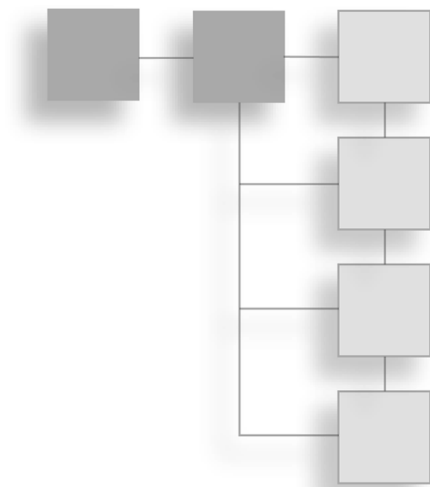


CHAPTER 2

HELLO, WORLD!



M3G provides a 3D graphics library, but not an application model. That's where CLDC and MIDP come in, which define a host environment to create applications for mobile phones. Whereas CLDC takes care of class compilation and execution, MIDP provides the user interface and the deployment model. In this chapter, you develop a MIDlet that you can use as a framework for your own 3D applications.

This chapter shows you how to:

- Use the MIDP application framework.
- Display graphics and receive user input.
- Create a render loop for animated applications.
- Build and test a MIDlet with the Java Wireless Toolkit.
- Download a MIDlet to a phone.

Developing a MIDlet

The application management software controls the lifecycle of your application. To make this possible, you have to derive a class from `javax.microedition.midlet.MIDlet` and implement methods that allow the Java platform to interact with your program.

MIDlet States

Figure 2.1 shows the possible lifecycle states of a MIDlet. Right at the start, your application transfers to the paused state. When paused, a MIDlet should minimize its use of system resources. Instead, allocating resources should be done in the active state, which indicates that your application is ready for execution. Your application enters the destroyed state when it's terminating, which typically happens when the user ends your application.

By overriding the following MIDlet interfaces, you can add custom behavior to the state transitions:

- **new:** When creating a new instance of your MIDlet derived class, the application management software calls the no-argument constructor. However, the MIDlet is in a paused state, which means it shouldn't hold any expensive resources. You should postpone these initializations until `startApp()` is called.
- **startApp():** The application management software calls this method to signal that the MIDlet has entered the active state. In contrast to `new`, this can happen more than once during the lifetime of an application because a MIDlet can transition between the paused and active states any time.
- **pauseApp():** When entering the paused state, the application management software wants your application to free resources. A paused state might indicate that another application now has the focus.

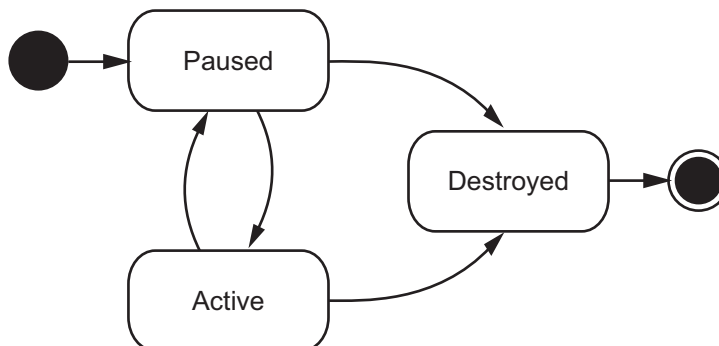


Figure 2.1
A MIDlet can be in one of three states.

Releasing resources allows this application to use them again. If you do this, make sure to reallocate the resources in `startApp()`.

- `destroyApp()`: When the MIDlet has entered the destroyed state, this is your last chance to clean up and save application data for the next start.

State transitions don't complete until the respective method has returned. To avoid blocking the user interface, any lengthy task should be offloaded to a thread. You will see an example of using a thread later in this chapter.

