CHAPTER 1

Getting Started

This book is all about creating mobile games for the Java platform. With that in mind, it is obviously important to have a working knowledge of the tools involved and to get them installed so that we can tackle the fun stuff later. So, I will start with introducing the tools in the first two sections to give you a general idea of what is involved. Then I will cover what you need to do to set up your computer for Java Micro Edition (Java ME) game development and how to get your games running on an actual target device. Once you have your development environment running, you can start by building and modifying the examples from this book. You can download all the source code for the examples from the Source Code/Download section of the Apress web site (`http://www.apress.com`). This includes all the image files, descriptor files, and other resource files.

Understanding Java ME

The Java Micro Edition (Java ME) is the version of the Java platform that's designed for use with small devices such as cell phones, personal digital assistants (PDAs), TV set-top boxes (for web browsing and e-mail without a whole computer), and embedded devices. Since these devices vary quite a bit in their capabilities, the Java ME platform has two different *configurations*, each with its own choice of *profiles*. The Connected Limited Device Configuration (CLDC) is the configuration you'll be working with in this book. It's designed for cell phones and low-level PDAs. More precisely, CLDC is intended for devices with a 16-bit or 32-bit processor, with at least 160KB of nonvolatile memory, at least 32KB of volatile memory, and some network connectivity, possibly wireless and intermittent. CLDC's unique profile is the Mobile Information Device Profile (MIDP). This book covers MIDP versions 1, 2, and 3, although the focus will be on MIDP 2 since that is the current industry standard for cell phones and will continue to be the standard for some time. The other configuration possibility associated with Java ME is the Connected Device Configuration (CDC), which isn't specifically covered in this book. Starting with MIDP 3, some CDC devices will be able to run MIDP applications; however, from the MIDP application's perspective, the configuration must behave as if it were CLDC, and therefore, no CDC-specific programming is covered here.

The configuration specifies the type of Java Virtual Machine (JVM) that's used and what will be included in the minimal class libraries (the `java.*` packages and the `javax.microedition.io` package in the case of CLDC). CDC specifies a complete JVM, but the JVM of CLDC has some limitations compared to the standard JVM. A profile is added on top of the configuration to define a standard set of libraries (the other `javax.microedition.*` packages in this case). MIDP contains packages for application lifecycle, a user interface, media control, input/output, data storage, and security.

In addition to its configuration and profile, a Java ME–capable device may also have a set of optional application programming interfaces (APIs) available. Some are proprietary, but many are standard. Standard optional APIs are defined in Java Specification Requests (JSRs), which are submitted, reviewed, and published through the Java Community Process (JCP). You can download the precise specifications for any JSR (free) from the JCP web site (`http://jcp.org/`). CLDC, CDC, and MIDP are all defined as JSRs through the Java Community Process.

How the CLDC Differs from the Rest of the Java Universe

If you're coming to Java ME from a background of programming for Java Standard Edition (Java SE) or Java Enterprise Edition (Java EE), things should look pretty familiar, but just a little bit different. It's like Java in a parallel universe where everything has been streamlined to be as small and efficient as possible. You have fewer tools at your disposal in terms of built-in libraries, and you need to place greater priority on writing tight, efficient code than a Java SE or Java EE developer would. Those are already good reasons to switch to Java ME if you're one of those people who loves a good challenge. Another motivation for making the switch to Micro Edition is that the applications are so small that you typically get to design most or all of the program yourself instead of being a cog writing a part of an obscure module.

The designers of the CLDC specification have made an effort to make CLDC resemble the standard platform as closely as possible, and they've done a pretty good job of it. Nothing critical to small applications appears to be missing. I'll give a general outline of the changes here, and I'll refer you to later chapters in this book for a more in-depth discussion of the aspects that have changed the most dramatically.

Differences in the JVM

The JVM specified in CLDC is mostly the same as the standard JVM. Unsurprisingly, a few of the costlier noncritical features have been eliminated. One example is the method `Object.finalize()`. According to the JavaDoc, the `Object.finalize()` method is called on an object when the JVM determines that it's time to garbage-collect that object. The actions the object can take in its `finalize()` method aren't restricted, so in particular it can make itself available again to currently active threads! The garbage collection algorithm is already expensive, and this method clearly undermines its efficiency by obligating the JVM to recheck objects that had already been marked as garbage. It's no wonder this method was eliminated in CLDC, since it's not hard at all to keep track of the objects that you're still using without requiring the JVM to check with you before throwing anything away.

Some of the other areas where the JVM's set of features have been reduced are in security, threads, and exceptions/errors. See the "Understanding Protection Domains and Permissions" section in Chapter 8 for a discussion of the differences in the security model. See the "Differences Between CLDC Threads and Threads in Standard Java" section in Chapter 4 for information about threads. The changes in the error-handling system are that CLDC doesn't allow asynchronous exceptions and that the set of error classes has been greatly reduced. Instead of 22 possible errors, you now have only `OutOfMemoryError`, `VirtualMachineError`, and `Error`. On the other hand, almost all the exceptions in the `java.lang.*` package have been retained.

You may not notice a few changes to the JVM just by looking at the API. In CLDC, the JVM is allowed to perform some optimizations (such as prelinking classes) that were disallowed to the standard JVM. Such changes shouldn't concern the application programmer in general. The one exception is that an additional preverification stage has been added after compilation. The preverification process adds extra information to the class file to make the bytecode verification algorithm easier at runtime when the device checks that your class file is valid before using it. You easily accomplish the preverification step with standard tools (see the "Using KToolbar" section and the "Building with Ant" sidebar later in this chapter). Preverification isn't technically required 100 percent of the time, but it aids in compatibility, and there's no reason not to do it.

