int` primitive data type.
* `Long` -- Wraps the `long` primitive data type.
* `Short` -- Wraps the `short` primitive data type.

---

java.lang helper classes

The helper classes for the java.lang package are:

* `Math` -- Contains methods for performing basic mathematical operations. Note that all the methods manipulating floating point values are omitted, leaving only the `abs()`, `min()`, and `max()` methods on `integers` and `longs`.
* `String` -- Represents `String` objects in Java, as in J2SE.
* `StringBuffer` -- Represents a string that can be modified, as in J2SE.

---

java.io input classes

The CLDC API contains many of the commonly used input classes from J2SE. In particular, the CLDC `java.io` package includes the following classes:

* `ByteArrayInputStream` -- Contains an internal buffer representing bytes that may be read from an input stream.
* `DataInput` -- An interface that provides for reading bytes from a binary input stream and translating them into primitive Java data types. `DataInputStream` provides an implementation of this interface.
* `DataInputStream` -- Allows an application to read primitive Java data types from the underlying input stream in a platform-independent way.
* `InputStream` -- An abstract class that is the superclass of all classes representing an input stream of bytes.
* `InputStreamReader` -- Reads bytes and translates them into characters according to a specified character encoding.
* `Reader` -- An abstract class for reading character streams.

Note that, as with the `java.lang` package, some of these classes may not contain all the methods supported by their J2SE cousin. In particular, the floating point and double precision methods are omitted.

---

java.io output classes

The CLDC API contains many of the commonly used output classes from J2SE. In particular, the CLDC java.io package includes the following output classes:

* `ByteArrayOutputStream` -- Implements an output stream where data is written into a bytes array.
* `DataOutput` -- An interface that provides for writing primitive Java data types to a binary output stream. `DataOutputStream` provides an implementation of this interface.
* `DataOutputStream` -- An output stream that allows an application to write primitive Java data types in a portable way.
* `OutputStream` -- An abstract class that is the superclass of all classes representing an output stream of bytes.
* `OutputStreamReader` -- Given characters, translates them into bytes according to a specified character encoding.
* `PrintStream` -- Adds a convenient way to print a textual representation of data values.
* `Writer` -- An abstract class for writing character streams.

Some of these classes may not contain all the methods supported by J2SE, such as the floating point and double precision methods.

---

java.util collections classes

The CLDC `java.util` package contains the most frequently used classes from the J2SE `java.util` package. These include four of the collections classes -- actually three classes and one interface -- as well as date/time and utility classes.

The `java.util` collections classes supported by CLDC are:

* `Enumeration` -- An interface that allows the calling routine to iterate through a collection of items, one at a time.
* `Hashtable` -- Implements a `hashtable`, which maps keys to values.
* `Stack` -- Represents a last-in-first-out (LIFO) collection or stack of objects.
* `Vector` -- Represents a resizable "array," or vector, of objects.

---

java.util -- other classes

The remaining `java.util` classes supported by CLDC include date and time classes and the `Random` utility class. These are summarized in the following table.

* `Calendar` -- An abstract class for getting and setting dates using a set of integer fields such as YEAR, MONTH, DAY, and so on.
* `Date` -- Represents a specific date and time, with millisecond precision.
* `Random` -- Utility class used to generate a stream of random `int` or `long` values.
* `TimeZone` -- Represents a time zone offset, and also handles daylight savings adjustments.

---

javax.microedition.io

So far, all the classes we have seen in the CLDC API have been a subset of the J2SE API. CLDC also includes one additional package -- the `javax.microedition.io` package.

The only class defined in this package is the `Connector` class, a factory class that contains methods to create `Connection` objects, or input and output streams.

`Connection` objects are created when a class name is identified dynamically. A class name is identified based on the platform name, as well as the protocol of the requested connection. The parameter string that describes the target adheres to the URL format required by the *RFC 2396 specifications* . Use the following format:

```
{scheme}:[{target}][{params}]
```

The `{scheme}` is the name of a protocol such as http or ftp. `{target}` is usually a network address, but non-network oriented protocols may treat this as a fairly flexible field. Any parameters, `{params}`, are specified as a series of assignments of the form ";x=y" (for example, `;myParam=value`).