The previously mentioned state transitions are initiated by the application management software. Conversely, your application can also notify the application manager to trigger state changes. `notifyPaused()` and `notifyDestroyed()` signal that your application wants to enter paused and destroyed states. Neither of these methods calls `pauseApp()` and `destroyApp()`; you must make sure to call any methods that release resources. `resumeRequest()` indicates that your application wants to change from paused to active state. If this request is successful, the application management software will call `startApp()`.

Displaying Screens

To display something on the screen, you can use your MIDlet's `startApp()` method as a starting point. The following code illustrates this:

```
/*
 * Mobile 3D Graphics
 * Learning 3D Graphics with the Java Micro Edition
 */

package m3g02;

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

/**
 * MIDlet for <code>HelloWorldSample</code>.
 *
 * @author Claus Hoefele
 */
public class Main extends MIDlet {
    /** The sample. */
    private HelloWorldSample sample;
```

16 Chapter 2 ■ Hello, World!

```
/**
 * Constructor.
 */
public Main() {}

/**
 * Enters the active state. Displays the sample.
 */
public void startApp() {
    if (sample == null) {
        sample = new HelloWorldSample();
        sample.init();
        Display.getDisplay(this).setCurrent(sample);
    }
}

/**
 * Enters the paused state.
 */
public void pauseApp() {}

/**
 * Enters the destroyed state.
 *
 * @param unconditional if true, MIDlet is required to exit
 *                       unconditionally.
 */
public void destroyApp(boolean unconditional) {
    if (sample != null) {
        sample.destroy();
    }
}
}
```

Main derives from MIDlet, so it has to implement the abstract methods that signal a state change: `startApp()`, `pauseApp()`, and `destroyApp()`. In `startApp()`, the MIDlet creates a new `HelloWorldSample` instance that represents the screen used to show your application's user interface. `startApp()` then brings `HelloWorldSample` to the foreground by calling `Display.getDisplay(this).setCurrent()`.

`HelloWorldSample` and any class used for display must inherit from one of `Displayable`'s subclasses. These come in two flavors: those you can use for high-level user interfaces and those you can use for low-level graphics access to the screen.

Screen is the common superclass of high-level screens, which you can use to create user interfaces out of pre-configured building blocks. For example, MIDP offers lists, dialogs, and forms. With forms, you can configure a custom screen out of items such as text fields and choice groups.

High-level user interfaces are convenient for the programmer because you don't have to worry about how to draw the individual components. If, however, you want more direct access to the screen, you have to use Canvas or its subclass GameCanvas. They put every pixel on the screen under your control.

Drawing Graphics

MIDP 2.0 introduced GameCanvas. It inherits all features of Canvas but is also always double-buffered—you first draw in an off-screen buffer and once you are finished, you copy the complete contents to the screen. This avoids the flickering that occurs when the system updates the screen in between drawing calls. Conversely, Canvas supports double-buffering only if the device implemented this feature. In this case, `Canvas.isDoubleBuffered()` returns true.

The implementation of `HelloWorldSample` demonstrates how to use GameCanvas. `HelloWorldSample` gives you a jumpstart on the usage of M3G, which will be complemented by more detailed explanations later in this book.

```
package m3g02;

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
import javax.microedition.m3g.*;

/**
 * Animates a three-dimensional Hello, World! text that's stored in a
 * binary file.
 *
 * @author Claus Hoefele
 */
public class HelloWorldSample extends GameCanvas {
    /** File that stores the 3D scene. */
    private static final String M3G_FILE = "/m3g02/helloworld.m3g";
    /** User ID to find the mesh inside the scene graph. */
    private static final int USER_ID_MESH = 1;

    /** Object that represents the 3D world. */
    private World world;
```