One more general item to be aware of is that although a CLDC-compliant platform is required to support Unicode characters, it's required to support only the International Organization for Standardization (ISO) Latin 1 range of characters from the Unicode standard, version 3.0. For more information about character encoding in Java, see `http://java.sun.com/javase/6/docs/technotes/guides/intl/encoding.doc.html`. Also, the initial version of CLDC (1.0) did not include floating-point numbers as simple data types (`double` and `float`). These were added in the newer version, CLDC 1.1.

Differences in the Libraries

As you may guess, the standard libraries have been drastically reduced. It's unfortunate in many cases, but—as I mentioned earlier—doing without some helpful tools is one of your challenges as a Java ME developer. The only `java.*` packages that you have available to you are `java.lang.*`, `java.util.*`, and `java.io.*`. That means you have to do without `java.lang.reflect.*`, `java.math.*`, `java.security.*`, and many others. Many of the missing packages have been replaced by MIDP packages that are more appropriate for small devices, as you'll see throughout this book.

Although the three remaining `java.*` packages have been greatly reduced, it's clear that the designers of CLDC have tried to keep as much as possible and create familiar replacements for classes and methods that had to be removed. The `java.lang.*` package has been pared down to just the classes that correspond to data types (`Integer`, `Boolean`, and so on) and a few necessary items: `Math`, `Object`, `Runnable`, `String`, `StringBuffer`, `System`, `Thread`, and `Throwable` (plus the exceptions and errors discussed previously). The `java.util.*` and `java.io.*` packages have been similarly reduced to their essentials. For examples of how to use the MIDP versions of the `java.io.*` classes, see the "Serializing More Complex Data Using Streams" section in Chapter 5. For a discussion of the changes to the `java.util.*` package, see the "Using the java.util Package" section in Chapter 2.

The Three Versions of MIDP

Unlike products where a new version completely replaces an earlier version, different versions of MIDP coexist on the market and likely will continue to do so for some time. It's not just a question of the new version being better or more innovative than an earlier version—in the case of MIDP there's also a question of earlier versions being more appropriate for less-powerful devices. The handset market seems to be constantly evolving toward more power and more features becoming cheaper, and the portion of the market using more advanced versions expands accordingly. On the other hand, there's a market for a range of models, and manufacturers will continue to make cheap, low-end models as long as they keep selling. Additionally, while people tend to upgrade their cell phones pretty frequently on average, there are still a lot of old handsets out there in the wild—and they're attached to potential customers for your game—so you probably don't want to just dismiss them out of hand.

MIDP 2 is a good level to write games for because it provides big advantages over MIDP 1 for game programming. Plus, if you shoot for MIDP 2 as a target when writing your game, there exist tools such as J2ME Polish to back-port it to MIDP 1 devices (see the sidebar "Using J2ME Polish" in Chapter 10). There's less advantage in specifically targeting MIDP 3 handsets (by writing games that won't run on MIDP 2 handsets) because MIDP 3 doesn't offer all that much over MIDP 2 in terms of tools for games. The biggest difference between MIDP 3 and MIDP 2 is found in terms of how MIDP applications interact with each other. In MIDP 3, `MIDlets` can share libraries and communicate with each other while running concurrently (see the "More Options" section of Chapter 7); plus, the `Permission` class has been changed to be compatible with CDC and the rest of the larger Java universe (see Chapter 8).

It's possible to write some fun basic games using MIDP 1. I wrote the example game in Chapter 2 to be compatible with MIDP 1 so you can see what's available there. It's easy to underestimate the value of simple games, but keep in mind that not only are they cheaper to produce, sometimes they actually sell better than more complicated games since many users just want a familiar diversion when playing a game on a cell phone instead of something cool yet involved, requiring time and attention to learn. But MIDP 2 is loaded with additional features that allow you to create a much richer gaming environment, and the added advantages—not only in terms of making a more exciting game but also in terms of making a simple game look more beautiful, exciting, and professional—shouldn't be underestimated either. The most obvious additional tool you get in MIDP 2 is the package `javax.microedition.lcdui.game`, which is a special package to help you optimize game graphics and controls (see Chapter 3). Also, a version of the Mobile Media API (JSR 135) is a required part of MIDP 2, so you can add music to your game (see the "Using Media" section in Chapter 4).

Much of the difference from one target device to the next is more a question of which optional APIs are supported than of which version of MIDP is being used. The optional APIs can open up new channels of communication (see the sections "Using SMS" in Chapter 6 and "Using Bluetooth" in Chapter 7). And one of the most enticing APIs for a game developer to try out and play with is the (optional but widely supported) Mobile 3D Graphics API, which is defined in JSR 184 (see Chapter 9).

Overall, dealing with the huge range of devices with different capabilities (in terms of both hardware and software) is one of the two big challenges of Micro Edition programming, right up there with the challenge of optimizing your game for small, limited devices. But even though Java's "write once, run anywhere" philosophy works less than perfectly here, Java ME goes a long way toward making it easy for you to write for a wide range of target devices at once. It's a great common platform for keeping up with the hot and fast-paced world of mobile!

Downloading and Installing the Toolkit

If you haven't already downloaded and installed a development toolkit, you can get the Sun Java Wireless Toolkit for CLDC at `http://java.sun.com/javame/downloads/index.jsp`. I downloaded the WTK version 2.2 for this book, but if you've selected a more recent release, it won't be very different. Some handset manufacturers offer specialized Java ME emulators for free download, but they're often based on Sun's Reference Implementation (RI), so for the rest of this chapter I'll assume you're using the Java ME Wireless Toolkit from Sun.