---

javax.microedition.io helper interfaces

In addition to the generic connection factory class, the `javax.microedition.io` package also contains the following connection-oriented interfaces:

* `Connection` -- Defines the most basic type of connection. This interface is also the base class for all other connection interfaces in this package.
* `ContentConnection` -- Defines a stream connection over which content is passed.
* `Datagram` -- Defines a generic datagram interface.
* `DatagramConnection` -- Defines a generic datagram connection and the capabilities it must support.
* `InputConnection` -- Defines a generic input stream connection and the capabilities that it must support.
* `OutputConnection` -- Defines a generic output stream connection and the capabilities that it must support.
* `StreamConnection` -- Defines a generic stream connection and the capabilities that it must support.
* `StreamConnectionNotifier` -- Defines the capabilities that a stream connection notifier must have.

Section 8. Development using KJava GUI components

Introduction

In this section, we'll look at GUI development using the KJava API. We'll start with an introduction to KJava GUI development, and then get into the KJava API and develop our first J2ME application. This application, HelloWorld, will demonstrate a bare-bones J2ME application using CLDC, the KJava profile, and the KVM for the Palm OS.

We'll continue with KJava GUI development in the next section, building another application and focusing on the event handling model.

Introducing the Spotlet

The KJava API provides a set of classes for developing applications for Palm OS devices. KJava provides a `Spotlet` class, `com.sun.kjava.Spotlet`, which is similar to the J2SE `Canvas` class with the addition of some callback methods for handling events. Applications can, therefore, extend the `Spotlet` class and override the appropriate event handling methods to provide the required functionality.

Applications can create and use multiple spotlets to display different windows. Just like with the J2SE `Canvas`, a spotlet is responsible for drawing itself as well as any GUI controls placed upon it.

We will be using the `Spotlet` class in both of our KJava examples, a HelloWorld application, which we'll get to shortly, and the Scribble application, which we'll build in the section Development using KJava event handling on page 26.

The HelloWorld KJava application

This application displays "Hello World!" in the center of the screen, as well as an Exit button, which terminates the application when pressed. The HelloWorld.java file starts with the following lines of code to import the classes that will be used later in the `HelloWorld` class:

```
import com.sun.kjava.Button;
import com.sun.kjava.Graphics;
import com.sun.kjava.Spotlet;
```

The following line of code defines our `HelloWorld` class as extending `Spotlet`:

```
public class HelloWorld extends Spotlet
```

Remember that the `Spotlet` class provides callbacks for handling events. In this simple example, we're only interested in one event -- when the user taps the Exit button. The next line of code stores a reference to the Exit button:

```
private static Button exitButton;
```

As in J2SE, the `main()` method defines the main entry point for the program. For a J2ME

application, `main` also defines the entry point. In this case, the `main()` method creates a new instance of the `HelloWorld` class, which runs our application.

```
public static void main(String[] args)
{
   (new HelloWorld()).register(NO_EVENT_OPTIONS);
}
```

The next block of code defines the constructor. Within the constructor, we first create a new `Button` and give it the label "Exit." This button initially is invisible. We then get a reference to the graphics object, which is the drawable screen, clear the screen, then draw the text "Hello World!" in the center. Finally, we add the Exit button to the screen.

```
public HelloWorld()
{
   // Create (initially invisible) the "Exit" button
   exitButton = new Button("Exit",70,145);

   // Get a reference to the graphics object;
   // i.e. the drawable screen
   Graphics g = Graphics.getGraphics();
   g.clearScreen();

   // Draw the text, "Hello World!" somewhere near the center
   g.drawString("Hello World!", 55, 45, g.PLAIN);

   // Draw the "Exit" button
   exitButton.paint();
}
```