18 Chapter 2 ■ Hello, World!

```
/** Text mesh. */
private Mesh mesh;

/** 3D graphics singleton used for rendering. */
private Graphics3D graphics3d;
/** 2D graphics singleton used for rendering. */
private Graphics graphics;

/**
 * Constructor.
 */
public HelloWorldSample() {
    super(true); // suppress game key events received in key handlers
}

/**
 * Initializes the sample.
 */
public void init() {
    // Get the singletons for rendering.
    graphics3d = Graphics3D.getInstance();
    graphics = getGraphics();

    try {
        // Load World from M3G binary file.
        Object3D[] objects = Loader.load(M3G_FILE);
        world = (World) objects[0];

        // Change the camera's properties to match the current device.
        Camera camera = world.getActiveCamera();
        float aspect = (float) getWidth() / (float) getHeight();
        camera.setPerspective(60, aspect, 1, 1000);

        // Find mesh in scene graph.
        mesh = (Mesh) world.find(USER_ID_MESH);
    } catch (Exception e) {
        System.out.println("Error loading" + M3G_FILE + ".");
        e.printStackTrace();
    }

    // Display scene.
    render(graphics);
    flushGraphics();
}
```

```
/**
 * Destroys the sample.
 */
public void destroy() {
}

/**
 * Renders the sample.
 *
 * @param graphics graphics context for rendering.
 */
protected void render(Graphics graphics) {
    graphics3d.bindTarget(graphics);
    graphics3d.setViewport(0, 0, getWidth(), getHeight());
    graphics3d.render(world);
    graphics3d.releaseTarget();
}
}
```

You have already seen the different states of a MIDlet that you can use to react on external events. In `HelloWorldSample`, `init()` and `destroy()` serve to relay this information from the MIDlet to the display class. Whereas `init()` is called by `Main.startApp()` to create all M3G resources, `destroy()` is called by `destroyApp()` when the application exits.

Part of the resource allocation in `init()` is to get access to a graphics context by calling `getGraphics()`. The returned `Graphics` object provides 2D drawing methods such as `drawLine()` that write in the off-screen buffer. `HelloWorldSample` also creates a `Graphics3D` object that provides 3D drawing methods. Both objects are stored in the class for later use.

Most other M3G parameters are hidden in `helloworld.m3g`, which is a binary file that contains a 3D scene. (The mechanism for creating such files is fully explained in Chapter 10.) `init()` reads the file's contents into a `World` object and draws it on the screen by calling `render()`. This method binds the `Graphics3D` object to the `Graphics` context to tell M3G where to write the output. When `render()` returns, the off-screen buffer contains the graphics output, which is then copied to the screen by calling `GameCanvas.flushGraphics()`. You can see the result in Figure 2.2.

Receiving User Input

For classes derived from `Canvas`, MIDP offers two mechanisms to receive user input: commands that tie soft keys to actions and key handlers.



Figure 2.2
The rendered Hello, World! example.

Commands

A Command object describes the functionality of a soft key and can be added to any Displayable. To receive an event, you write a class that implements CommandListener and register it with Displayable.setCommandListener():

```
public class Main extends MIDlet implements CommandListener {
    /** Exit command. */
    private Command exitCommand = new Command("Exit", Command.EXIT, 1);

    ...

    /**
     * Enters the active state. Displays the sample.
     */
    public void startApp() {
        if (sample == null) {
            sample = new HelloWorldSample();
            sample.init();
            sample.addCommand(exitCommand);
        }
    }
}
```

```
        sample.setCommandListener(this);
        Display.getDisplay(this).setCurrent(sample);
    }
}

...
/**
 * Receives command actions.
 *
 * @param command command.
 * @param displayable source of the command.
 */
public void commandAction(Command command, Displayable displayable) {
    if (command == exitCommand) {
        destroyApp(true);
        notifyDestroyed();
    }
}
}
```

A `Command` contains a soft key's label, type, and priority. Many devices have a standard way of displaying typical operations such as exit and back. If you indicate the correct type, the MIDP implementation can place your command in accordance with the user interface guidelines of your device. The priority describes the importance of the command. A MIDP implementation might make a high-priority command accessible with one button click, whereas a low-priority command might be hidden in a submenu.

A command is added to a `Displayable` with `addCommand()`. After registering a class that implements the `CommandListener` interface, pressing a soft key will result in a call to `commandAction()` in the observing class. `Main` uses this mechanism to exit the application.