If you have trouble downloading the toolkit from Sun, keep in mind that you need to register at the Sun site and log in. This shouldn't be a problem—it doesn't cost anything. You have to submit your e-mail address, but Sun has never sent me any spam as a result of my registration, so don't worry about anything.

The Java ME Wireless Toolkit contains a minimal `MIDlet` development environment (called *KToolbar*), a cell phone emulator, and a number of helpful demo applications with source code. It also contains a clear and comprehensive manual in both Hypertext Markup Language (HTML) format and PDF format (the "User Guide" in the WTK `docs` folder). You should definitely take the time to familiarize yourself with it so that you can find more details on what's available once you've got the basics down.

Building an Application for MIDP

I'll stick with tradition and start with the classic "Hello, World" application. This example will illustrate how to get a minimal `MIDlet` compiled and running.

When you examine the demo applications that are bundled with the toolkit, you'll notice that they consist of a JAR (Java archive) file and a JAD (Java application descriptor) file. The JAR file contains the class files, the resources, and a manifest file (just as you'd expect to find in any JAR file). The JAD file is a Java properties (text) file that contains information to help the device install and run the application. The manifest file (`MANIFEST.MF`) found inside the JAR file contains a lot of the same information as the JAD file, and properties that appear in both must have identical values; otherwise (for security reasons), the application can't be installed on any MIDP device, including the WTK emulator. The JAD file and the manifest file contain data that the platform needs to run the application, such as the CLDC version that the application requires. The reason for having two separate files for these same properties is that one goes inside the JAR file and one is outside. So security-critical properties go in the manifest file inside the JAR (where they can be protected from corruption by digitally signing the JAR, as you'll learn in the "Using Digital Certificates" section of Chapter 8), and properties that are needed before the application is installed—to aid in the installation process—go in the JAD file. Notably, the JAD file contains the two installation-specific properties `MIDlet-Jar-URL` and `MIDlet-Jar-Size`.

Listing 1-1 is an example of the JAD file I wrote for my "Hello, World" application. I called this file `hello.jar`.

Listing 1-1. *hello.jar*

```
MIDlet-1: Hello World, /images/hello.png, net.frog_parrot.hello.Hello
MMIDlet-Description: Hello World for MIDP
MIDlet-Jar-URL: hello.jar
MIDlet-Name: Hello World
MIDlet-Permissions:
MIDlet-Vendor: frog-parrot.net
MIDlet-Version: 2.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-2.0
MIDlet-Jar-Size: 3201
```

The `MIDlet-1` (and `MIDlet-2`, and so on) property gives the name of the `MIDlet`, the location of the `MIDlet`'s icon, and the fully qualified name of the `MIDlet` class to run (see the section on "Understanding the MIDlet Lifecycle" in Chapter 2). The first two items describe how the `MIDlet` will appear on the menu of `MIDlets`. The icon should be in the JAR file, and its location should be given in the same format as is used by the method `Class.getResource()`. Thus, in this example, your JAR file should contain a top-level folder called `images`, which contains an icon called `hello.png`, as shown in Figure 1-1.

Figure 1-1. *This is the icon hello.png.*

■ **Note** The `MIDlet-Jar-Size` property gives the size of the corresponding JAR file in bytes, which you can find by looking at the properties or long listing of the JAR file. Be aware that if you rebuild the demos or your own applications using the build script or batch file bundled with the toolkit, you must manually update the size of the JAR file in the JAD file. If the `MIDlet-Jar-Size` property in the JAD file doesn't match the size of the JAR file, the `MIDlet` won't run. The simplest way to deal with this is just to use the KToolbar application (see the section on "Using KToolbar" later in this chapter). For a more complex project that may require customized build steps, the standard solution is to use Ant (see the "Building with Ant" sidebar later in this chapter).

The `MIDlet-Jar-URL` property gives the address of the `MIDlet jar` as a uniform resource locator (URL). According to the MIDP specifications, this can be a relative path (relative to the location of the JAD file, so if the JAR file and the JAD file are kept in the same directory, this is just the name of the JAR file); however, there are some handsets on the market that require an absolute (complete) URL. The "Requesting Permissions" section of Chapter 8 discusses the `MIDlet-Permissions` property, but for simple games, you can omit it or leave it blank. The other properties are self-explanatory.

Creating the "Hello, World" Application

This section shows the "Hello, World" application. The `MIDlet` will display the message *Hello World!* on the screen and remove it (or later put it back) when you click the Toggle Msg button. Clicking the Exit button will terminate the `MIDlet`. The application consists of two classes: the `MIDlet` subclass called `Hello` and the `Canvas` subclass called `HelloCanvas`. How it works is very simple. The `Hello` class implements a list of standard methods to receive information from the device. This includes the `MIDlet` lifecycle methods that the platform calls in order to start and stop the application (see the section on "Understanding the MIDlet Lifecycle" in Chapter 2 for more details) as well as the implementation of the `CommandListener` interface that the platform calls if the user presses a key while the application is running. The `HelloCanvas` class repaints the screen with or without the *Hello World!* message when the `toggleHello()` method is called. Listing 1-2 shows the code for `Hello.java`.