Finally, we define the `penDown` event handler, which simply checks to see if the Exit button was pressed, and if so, exits the application.

```
public void penDown(int x, int y)
{
   // If the "Exit" button was pressed, end this application
   if (exitButton.pressed(x,y))
     System.exit(0);
}
```

HelloWorld -- complete code listing

Following is the complete code sample for the HelloWorld application for a Palm device:

```
import com.sun.kjava.Button;
import com.sun.kjava.Graphics;
import com.sun.kjava.Spotlet;

/**
 * Simple demonstration, "Hello World" program. Note that Spotlet is
 * the class that provides callbacks for event handling.
 */
public class HelloWorld extends Spotlet
{
   /** Stores a reference to the "Exit" button. */
   private static Button exitButton;

   /**
    * Main entry point for this program.
    */
   public static void main(String[] args)
   {
      (new HelloWorld()).register(NO_EVENT_OPTIONS);
```

```
    }

    /**
     * Constructor: draws the screen.
     */
    public HelloWorld()
    {
       // Create (initially invisible) the "Exit" button
       exitButton = new Button("Exit",70,145);

       // Get a reference to the graphics object;
       // i.e. the drawable screen
       Graphics g = Graphics.getGraphics();
       g.clearScreen();

       // Draw the text, "Hello World!" somewhere near the center
       g.drawString("Hello World!", 55, 45, g.PLAIN);

       // Draw the "Exit" button
       exitButton.paint();
    }

    /**
     * Handle a pen down event.
     */
    public void penDown(int x, int y)
    {
       // If the "Exit" button was pressed, end this application
       if (exitButton.pressed(x,y))
         System.exit(0);
    }
}
```

KJava GUI components

In addition to the `Spotlet` class, the KJava API also defines some basic GUI components. Some of the more fundamental GUI components provided by KJava are listed here. Note their similarities with the J2SE AWT components by the same name.

* `Button` -- Defines a simple GUI push button. The button can contain either a text label such as "OK" or "Cancel", or a bitmap image.
* `Checkbox` -- Defines a GUI checkbox component that either can be checked or unchecked.
* `Dialog` -- Defines a pop-up, modal dialog box containing a title, a text string, and a "Dismiss" button.
* `Graphics` -- This class is similar to its J2SE cousin, and provides various methods for drawing on a display.
* `RadioButton` -- Defines a two-state radio button. Usually used as part of a group of radio buttons, grouped using a `RadioGroup` object, where only one can be on at a time.
* `RadioGroup` -- Represents a group of radio buttons, only one of which can be on, or selected, at a time.
* `ScrollTextBox, SelectScrollTextBox` -- Defines a scrolling text box component where the user can enter multiple lines of text. Similar functionality to the J2SE TextArea AWT component.
* `Slider` -- Defines a graphical slider, where the user can select a value by sliding a marker along a scale.
* `TextBox` -- Defines a basic text box, useful for displaying small amounts of text only. For larger quantities of text, consider using the ScrollTextBox.
* `TextField` -- Defines a text box that provides for user input. Similar to the J2SE

`TextField` AWT component.

* `ValueSelector` -- A GUI component that accepts an integer value from the user. The user can select "+" to increment the value or "-" to decrement the value.

Other KJava classes

KJava defines some additional classes. It is less likely that you will use these classes in your early development efforts, but it is worthwhile knowing what they do in case you should need to use them at a later point.

* `Bitmap` -- Represents a black and white bitmap image.
* `Caret` -- Used only by `TextField`. (The API documentation indicates that this class should probably be private to the `TextField` class.)
* `Database` -- Provides an interface to the Palm OS database manager.
* `DialogOwner` -- An interface to be used by any class wanting to display a modal dialog.
* `HelpDisplay` -- Defines a simple Help dialog.
* `IntVector` -- Not really a GUI component as such, this class provides an expandable vector of integers much like `java.util.Vector`.
* `List` -- Not really a GUI component, this is another helper class that represents a list of objects, like `java.util.Vector`.
* `ScrollOwner` -- Used by `ScrollTextBox`.
* `VerticalScrollBar` -- Defines a vertical scroll bar component.