Key Events and Game Actions

Unlike commands, you will receive key events without registering a listener. However, key events are unique to subclasses of `Canvas`, which can override `keyPressed()` and `keyReleased()` to implement a key handler.

When choosing which key to use for your application, you will run into the problem whereby different devices have different key layouts. A key to fire a

22 Chapter 2 ■ Hello, World!

bullet might be conveniently located on one device, but not on another. To this end, MIDP introduced game actions that abstract common operations. Instead of listening to the number eight key for example, you can listen to `Canvas.DOWN`. The device manufacturer ensures that the action is mapped to a convenient location such as the navigation key most devices have.

One key event maps to one game action, but not necessarily the other way around. For example, a device might map the same game action to two locations so that the user has a choice. To convert from a key code to a game action, call `Canvas.getGameAction()` in your key handler. Another convenient method is `getKeyName()`, which returns a textual representation of a key code.

When using `GameCanvas` instead of `Canvas`, you can suppress receiving game key events in `keyPressed()` and `keyReleased()`. The device then stops delivering such events, which might speed up your application. Because `HelloWorldSample` doesn't need these events, it calls `super(true)` in its constructor. If you want to receive them, call the superclass's constructor with `false` instead.

Animating Your World

In a game, you usually have constant animations going on to make some interesting effects. This requires an independently running task that updates the animation in regular intervals. In Java, such an independent task is a subclass of `Thread`. However, because `HelloWorldSample` already derives from `GameCanvas`, it implements the interface `Runnable` instead.

```
public class HelloWorldSample extends GameCanvas implements Runnable {
    /** Flag for stopping the animation thread.*/
    private boolean isRunning;
    ...

    /**
     * Initializes the sample.
     */
    public void init() {
        ...

        // Start animation.
        Thread thread = new Thread(this);
        isRunning = true;
        thread.start();
    }
}
```

```
/**
 * Destroys the sample.
 */
public void destroy() {
    isRunning = false;
}

/**
 * Drives the animation.
 */
public void run() {
    while(isRunning) {
        // Advance the animation by rotating the mesh.
        if (mesh != null) {
            mesh.postRotate(-1, 0, 0, 1);
        }

        // Display scene.
        render(graphics);
        flushGraphics();

        try {
            Thread.sleep(20);                // max 50 fps
        } catch (Exception e){}
    }
}
...
}
```

In `init()`, the `Runnable` is turned into a `Thread` and started. This causes the call of the `run()` method where you implement your animation. In `HelloWorldSample`, this method consists of rotating the 3D text, rendering the result to the screen, and waiting for 20 milliseconds. The latter limits your frame rate to a maximum of 50 frames per second. Depending on the time it takes to render the world and the accuracy of `sleep()`, the actual frame rate is slower than that.

When implementing an animation loop, it's inconvenient to receive key events asynchronously via `keyPressed()`. You'll have to think about synchronization because one thread animates the game, and the system delivers key events on another. To make game development easier, `GameCanvas` provides `getKeyStates()` that synchronously retrieves the currently pressed keys. You can test the resulting

int value against bit patterns stored in GameCanvas such as FIRE_PRESSED. With this method, you avoid synchronization issues by calling getKeyStates() as part of your animation loop inside run().

Running Your Application on the Emulator

Before you can run your application, you have to build it. Sun provides you with the Java Wireless Toolkit, which contains the tools necessary to build your application and execute the result in an emulator.

The Build Process

CLDC and MIDP govern different parts of the tool chain to build an application: whereas CLDC specifies how to prepare the class files, MIDP defines how to package your application and deliver it to a device.

The CLDC Tool Chain

The build process of CLDC-based applications differs from the Java Standard Edition in order to accommodate the memory and speed constraints of embedded devices. To decrease the resources necessary to process an application when it's executed, the class file verification is split into an offline part that's done at development time and an online part done at runtime. Class file verification ensures that only valid Java classes are executed.