Listing 1-2. *Hello.java*

```
package net.frog_parrot.hello;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

/**
 * This is the main class of the "Hello, World" demo.
 *
 * @author Carol Hamer
 */
public class Hello extends MIDlet implements CommandListener {

  /**
   * The canvas is the region of the screen that has been allotted
   * to the game.
   */
  HelloCanvas myCanvas;

  /**
   * The Command objects appear as buttons in this example.
   */
  private Command exitCommand = new Command("Exit", Command.EXIT, 99);

  /**
   * The Command objects appear as buttons in this example.
   */
  private Command toggleCommand = new Command("Toggle Msg", Command.SCREEN, 1);

  /**
   * Initialize the canvas and the commands.
   */
  public Hello() {
    myCanvas = new HelloCanvas();
    myCanvas.addCommand(exitCommand);
    myCanvas.addCommand(toggleCommand);
    // you set one command listener to listen to all
    // of the commands on the canvas:
    myCanvas.setCommandListener(this);
  }

  //------------------------------------------------------------------
  //  implementation of MIDlet

  /**
   * Start the application.
   */
  public void startApp() throws MIDletStateChangeException {
    // display my canvas on the screen:
    Display.getDisplay(this).setCurrent(myCanvas);
    myCanvas.repaint();
  }

  /**
   * If the MIDlet was using resources, it should release
   * them in this method.
   */
  public void destroyApp(boolean unconditional)
      throws MIDletStateChangeException {
  }

  /**
   * This method is called to notify the MIDlet to enter a paused
   * state. The MIDlet should use this opportunity to release
   * shared resources.
   */
  public void pauseApp() {
  }

  //-----------------------------------------------------------------
  //  implementation of CommandListener

  /*
   * Respond to a command issued on the Canvas.
   * (either reset or exit).
   */
  public void commandAction(Command c, Displayable s) {
    if(c == toggleCommand) {
      myCanvas.toggleHello();
    } else if(c == exitCommand) {
      try {
        destroyApp(false);
        notifyDestroyed();
      } catch (MIDletStateChangeException ex) {
      }
    }
  }

}
```

Listing 1-3 shows the code for `HelloCanvas.java`.

Listing 1-3. *HelloCanvas.java*

```
package net.frog_parrot.hello;

import javax.microedition.lcdui.*;

/**
 * This class represents the region of the screen that has been allotted
 * to the game.
 *
 * @author Carol Hamer
 */
public class HelloCanvas extends Canvas {

  //---------------------------------------------------------
  //   fields

  /**
   * whether the screen should currently display the
   * "Hello World" message.
   */
  boolean mySayHello = true;

  //-----------------------------------------------------
  //    initialization and game state changes

  /**
   * toggle the hello message.
   */
  void toggleHello() {
    mySayHello = !mySayHello;
    repaint();
  }

  //-------------------------------------------------------
  //  graphics methods

  /**
   * clear the screen and display the "Hello World" message if appropriate.
   */
  public void paint(Graphics g) {
    // get the dimensions of the screen:
    int width = getWidth ();
    int height = getHeight();
    // clear the screen (paint it white):
    g.setColor(0xffffff);
    // The first two args give the coordinates of the top
    // left corner of the rectangle.  (0,0) corresponds
    // to the top-left corner of the screen.
    g.fillRect(0, 0, width, height);
    // display the "Hello World" message if appropriate.
    if(mySayHello) {
      Font font = g.getFont();
      int fontHeight = font.getHeight();
      int fontWidth = font.stringWidth("Hello World!");
      // set the text color to red:
      g.setColor(255, 0, 0);
      g.setFont(font);
      // write the string in the center of the screen
      g.drawString("Hello World!", (width - fontWidth)/2,
                   (height - fontHeight)/2,
                   g.TOP|g.LEFT);
    }
  }

}
```

The “Hello, World” application is simple enough to run with MIDP 1 as well as with MIDP 2. Figure 1-2 shows what the “Hello, World” application looks like when running on an early version of the Wireless Toolkit (version 1.0.4) using the `DefaultGrayPhone` emulator. It is possible to use the WTK 2.2 and later to see how your program works on a MIDP 1 device by changing the target platform in the project settings, as you’ll see in the next section.

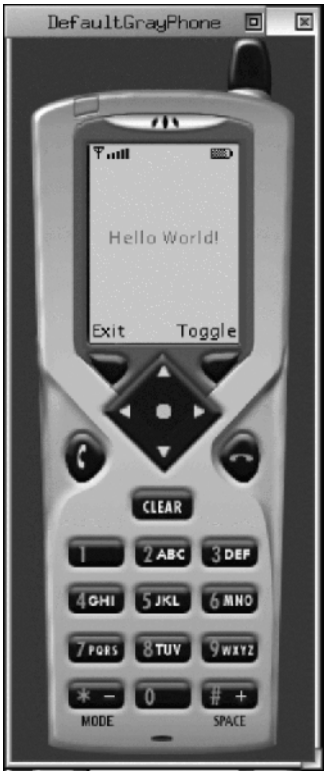
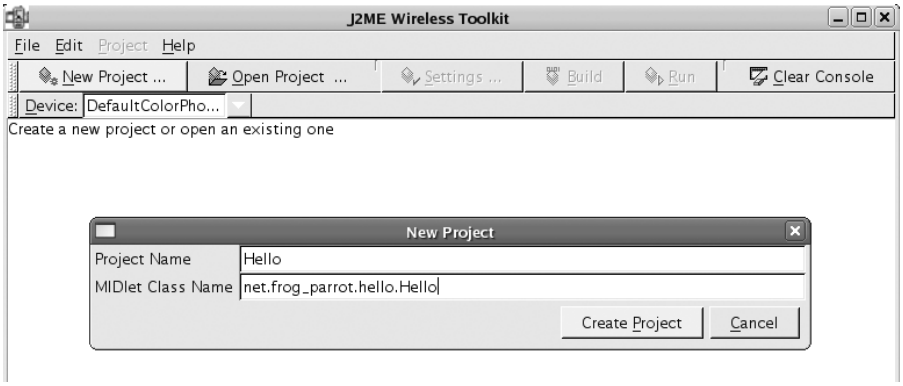


Figure 1-2. *The “Hello, World” application*

Using KToolbar

The `MIDlet` development environment KToolbar is easy to use, and it’s well documented. KToolbar is a “minimal” development environment in the sense that, unlike complete integrated development environments (IDEs) such as JBuilder, it doesn’t contain a text editor. But you probably already have a text editor, so if you don’t mind using an ordinary text editor to write your source code, KToolbar will take care of the MIDP-specific tasks.