Section 9. Development using KJava event handling

Introduction

In this section we will explore KJava event handling, and demonstrate how it works with our simple drawing application, Scribble.

The KJava event handling model is fairly rudimentary compared to J2SE's action-listener event handling model. All events of interest are accessible by subclassing the `Spotlet` class, which your KJava application will do anyway. Only the spotlet that currently has the focus is notified of the event. To give a spotlet the focus, use the `register()` method. To stop being notified of events, use the `unregister()` method.

Note: If you register a spotlet with WANT_SYSTEM_KEYS, the device will not terminate the application automatically by trapping the button press and queueing an application that stops it. Instead, the application will be notified of the button press event, and will then be responsible for handling that event appropriately. The application will continue to run indefinitely unless you provide a way to terminate it by calling `System.exit` when a button is pressed. Otherwise, the only way to stop the application is to reset the device.

KJava supports three basic types of events: pen movements, keyboard input, and beam send/receive. In addition, there is a general catch-all method, `unknownEvent()`. We will discuss these different types of events.

Handling pen movements

The event handlers that handle the movement of the stylus on the PDA display are , `penMove`, and `penUp`.

The `penDown()` method is invoked if the user places the pen on the display. It passes the X and Y coordinates of the point at which the stylus was placed on the display.

```
public void penDown( int x, int y )
```

The `penMove()` method is invoked if the user moves the pen over the display. The X and Y coordinates define the current position of the pen.

```
public void penMove( int x, int y )
```

The `penUp()` method is invoked if the user removes the pen from the display. It passes two parameters, the X and Y coordinates of the point from which the pen was removed.

```
public void penUp( int x, int y )
```

Handling keyboard input, beam send/receive, and unknown events

In the J2SE AWT the `java.awt.event.KeyListener` interface includes `keyPressed`,

`keyReleased`, and `keyTyped` methods for handling different keyboard events. Contrast this with KJava where only one such method exists, `keyDown()`.

The `keyDown` event is invoked if the user writes a character on the graffiti area, taps the calculator or menu icon, or presses any of the "hard keys" (by default, these are the Date Book, Address, page up, page down, To Do List, or Memo Pad keys). The `keyCode` parameter identifies the code of the key the user enters. If one of the hard keys is pressed, the `keyDown` event will match one of the corresponding constants defined in this class.

```
public void keyDown( int keyCode )
```

The `beamReceive()` method is used for receiving packets of data from other Palm devices via infrared. The data is received in a byte array, which is allocated by the virtual machine automatically.

```
public static boolean beamReceive( byte[] data )
```

The `beamSend()` method is not an event handler, but is obviously related to the `beamReceive()` method so we'll include it here. It is used for beaming data packets to another Palm device via infrared. You can call this method explicitly to beam data to another device, however, the other device must have registered a `beamReceive` handler in its current spotlet to receive data.

```
public static boolean beamSend( byte[] data )
```

The `unknownEvent` is a general catch-all event handling routine.

```
public void unknownEvent( int event, java.io.DataInput in )
```

Introducing the Scribble application

Now that we understand the basics of event handling, we're ready to move on to somewhat more advanced J2ME development. In this section, we will develop the Scribble application. We will not describe the code line by line but, instead, describe important lines, as well as the purpose of each method. You can access the complete code listing from Resources on page 35.

The Scribble application is a self-contained application that demonstrates the use of Sun's KJava API for the Palm OS. Scribble allows you to draw freehand pictures on the screen, adding regular text if desired.