Figure 2.3 shows the CLDC tool chain. After compilation, all classes must go through the preverifier. This tool prepares the classes by replacing some byte-codes with easier to verify, equivalent substitutions and adding information that speeds up the online verification process. The tradeoff is that this process results in an increased file size. Non-CLDC class file verifiers simply ignore the additional information, so the classes are still valid input to conventional VMs.

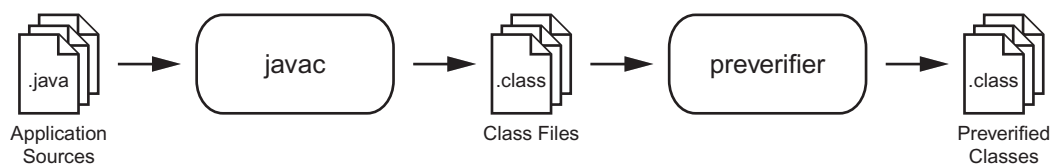


Figure 2.3

Java classes have to be compiled and preverified before using them on a CLDC device.

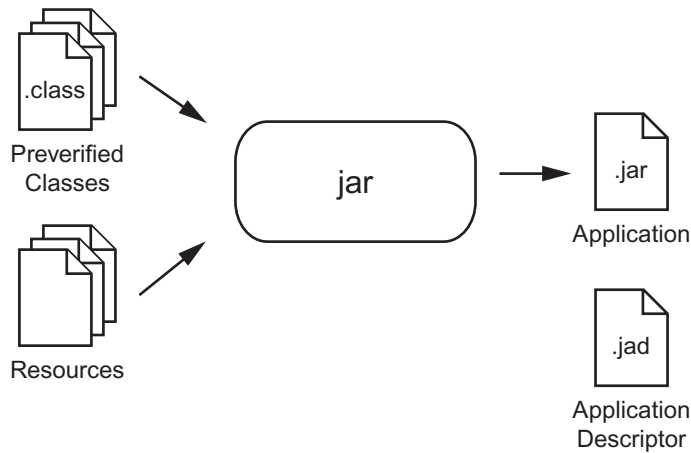


Figure 2.4
The application's classes and resources are packed into a JAR file.

Creating a MIDlet Suite

The application management software deals with *MIDlet suites*, which consist of one or more MIDlets. A MIDlet suite comprises all preverified classes, resources, and other data that is necessary to run your applications in a Java archive file (JAR). MIDlets in the same suite can share this data. Figure 2.4 depicts the process of creating a JAR file from your files.

Tip

You can use a standard zip utility to look into your JAR file to check that all the files are included.

You can accompany the JAR file with an optional Java application descriptor with the file ending .jad. With this file, a device checks whether it can execute your application successfully before starting the possibly lengthy process of downloading the application itself.

The following is the contents of the JAD file for the example in this chapter:

```
MIDlet-1: HelloWorld, HelloWorld.png, m3g02.Main
MIDlet-Jar-Size: 7048
MIDlet-Jar-URL: HelloWorld.jar
MIDlet-Name: HelloWorld
MIDlet-Vendor: Unknown
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.0
```

26 Chapter 2 ■ Hello, World!

The JAD file consists of name value pairs separated by a colon. MIDlet-1 identifies the MIDlet's name, icon, and MIDlet derived class. Because you can have several MIDlets in one suite, you can add additional entries with increasing values for the MIDlet-<n> parameter. The suite itself has a name specified in the MIDlet-Name attribute.

MIDlet-Jar-URL and MIDlet-Jar-Size indicate where to download your application's JAR file and how big it is. Based on the size, the device can reject applications that won't fit on the device. In a similar way, a device will download your application only if the minimum versions specified in MicroEdition-Configuration and MicroEdition-Profile are met. Also when downloading, the MIDlet-Version can be used to check whether the same version is already installed. Finally, you can add some information about yourself in the MIDlet-Vendor attribute.

Tip

You can add your own named properties to the JAD file and read the values with MIDlet.getAppProperty().

The JAD file is optional because the same information can also be added to the MANIFEST.MF file stored inside the JAR file. This is Java's standard way of providing meta information about applications. However, this prevents the device from checking your application before downloading it.