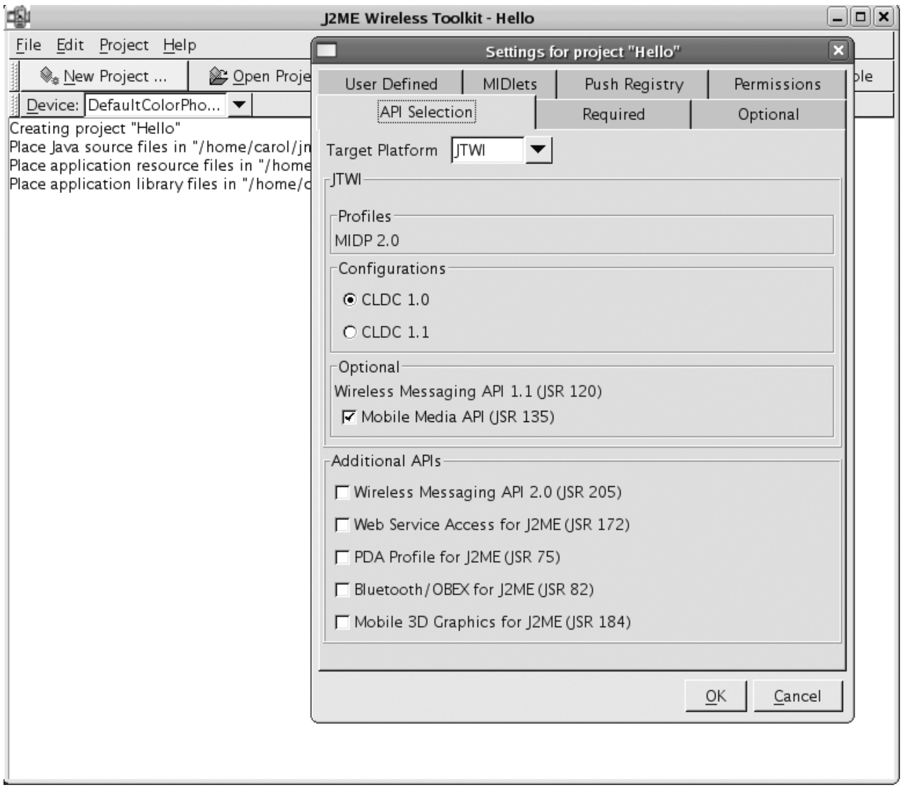
The first thing to do after launching KToolbar is click the New Project button just below the menu bar. This will automatically open some GUI windows that prompt you to fill in information that KToolbar will use to construct the JAD and manifest files as well as the directories for the project (see Figures 1-3 and 1-4).


J2ME Wireless Toolkit
File Edit Project Help
New Project ... Open Project ... Settings ... Build Run Clear Console
Device: DefaultColorPho...
New Project
Create Project
Cancel

Create a new project or open an existing one

Project Name: Hello
MIDlet Class Name: net.frog_parrot.hello.Hello

Figure 1-3. *Creating a new project in KToolbar*


J2ME Wireless Toolkit - Hello
File Edit Project Help
New Project ... Open Proje
Device: DefaultColorPho...
Settings for project "Hello"
User Defined MIDlets Push Registry Permissions
API Selection Required Optional
OK Cancel

Creating project "Hello"
Place Java source files in "/home/carol/jn
Place application resource files in "/home
Place application library files in "/home/c

Target Platform: JTWI

JTWI

Profiles
MIDP 2.0

Configurations
(● CLDC 1.0) ○ CLDC 1.1

Optional
Wireless Messaging API 1.1 (JSR 120)
[✓] Mobile Media API (JSR 135)

Additional APIs
☐ Wireless Messaging API 2.0 (JSR 205)
☐ Web Service Access for J2ME (JSR 172)
☐ PDA Profile for J2ME (JSR 75)
☐ Bluetooth/OBEX for J2ME (JSR 82)
☐ Mobile 3D Graphics for J2ME (JSR 184)

Figure 1-4. *Setting up a new project in KToolbar*

Once you've filled in the project settings, KToolbar creates a set of directories for the project under the WTK's apps folder. It creates a `src` directory (where you place your source code), a `res` directory (where you place resource files such as images that should be built into the JAR), and a `bin` directory (where KToolbar places the finished JAR and JAD files it constructs for your project). Note that unlike some IDEs such as JBuilder, it doesn't create its own project file describing your project, so you can open a "project" in KToolbar, even if you created the directory tree for the project yourself, instead of having KToolbar create it for you. All you need to do is make sure your project's root directory is in the right place (in the `apps` folder inside the `WTK2.2` folder), and you can open it as a project in KToolbar.