Getting started with Scribble

As with the HelloWorld application, we need to import the classes to be used by the Scribble application. Following is the block of text that identifies the classes used by Scribble:

```
import com.sun.kjava.Bitmap;
import com.sun.kjava.Button;
import com.sun.kjava.Dialog;
```

```
import com.sun.kjava.Graphics;
import com.sun.kjava.HelpDisplay;
import com.sun.kjava.Spotlet;
```

The `Scribble` class extends the `Spotlet` class used in J2SE. It begins by defining the final static class variables.

```
public class Scribble extends Spotlet
```

The class variable `g` acts as a reference to the singleton `Graphics` object used throughout the class:

```
static Graphics g = Graphics.getGraphics ();
```

Defining methods and event handlers

Again, the `main()` method defines the main entry point for our application.

```
public static void main(String[] args)
```

The default constructor, `Scribble`, initializes the member variables, clears the screen, and draws the initial frame.

```
public Scribble()
```

The `paint()` method is responsible for updating or redrawing the display. It makes use of the class variable `g` -- a `Graphics` object similar to the one used in the Java 2 AWT.

```
private void paint()
```

The `penDown()` method implements the event handler to handle the event of putting the stylus, or pen, on the screen. It passes in the X and Y coordinates. In our Scribble application, this method tests to see if either the Clear or Exit button was pressed and if so, handles those events.

```
public void penDown(int x, int y)
```

The `keyDown()` method handles entering a graffiti letter on the graffiti writing area on your Palm device. The integer value, `keyCode`, passed into this method is the value of the character key entered. In the Scribble application we store the key pressed in the `lastKey` member variable then invoke the `paint()` method to update the screen.

```
public void keyDown(int keyCode)
```

The `penMove()` method handles the event of dragging the stylus across the screen. In the Scribble application, it is responsible for drawing with the stylus.

```
public void penMove(int x, int y)
```

The last method used, `clearDrawingArea()`, is the method invoked by the `penDown` event handler when the user taps the Clear button. It is a private method because it is intended only to be used internally within the `Scribble` class.

```
    private void clearDrawingArea()
```

Section 10. MIDP API

Introduction

The Mobile Information Device Profile (MIDP) is geared toward devices like cellular phones and pagers. MIDP, like KJava, also is built upon CLDC. The MID Profile provides a standard run-time environment that allows new applications and services to be deployed dynamically on end-user devices.

In this section, we will discuss in detail each of the seven packages defined by the MID Profile. We also will build a sample MIDP application.

UI design considerations

MIDP includes both a low-level UI API and a high-level UI API. The low-level API allows you complete access to a device's screen, as well as access to raw key and pointer events. However, with the low-level API, there are no user interface controls available. The application must explicitly draw buttons and all other controls.

Conversely, the high-level API provides simple user interface controls but no direct access to raw input events or to the screen. The controls are abstract due to the differences in screen sizes and input methods of MIDP devices. The MIDP implementation determines the way to draw the control, and it determines how to manage user input.

Let's take a closer look at the MIDP packages and classes.

MIDP API

MIDP encompasses the four core CLDC packages (`java.lang`, `java.io`, `java.util`, and `javax.microedition.io`), plus the following three MIDP-specific packages:

* `javax.microedition.lcdui`
* `javax.microedition.midlet`
* `javax.microedition.rms`

We will detail each of the MIDP-specific packages later in this section. In addition to the above new packages, MIDP also adds four new classes, shown below, to the core CLDC packages.

* `java.util.Timer` -- Used to schedule tasks for future execution in a background thread.
* `java.util.TimerTask` -- Used by the `java.util.Timer` class to define tasks for later execution in a background thread.
* `javax.microedition.io.HttpConnection` -- An interface that defines the necessary methods and constants for an HTTP connection.
* `java.lang.IllegalStateException` -- A `RuntimeException` that indicates that a method has been invoked at an illegal or inappropriate time.

Introducing MIDlets

A `MIDlet` is a Java class that extends the `javax.microedition.midlet.MIDlet` abstract class. It implements the `startApp()`, `pauseApp()`, and `destroyApp()` methods, which you can think of as being similar to J2SE's `start()`, `stop()`, and `destroy()` methods in the `java.applet.Applet` class.