Tip

Avoid conflicts between attributes in the JAD and the MANIFEST.MF file by always providing the same values.

The Java Wireless Toolkit and the KToolbar

Fortunately, you don't have to execute the steps to create your application deliverables manually. Sun offers the Java Wireless Toolkit that comes with a MIDlet emulator as well as tools to make a developer's life easier. Among them is the KToolbar, which serves as a dashboard to build and run MIDlet applications.

Installing the Software

Before you install the Wireless Toolkit, you need a Java Standard Edition Development Kit. Make sure to download the Development Kit rather than the Runtime Environment because you need the Java compiler to build your MIDlets.

The Wireless Toolkit supports the M3G 1.0 API since version 2.2 and supports M3G 1.1 since 2.5. The examples in this book are compatible with either M3G version.

Note

You can find the Java SDK at <http://java.sun.com/javase/> and the Wireless Toolkit at <http://java.sun.com/products/sjwtoolkit/>.

Tip

The Java Wireless Toolkit only supports Windows XP. If you are a Mac OS X or Linux user, try the mpowerplayer (<http://www.mpowerplayer.com/>).

After installation, you will have a new entry for the Wireless Toolkit in the Windows Start menu. Among the new applications, you will also find the KToolbar.

Creating a New Project

Figure 2.5 shows the dialog box that opens after you start the KToolbar and click the New Project button. Enter the project name and the MIDlet's class name in the upcoming dialog box.

The project name decides the name of the folder where your application resides, the name of the JAR and JAD files, and the MIDlet's name as it will show up in the phone's menu. Except for the application folder, you can give each setting individual values after you have finished creating the project.

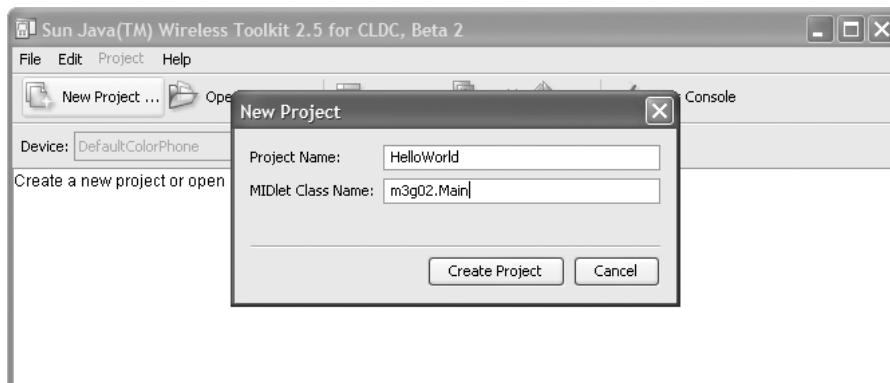


Figure 2.5
Creating a new project.

28 Chapter 2 ■ Hello, World!

In the second text box, you have to enter the fully qualified name of a class derived from MIDlet. Because the samples in this book each live in a chapter specific package, the correct setting in this case is “m3g02.Main.” In the previous section, you find this value in the package definition in the source code. Initially, you’ll have one MIDlet in the suite, but you can add more later on.

After entering the project and MIDlet name, the KToolbar will automatically open the settings dialog box. The API Selection page, shown in Figure 2.6, allows you to select which APIs your application can use. Make sure that Mobile 3D Graphics for J2ME (JSR 184) is included in the list of APIs. If it’s missing, you will



Figure 2.6
The API Selection dialog box.

get compilation errors. The default selection for target platform, the Mobile Service Architecture for CLDC, is fine.

Note

The Java Wireless Toolkit 2.5 doesn't allow you to choose between M3G 1.0 and 1.1. Be careful which methods you call in your application when targeting older devices that only support version 1.0. Alternatively, you can install the toolkit in version 2.3 in parallel to the newer version. This software supports M3G 1.0 exclusively.