In addition to setting up your project, KToolbar will preverify and build your project and then run it in the emulator at the click of a GUI button. It also has a whole suite of useful features for debugging and performance monitoring. Plus, it has tools for digitally signing your `MIDlet` (see the "Using Digital Certificates" section of Chapter 8). If you place a third-party obfuscation tool JAR in the WTK's `bin` directory, KToolbar will also obfuscate the class files automatically during the build. If you click KToolbar's Build and Run buttons, the current project will be run from the project's directory without building it into a JAR file. However, if you select Project ➤ Package ➤ Create Package, KToolbar will create a JAR and JAD pair that can be run on an actual device. By selecting Project ➤ Run via OTA, you can install the `MIDlet` on the emulator (via local HTTP) exactly the same way you download and install the `MIDlet` onto a handset.

The WTK offers a choice of different default emulators to test your `MIDlet` on, such as the `DefaultColorPhone`, `DefaultGrayPhone`, `MediaControlSkin`, and `QwertyDevice`. KToolbar has a drop-down menu that allows you to choose which device to use. All of these devices are defined in subdirectories of the WTK's `wtklib/devices` directory. If you look there, you'll see that each emulator skin has a corresponding properties file, which you can modify if you'd like to change something, such as the emulator's screen size. You can also add new directories there (such as skins offered by manufacturers to emulate their devices), and the new emulators will automatically show up as options in KToolbar. Through KToolbar you can also set parameters such as network speed, Virtual Machine (VM) speed, and heap size in order to better emulate the behavior on specific devices, but in practice it's not terribly effective at reproducing the behavior on specific devices. It's not a real substitute for testing on a number of devices both for optimization and for platform-specific bugs.

BUILDING WITH ANT

For a basic `MIDlet`, the Wireless Toolkit's KToolbar application is pretty convenient, but as soon as you want to do anything fancy with your build at all (such as add extra steps for style checking, preprocessing, custom resource building, etc.) it's a good idea to set up your build process to use Ant.

The nice thing about Ant is that it's easy to use and it's easy to extend it to do essentially anything you want. The build process is controlled through XML files, which group sets of commands as "targets." The commands themselves (Ant tasks) can be calls to Java classes, so if there's no existing command to perform some task that you'd like to have in your build process, you can write your own Ant task to do it.

An MIDP project has some standard steps (compile, preverify, build the JAD file, write the corresponding JAR), and as you might imagine, other people have already written Ant tasks for all of this. So after installing Ant, the next step for an MIDP project is to download and install a JAR of Java ME–specific build tasks such as those provided by Antenna (`http://antenna.sourceforge.net/`).

All you really need to do to install Antenna is download the JAR file and drop it into your Ant `lib` directory. Then the Antenna Home overview page gives you all the details about what Ant tasks are available and describes the corresponding properties and attributes you need to set. If you download the source code as well, it comes with some sample demo build files that are designed to work with example programs in the WTK to get you started.

Running Your Game on an Actual Cell Phone

The emulator is a helpful development tool, but even though it works well, it's no substitute for testing your game on an actual device. Plus, playing your game on your own cell phone is the fun part! The idea of how to load the file onto the phone is pretty simple, but you need to be aware of a few details.

You have two ways to proceed. The first is to transfer files from your PC using a serial/Universal Serial Bus (USB) cable or an infrared connection or Bluetooth. This option doesn't require the data to even leave your house. The second option is more exciting. It consists of placing the required files on a server on the Internet and downloading them using a data connection from the phone—using Global System for Mobile Communications/General Packet Radio Service (GSM/GPRS).

Both methods have advantages. The first method doesn't require you to make a call from the phone and is therefore completely free (except for the cost of the cable if one isn't included with the phone). Also, since the transfer takes place entirely within your own network (generally behind your firewall), there's no danger of your game being downloaded by unauthorized users. But this technique can place additional requirements on your local system. For example, in the case of Nokia, the software that's used to perform the transfer works only on Windows, and the PC that the data is being loaded from must have an infrared port or a separate USB cable.

The second method, placing the games on a Wireless Application Protocol (WAP)–accessible web page, is clearly preferable if you intend to distribute the game yourself—even if you're distributing it only to your friends. After all, if you set it up so that you can download your game off the Internet, you can tell other people where it is and they can download it as well. For this option you need to be sure that your phone service contract includes WAP access and application downloading. (This is a typical option that's offered with a Java-enabled phone, so the salesperson who sells you the phone will probably suggest it to you before you even have to ask about it.) You'll also need a server on which to place the files. This shouldn't be too difficult to come by since most Internet service providers (ISPs) offer some personal web space with standard Internet access contracts. All additional software needed for this means of data transfer exists in free versions for all platforms.

In this book I'll cover only how to transfer your games to the device through the Internet and not through direct file transfer because transferring the files directly is vendor dependent. If you'd like to transfer the files to your phone directly, the first step is to go to the web site of the phone's manufacturer. In the case of Nokia, for example, the necessary software is easy to find on the site and is well documented. The same should be true of most other makers of CLDC devices.

Using WAP

WAP is the protocol that small devices use to access the Internet. The principle of WAP is that your cell-phone provider makes available a gateway through which your phone can access the Internet. Since small screens make standard browser functions and standard HTML pages unusable on low-end devices, there's another markup language specially designed for cell phones and other small devices called Wireless Markup Language (WML). If your phone contract specifies WAP access and doesn't restrict browsing to some specific portal and sites, you can direct your phone to a WML page listing your `MIDlets`. From there you can download them. This is similar to a standard HTML web page embedding a Java applet. Many MIDP handsets are capable of loading simple HTML pages, but since the principle is the same, I'll give a WML page as an example. For an example of a corresponding HTML page for `MIDlet` download, look no further than your project's `bin` directory, where KToolbar generates an HTML page to go with the JAR and JAD it builds.