In addition to the primary `MIDlet` class that extends `javax.microedition.midlet.MIDlet`, an MIDP application usually includes other classes, which can be packaged as jar files along with their resources -- this is known as a *MIDlet suite*. The various MIDlets in a MIDlet suite can share the resources of the jar file, although MIDlets from different suites cannot directly interact.

A MIDlet exists in one of three possible states during the application life cycle -- active, paused, or destroyed. *Active state*, as the name implies, means the MIDlet is running. This state begins when the `startApp` method is called. In a *paused state*, all resources the MIDlet is holding are released, but it is prepared to be activated again. The `notifyPaused` method invokes the paused state. In the *destroyed state*, a MIDlet has permanently shut itself down, releasing all resources, and is awaiting the garbage collector. It is invoked with the `notifyDestroyed` method.

In the next couple of panels, we'll look at a simple HelloWorld MIDlet.

The HelloWorld MIDlet

As with the KJava HelloWorld application, this MIDlet also displays, "Hello World!" on the screen of an MIDP device, as well as an Exit button, which terminates the application when pressed.

The HelloWorld.java file starts with the following lines of code to import the classes that will be used later in the `HelloWorld` class:

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
```

The `HelloWorld` class extends MIDlet since it is an MIDP application. It also implements `CommandListener` interface to handle events:

```
public class HelloWorld extends MIDlet implements CommandListener
```

The following method is the default constructor, which creates a new form, initializes the controls on it, and then displays it:

```
private Form form;

public HelloWorld()
{
    // Create a new form on which to display our text
   form = new Form("Test App");
```

```
        // Add the text "Hello World!" to the form
        form.append("Hello World!");

        // Add a command button labeled "Exit"
        form.addCommand( new Command( "Exit", Command.EXIT, 1 ) );

        // Register this object as a commandListener
        form.setCommandListener( this );
    }
```

The `startApp()` method is invoked to start the application much like an applet's start method. It may be called numerous times during a single execution of a MIDlet. If the MIDlet is paused, `pauseApp()` will be invoked. To restart the MIDlet, `startApp()` will be called. Main initialization code that needs to be performed only once should be placed in the constructor:

```
    public void startApp()
    {
        // Get a reference to the display, and show the form
        Display display = Display.getDisplay(this);
        display.setCurrent( form );
    }
```

`pauseApp()` is invoked to put the MIDlet into a paused state. In this application, we do nothing when entering the paused state; however, we must still implement the `pauseApp` method in our MIDlet because it is an abstract method in the parent MIDlet class.

```
    public void pauseApp() { }
```

`destroyApp()` is invoked to destroy the MIDlet and put it into a destroyed state. In this application, we release our reference to the form by setting it to `null`.

```
    public void destroyApp(boolean unconditional)
    {
        form = null;
    }
```

The `commandAction()` method is the event handler required to implement the `CommandListener` interface. Currently it destroys the application, and notifies the application management software that the MIDlet is complete.

```
    public void commandAction(Command c, Displayable d)
        {
            // Destroy this MIDlet
            destroyApp(true);