Should you need to go back to this dialog box, you can open it by clicking the Settings button in the KToolbar's main screen. The MIDlet has sensible default values, but you might want to have a look at the different pages in the settings dialog box if you want to add another MIDlet to your suite, for example. These settings are written to the JAD file and determine the properties of your application.

Adding Source Code and Resources

After you closed the settings dialog box with the OK button, the KToolbar will create a new directory for your project. This directory is located below the installation directory where you installed the Wireless Toolkit. In there, you will find a directory called apps that contains all the applications. Your application lives in its own folder that has the same name as the project.

Note

The KToolbar doesn't allow you to change where the applications reside; it's always below the apps folder. Neither can you import an existing project from another directory or change the project's directory structure.

Copy the source code of your Java application to the src folder and the resources to res. The sample in this chapter uses two Java classes (Main.java and HelloWorldSample.java) and one resource file (helloworld.m3g). You should end up with a directory structure like this:

```
<Wireless Toolkit Installation Directory>\
- apps\
  - HelloWorld\
    - bin\
      - HelloWorld.jad
      - MANIFEST.MF
```

30 Chapter 2 ■ Hello, World!

```
- lib\  
- res\  
  - m3g02\  
    - helloworld.m3g  
- src\  
  - m3g02\  
    - HelloWorldSample.java  
    - Main.java  
- project.properties
```

Note

You can find all the files of the sample application on the CD-ROM. The files are already sorted in the proper directory structure so you can simply copy the complete folder. Don't forget to copy helloworld.m3g or the application will stop with a `java.io.IOException`.

Building and Running

You are now ready to execute the application. Simply click the Build button and then choose Run. Figure 2.7 shows the emulator's startup screen. You start the MIDlet by using the Select button.



Figure 2.7

You can select a MIDlet of your suite for execution when starting the emulator.

In Figure 2.7, you can see the DefaultColorPhone device. The Wireless Toolkit allows you to customize the skin of the emulator to adapt to different looks, screen sizes, and key layouts. You usually get a skin adapted to a specific device from the device manufacturer.

You can select skins in the KToolbar with the drop-down list labeled Device. I like the DefaultColorPhone device because it has a large screen, which makes it ideal for the screenshots in this book.

Tip

If you want to do your own customizations, have a look at the Basic Customization Guide that comes with the Wireless Toolkit's documentation.

The build step reports any errors in the KToolbar's console window. The same window also displays all system output of your application. However, the toolkit doesn't come with a code editor. For this, you have to install a separate application.

Packaging Your Application

When the KToolbar builds your application, it will automatically compile and preverify all classes so the emulator can execute the application. To create a binary delivery of your application, you have to do one additional step. Figure 2.8 shows the Create Package menu item that triggers this process.

The Create Package command creates a JAR file with all preverified classes and the resources in it and puts it in the bin directory. Together with the JAD file in the same directory, these two files are the binaries for deployment on your device.

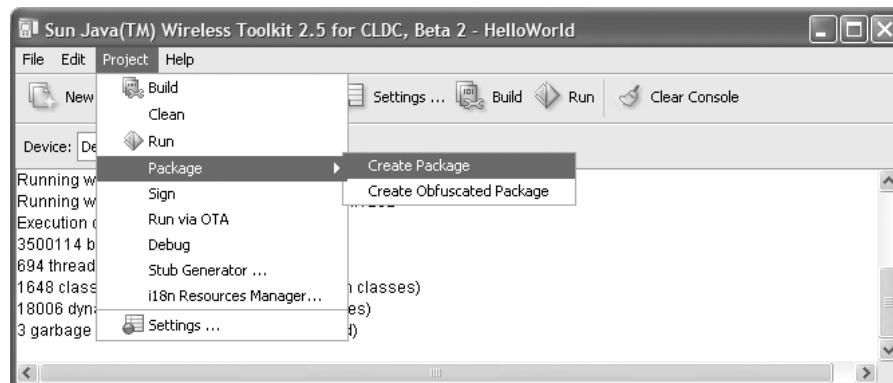


Figure 2.8
Packaging an application for distribution.