To prepare your games for download, you first need to upload them onto a web server. You need to place the WML file on the server as well as the JAR and JAD files. In addition, you may have to perform some configuration so that the web server, when accessed, returns a correct Multipurpose Internet Mail Extensions (MIME) type description for those files. Otherwise, the phone may be unable to recognize them. The following sections explain these steps in detail using the Nokia 6100 as the example phone.

Preparing the WML File

You can use WML to display interesting content by itself, offering user interface (UI) elements such as forms, buttons, and so on. But you don't need to do anything fancy to make a page from which your game can be downloaded. In fact, it's better to resist the temptation to make a complex WML page because of the screen limitations of the target device. Try to keep it small and simple. Listing 1-4 shows a minimal example suitable for a download page. The file is called `hello.wml`.

Listing 1-4. *hello.wml*

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
 "http://www.WAPforum.org/DTD/wml_1.1.xml">
<wml>
<card id="hello">
<p>
Hello world!
</p>
<p>
<a href="hello.jad">Hello World App</a>
</p>
</card>
</wml>
```

Here's how it works: The first two lines are mandatory to identify the file as WML. The content of the page must be enclosed between the `<wml>` opening tag and the `</wml>` closing tag. The `<card>` tag delimits one screen of data for the device (not much!). As you may guess if you know some HTML or Extensible Markup Language (XML), this page will display one line containing the text *Hello world!* and another line with a link with the text *Hello World App*, as shown in Figure 1-5.

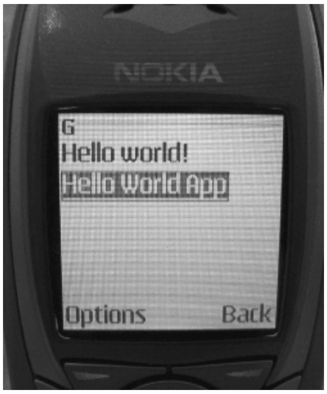


Figure 1-5. *The hello.wml WAP page running on the Nokia 6100*

For this link to work, the server directory containing the file `hello.wml` must contain a JAR file called `hello.jar` and a JAD file called `hello.jad`.

The page can contain multiple game links, and you add them in the obvious way. For example, inside the enclosing `<card>` tags, you could add a second triple such as the following:

```
<p>
<a href="maze.jad">Amazing Maze!</a>
</p>
```

on the lines immediately following these lines:

```
<p>
<a href="hello.jad">Hello World App</a>
</p>
```

(Obviously, you must also upload the corresponding JAR and JAD files to the server for this link.) But if you have a large number of downloadable JAR files, you'll probably want to arrange them on a series of separate pages. If you have multiple versions of the same game suite that are optimized for different devices, it's a good idea to make a separate WML page for each device rather than making a page for each game suite and having the page contain the versions for multiple devices. The standard technique for a professional game is to configure the server to return the page with links to the appropriate game versions based on the "user-agent" header sent by the handset when requesting the page (see the "Identifying the Platform" sidebar in Chapter 10).

One word of warning: The `<p></p>` tags enclosing the links in the previous files aren't optional. In the case of the Nokia 6100, the device failed to recognize the links without the `<p></p>` tags.

Configuring the Server

MIDP specifies a particular protocol for downloading and installing `MIDlets` called Over the Air (OTA) provisioning. The protocol requires that the handset first download the JAD file (and do some checking to see if the `MIDlet` can be installed and if it's replacing another `MIDlet`) and then download the JAR file and—after some security checks to make sure it conforms to the JAD description—install the `MIDlet`. Fortunately, the protocol was designed in such a way that any ordinary HTTP server can be used without any special configuration except to be sure that the appropriate file types are recognized. Many web servers aren't configured by default to recognize the file types associated with WAP/WML/Java ME by the file extensions. If you're lucky, just uploading the files (described in the previous section) to a directory in the public area of the server will be sufficient. If you're not, the cell phone will complain that the files are in an unrecognized format (even though the JAD file is just text . . .). If you run into this problem, you'll need to do a little bit of server configuration. I'll explain what to do in the case of the popular Apache server.

If the Apache server is running on your own machine—which is a convenient option if you have cable or Asymmetric Digital Subscriber Line (ADSL)—all you need to do is update the `httpd.conf` file. (On Red Hat 9, this file is located in `/etc/httpd/conf/httpd.conf`.) Just add the following lines to the file:

```
#### WAP/WML/JAD
##
AddType text/vnd.wap.wml wml
AddType text/vnd.wap.wmlscript wmls
AddType application/vnd.wap.wmlc wmlc
AddType application/vnd.wap.wmlscriptc wmlsc
AddType image/vnd.wap.wbmp wbmp
AddType text/vnd.sun.j2me.app-descriptor jad
AddType application/java jar
####
```

Then you must restart the server to make this information available. On Red Hat 9 you can restart the server by typing the following command as root:

```
# service httpd restart
```

If the web server you're using belongs to your ISP, first check if the server is already configured to recognize the required types. (To check, create the page and then try to access it with your cell phone as described in the following section.) If you're using web space made available by the cell-phone provider, then there's a high probability that the ISP has already taken care of the proper configuration.

If the server hasn't been configured correctly, you may be able to fix it yourself. Here's what to do if your ISP uses an Apache server (other servers may have similar tricks that you can find by consulting the documentation or Google): you won't be allowed to change the main configuration file, but you can inform Apache of the correct MIME types by placing the same lines as previously in a file called `.htaccess` somewhere in your web space area. (Note that you must put such a file in every directory in which you place WAP/WML/J2ME files.) Here's the file `.htaccess`:

```
#### WAP/WML/JAD
##
AddType text/vnd.wap.wml wml
AddType text/vnd.wap.wmlscript wmls
AddType application/vnd.wap.wmlc wmlc
AddType application/vnd.wap.wmlscriptc wmlsc
AddType image/vnd.wap.wbmp wbmp
AddType text/vnd.sun.j2me.app-descriptor jad
AddType application/java jar
####
```

Of course, you can't ask the ISP to restart its server, but it isn't necessary because Apache will notice the new file automatically. It's possible that the main configuration (which you don't control) forbids this type of user-directed overriding. In that case, you'll have to ask the ISP to change its policy or use another ISP.

Accessing the WML File and Downloading Applications

Now for the fun part: downloading the games onto the phone! Recall that in this section I'll use the Nokia 6100 as an example. This is a typical MIDP 1 CLDC-enabled phone, but it works the same as later model devices.

Remember, you need WAP access as mentioned previously. Be sure to check the costs involved in your contract with WAP connections. For friends who simply download your games and keep them in their phones, this is unlikely to be expensive since `MIDlets` are quite small and the time required for downloading them will rarely exceed a single minute. For the developer, however (that's you!), it's best to get a contract that has a fixed price with unlimited WAP usage since you'll certainly have to perform this operation a number of times (unless you're placing your games on your phone using a direct PC connection during the development phase).

Before you start, you must verify that the WAP access is configured on the phone. This should have been done when you got the phone, or you should have documentation from your phone service and WAP provider giving the details. In my case, I configured the phone by going to a particular web site and giving the phone number and a PIN code. The server then sent a Short Message Service (SMS) message containing all required information to the phone, which responded by prompting me through the procedure of entering the correct settings. On the Nokia phone you can view or edit the settings by pressing the Menu softkey and then selecting Services ➤ Settings ➤ Edit Active Service Settings.

You can then connect to the `hello.wml` page by selecting Services ➤ Go To and typing the URL just as you would for a regular web page. So, for example, if your domain name is `frog-parrot.net`, you'd type `http://frog-parrot.net/hello.wml` if the `hello.wml` file is in the top-level directory. (If your `hello.wml` is in a subdirectory, add the names of the subdirectories to the URL just as you would for any other URL.)

If you're using your own server at home and connecting through cable or ADSL, then you may not have a nice domain name, but you should still have an Internet Protocol (IP) address to which the phone can connect. Depending on how your connection works, your IP address may change from time to time. The operating system will tell you what your current IP address is. In the case of Linux you can find out by looking for the `inet addr` in the output you get from typing the following command:

```
 $ /sbin/ifconfig ppp0
```

If your address is just a set of numbers, it still works perfectly well in the URL. Suppose, for example, that you entered the previous command and you got the following output:

```
ppp0      Link encap:Point-to-Point Protocol
          inet addr:81.49.195.43  P-t-P:193.253.160.3  Mask:255.255.255.255
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1492  Metric:1
          RX packets:1108 errors:0 dropped:0 overruns:0 frame:0
```

```
TX packets:1097 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:3
RX bytes:798514 (779.7 Kb)  TX bytes:98348 (96.0 Kb)
```

In this case, the URL to enter into your phone is `http://81.49.195.43/hello.wml`.

Merely typing the URL into the phone can be tricky! In the case of the Nokia 6100 you can get a list of symbols by hitting the asterisk+plus key (*+). Navigate through this list using the right and left arrow keys until you find the desired symbol (such as . or / for a URL), and then select Use. Fortunately, the fact that the period (.) is the default symbol may save you a little effort. If you have a fixed IP address or domain name, you can also save yourself some typing by making a bookmark to your page.

Once the URL is entered, click OK, and the phone should open your WML page! From here you can click the link to your game, and your phone will download and install it.

So you can see that it's easy to get started developing MIDP games and even to get them up and running on your handset. Just download and install the free development toolkit, build a demo and upload it with a few simple files to a web site, then play!

MAKING IMAGE FILES

If you're wondering where all the image files in this book came from, I drew them myself with a free program called *the gimp*, which you can download from `http://www.gimp.org/downloads`. If you decide to draw your image files with the gimp and you'd like them to have transparent backgrounds, be sure to select a transparent background when you first create a new image file. Transparency can be added later, but it's simpler to have it from the beginning. (A transparent background is nice for game objects because you don't want the rectangular frame of one game object obscuring another game object.) Then when you're done with the image, select the transparent part and choose Filters ➤ Colors ➤ Color to Alpha so that the transparency will be correctly recognized; also, make sure you select Save Background Color when you save the file. You should save it with the `.png` extension so that the gimp will save it in the right format to be used by a J2ME game. One thing to keep in mind when making images is that the difference in screen size from one device to another is the factor that's likely to break your game most dramatically when you try to port it from one device to another. With very small screens, every pixel counts, so a screen size difference that seems insignificant can translate to a major problem for a game. To make your game more portable, you should of course avoid using hard-coded numerical values when drawing the graphics. Additionally, it helps to make different versions of your images in different sizes. If you have only a few image files, it's probably OK to have the game dynamically choose which images to use based on screen calculations, but if you have quite a number of image files, it's usually preferable to maintain different versions of the game's JAR file for different devices. Graphics in the PNG format tend to be pretty small, but they can add up quickly and therefore significantly impact the size of your JAR. This type of optimization is covered in detail in Chapter 10.

Summary

In this chapter you've seen how Java Micro Edition works, how it's implemented, and where it fits in the Java universe. Then we've covered what you need to do in order to set up your development environment and to write an actual program and get it installed and running on your handset. Now that you've finished these steps, you're ready to write some real games!