            // Notify the application management software that this MIDlet
            // has entered the destroyed state
            notifyDestroyed();
        }
```

MIDP packages

In addition to the standard CLDC packages, MIDP also includes three additional packages:

* `javax.microedition.lcdui` -- Defines classes that provide for control over the UI.

This package includes both the high-level UI classes (such as `Form`, `Command`, `DateField`, `TextField` and more), as well as the low-level UI classes (allowing low-level control over the UI).

* `javax.microedition.midlet` -- Contains one of the main MIDP classes, the MIDlet class, which provides MIDP applications access to information about the environment in which they are running.
* `javax.microedition.rms` -- Defines a set of classes that provide a mechanism for MIDlets to persistently store, and later retrieve, data.

Section 11. CDC API

Introduction

In this section, we describe the devices CDC targets and the requirements of those devices. You also will learn what packages and classes CDC supports.

Because this tutorial focuses on small, mobile devices, we will not go into the CDC API in depth as we have with CLDC. However, we will identify the J2SE packages and classes CDC uses, as well as the additional features CDC brings to the J2SE packages.

Devices CDC targets

CDC allows you to develop applications for consumer electronic and embedded devices, such as smartphones, two-way pagers, PDAs, home appliances, point-of-sale terminals, and car navigation systems. These devices run a 32-bit microprocessor and have more than 2 MB of memory, which is needed to store the C virtual machine and libraries.

CDC runs on top of the C virtual machine (CVM), and it is associated with the Foundation Profile. The Foundation Profile (FNDp) is a set of Java APIs intended for higher-end devices requiring a custom user interface (UI), typically provided by a device manufacturer.

CDC API overview

The CDC is an API built on top of the CLDC and is a more complete subset of the full J2SE API. It also includes an extra package -- the `javax.microedition.io` package -- containing all the same classes and interfaces defined in CLDC, and more.

Some of the more notable features that CDC includes over and above CLDC include:

* Floating point support (including `java.lang.Float`, `java.lang.Double`, and `java.lang.StrictMath` classes)
* A classloader class (`java.lang.ClassLoader`)
* Support for native processes (`java.lang.Process`)
* Advanced multithreaded support (including support for thread groups and more)
* Serialization classes (`java.io.Serialiable` and `java.io.Externalizable`)
* Reflection API (including `java.lang.reflect package`)
* File system support
* J2SE style network support (`java.net`)
* More complete support for the J2SE Collections API
* The addition of an `HttpConnection` interface to the `javax.microedition.io` package. This provides the necessary methods and constants for an HTTP connection.
* Support for the J2SE `java.lang.ref`, `java.math`, `java.security`, `java.security.cert`, `java.text`, `java.util.jar`, and `java.util.zip` packages.

Section 12. Summary

Wrapup

In this tutorial, we examined the background of J2ME and explored the J2ME configurations and profiles. We then took a look at setting up your development environment for developing J2ME applications.

We covered topics such as the K virtual machine (KVM) and the KJava profile used in conjunction with the Connected Limited Device Configuration (CDLC) API. We also discussed Mobile Information Device Profile (MIDP), which also uses CLDC. We also briefly discussed the Connected Device Configuration (CDC), which is used for larger applications.

Finally, you received hands-on experience by building a simple HelloWorld application that allowed you to see what you can do with J2ME. We used CLDC and KJava to develop Scribble, a basic drawing application, and also developed a small MIDP application.

While still in its infancy, Java for mobile devices already has changed the way people conduct their business and personal communications. As J2ME evolves and mobile devices become more sophisticated, so will the applications that support business and personal mobile communications.

Resources

Use these resources to follow up on this discussion of J2ME and expand your knowledge of wireless programming.

* The *KJava Sample applications* zip file includes complete source code for HelloWorld.java and Scribble.java, as well as supporting files.
* The *MIDP Sample application* zip file includes complete source code for HelloWorld.java, as well as supporting files for use with the J2ME Wireless Toolkit.
* For all things J2ME (including downloads for all the APIs and VMs used in this tutorial), visit the official *J2ME page* .
* Both *Sun* and *IBM* provide various versions of the Java 2 SDK.
* Download the *Palm OS Emulator (POSE)* .
* IBM *VisualAge Micro Edition* , which was recently named best embedded development tool by *JavaPro* magazine, provides everything you need to quickly create embedded Java applications for wireless devices.
* Interested in how J2ME has changed since 1999? Read Todd Sundsted's *"J2ME grows up"* (developerWorks, May 2001), which was inspired by a recent all-day tech session with Sun Java evangelist Bill Day.
* Visit our *Wireless Java Programming discussion forum* for help with your J2ME-related questions.
* Wireless programming extends well beyond the realm of the Java platform. For information, visit the *Wireless page* on developerWorks.

Your feedback

Please let us know whether this tutorial was helpful to you and how we could make it better.

Feel free to suggest improvements or other tutorial topics you'd like to see covered. Thanks!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.