Integrated Development Environments

The KToolbar is an effective development environment, but if you prefer to edit, build, debug, and execute MIDlet applications in one comfortable user interface, you might want to look into a full-featured IDE. Two great IDEs are the open source packages NetBeans and Eclipse.

The NetBeans Mobility Pack extends NetBeans with support for MIDP applications. The Mobility Pack includes a version of the Wireless Toolkit, so it's very easy to set up. NetBeans also supports advanced features such as conditional compilation of code and a GUI editor. Eclipse is an alternative to NetBeans, and the free plug-in EclipseME adds MIDP development capabilities.

Note

To get the latest versions of the IDEs, have a look at <http://www.netbeans.org/> and <http://www.eclipse.org/>. You can find EclipseME at <http://eclipseme.org/>.

The plug-ins for NetBeans and Eclipse internally use the tools that come with the Wireless Toolkit. If there is a new version, you can install the latest version in parallel to older versions and select which one to use to test your applications. Best of all, both IDEs support source code debugging—an invaluable tool to verify your MIDlets.

Downloading Your Application to the Phone

Now that you have the JAR file with your application and the JAD file with the application's properties, you can install your software on a device.

Installing Over the Air

An extra section in the MIDP standard, the Over the Air User Initiated Provisioning Specification, describes how MIDlet suites can be deployed wirelessly over a mobile phone network. Installing over the air requires you to have a properly set up network connection for your mobile phone's browser.

The application's JAR and JAD files must reside on a Web server that the device's mobile browser can access. You have to configure the Web server to map JAD files to the MIME type `text/vnd.sun.j2me.app-descriptor` and JAR files to `application/javaarchive`. These settings tell the receiving device what kind of files it downloads, so it can initiate the installation of Java applications. How to

configure MIME types depends on your Web server. If you use a hosted Web server, make sure you talk to your service provider to determine whether these settings are already done.

Next, type the URL of your JAD file into the mobile phone's browser. Your phone will download and inspect the properties of your application and check whether the application is acceptable for download. For example, it will check whether it can provide the MIDP version that your application requires and that is specified in the JAD's `MicroEdition-Profile` attribute. You are also usually given the option to display the application's properties such as size and vendor name.

If you accept the application, the phone will download the JAR file from the URL specified in the JAD's `MIDlet-Jar-URL` attribute. If everything went fine, the application is installed and ready to execute.

Alternative Methods of Installation

To test applications on real devices at development time, it's better to install applications offline because it's fast and cheap. What tools exist and how they work is highly specific to a particular phone and its manufacturer.

Many phones now come with a software suite and a data cable that allows you to back up the phone's memory. In the same way, Java applications can find their way to the phone, although some manufacturers require you to install separate software to do this.

If your phone doesn't come pre-packaged with a data cable, you might be able to buy it as an accessory. Another possibility is to send an application via infrared connection or Bluetooth. You might also want to check whether you can copy an application to a removable memory card and use this medium to install software.

Tool support varies widely; check your manufacturer's developer Web site to determine what's available for your specific phone.

Summary

Well done. You have successfully created and executed your first M3G application.

This chapter introduced developing M3G applications with the MIDP framework. To create a new application, you have to write a class that derives from `MIDlet`. This is the entry point that allows the application management software

34 Chapter 2 ■ Hello, World!

to control the lifecycle of your application. To display something on the screen, you have to create another class that derives from `Displayable`. The example in this chapter used `GameCanvas`, which provides double-buffered access to the screen.

If you need user input, you can use `Commands` that define soft key actions and receive key events. Key events are delivered asynchronously via `keyPressed()` or synchronously via `getKeyStates()`.

When you have finished coding your application, you can build and test it on the emulator. The Java Wireless Toolkit provides all necessary tools. To install the application on real devices, you can either download it or use device specific tools for offline installation. Testing your application on a device is important to avoid any incompatibilities that the emulator didn't catch.

This chapter provided you with an application that you can use as a framework for your own experiments. In the remainder of this book, you learn how to add more M3G functionality to it.