# ;login:

THE MAGAZINE OF USENIX & SAGE

April 2004 • volume 29 • number 2


inside:




USENIX

The Advanced Computing Systems Association

# wireless Java™ application development

**by Qusay H. Mahmoud**

Dr. Qusay H. Mahmoud is an assistant professor at the Department of Computing and Information Science, University of Guelph, and associate chair of the Distributed Computing and Communications Systems Technology program (University of Guelph-Humber).

*qmahmoud@cis.uoguelph.ca*

Developing wireless applications using the Wireless Application Protocol (WAP) is similar to developing Web pages with a markup language, because WAP is browser-based. While Java Servlets and Java Server Pages (JSPs) can be used to generate WAP's WML pages dynamically, all communications between the device and the application go over the wireless link, and this is expensive. In addition, WAP isn't really suitable for developing wireless interactive applications such as mobile games. The Sun Java 2 Micro Edition (J2ME) platform can be used to develop wireless interactive applications or MIDlets that can be downloaded over the air and installed on the device.

This article presents an overview of the genesis of the J2ME platform and walks you through a sample wireless application to give you a flavor of what's involved in developing wireless Java applications. It is worth noting that the J2ME is already deployed on millions of devices, such as cell phones, that are available from Motorola/Nextel, Nokia, and other vendors.

## Introduction to J2ME

The Java 2 Micro Edition (J2ME) is aimed at the consumer and embedded-devices market. It specifically addresses the rapidly growing consumer space that contains commodities such as cellular telephones, pagers, Palm Pilots, set-top boxes, and other consumer devices. It is targeted at two product groups: *personal, mobile, connected information devices* (e.g., cellular phones, pagers, and organizers) and *shared, fixed, connected information devices* (e.g., set-top boxes, Internet TVs, and car entertainment and navigation systems). The groups are addressed using different configurations and profiles.

## Configurations

Cellular telephones, pagers, organizers, etc., are diverse in form, functionality, and feature. For these reasons, the J2ME supports minimal configurations of the Java Virtual Machine (JVM) and APIs that capture the essential capabilities of each kind of device. At the implementation level, a J2ME configuration defines a JVM and a set of horizontal APIs for a family of products that have similar requirements on memory budget and processing power. In other words, a configuration specifies support for: (1) Java programming language features, (2) JVM features, and (3) Java libraries and APIs.

Currently, there are two standard configurations: the Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC). The CLDC is aimed at cellular phones, pagers, and organizers, while the CDC targets set-top boxes, Internet TVs, and car entertainment and navigation systems. In this article we are more concerned with the CLDC.

As you can see from Figure 1, a JVM (e.g., the K Virtual Machine or KVM) is at the heart of the CLDC. Note that CLDC 1.0 was the initial version, but today CLDC 1.1, the enhanced version, is the standard. A major difference between the two is that

CLDC 1.0 didn't include support for floating point numbers (so you could not declare variables of type float or double), but CLDC 1.1 does.

## The K Virtual Machine

The K Virtual Machine (KVM) is a compact, complete, and portable Java virtual machine specifically designed from the ground up for small, resource-constrained devices. The design goal of the KVM was to create the smallest possible complete JVM that would maintain all the central aspects of the Java programming language but would run in a resource-constrained device with a few hundred kilobytes of total memory. The J2ME specification describes that the KVM was designed to be: (1) small, with a static memory footprint (40–80 KB), (2) clean and highly portable, (3) modular and customizable, and (4) as "complete" and "fast" as possible.

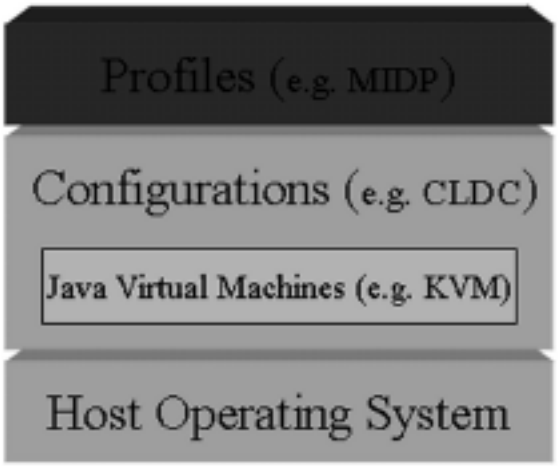


*Figure 1: High-level architecture of J2ME*

## Profiles

The J2ME makes it possible to define Java platforms for vertical markets by introducing profiles. At the implementation level, a profile is a set of vertical APIs that reside on top of a configuration, as shown in Figure 1, to provide domain-specific capabilities such as GUI APIs.

Currently, there is one profile implemented on top of the CLDC, the Mobile Information Device Profile (MIDP), but other profiles are in the works. The MIDP 1.0 was the initial profile and has several constraints (e.g., no support for low-level sockets). MIDP 2.0 is the enhanced version of MIDP with several new features, including end-to-end security (support for HTTPS), as well as support for sockets.

## JVM Supporting CLDC vs. J2SE JVM

There are several differences between a JVM supporting CLDC and the Java 2 Standard Edition (J2SE) JVM. A number of features have been eliminated from a JVM supporting CLDC, either because they are too expensive to implement or because their presence would have imposed security problems. Therefore, in a JVM-supporting CLDC such as the KVM, there is:

- *No floating point support:* CLDC 1.0 does not support floating point numbers and therefore no CLDC-based application can use any floating point numbers and types such as `float` or `double`. This is mainly because CLDC target devices do not have hardware floating point support. Note that CLDC 1.1 *does* include support for floating point numbers.
- *No finalization*: Finalization is not supported, meaning that the CLDC APIs do not include the method `Object.finalize()`, and therefore there is no finalization of class instances.
- *Limited error handling*: Runtime errors are handled in an implementation-specific manner. The CLDC defines only three error classes: `java.lang.Error`, `java.lang.OutOfMemoryError`, and `java.lang.VirtualMachineError`. Other types of errors are handled in a device-dependent manner that would involve terminating the application or resetting the device.
- *No Java Native Interface (JNI)*: A JVM-supporting CLDC does not implement the JNI, mainly for security reasons and because implementing JNI is considered expensive given the memory constraints of the CLDC target devices.
- *No user-defined class loader*: A JVM-supporting CLDC must have a built-in class loader that cannot be overridden or replaced by the user. This is mainly for security reasons.

- *No support for reflection*: No reflection features are supported, and therefore there is no support for RMI or object serialization.
- *No thread groups or daemon threads*: While a JVM-supporting CLDC implements multi-threading, it should not have support for thread groups or daemon threads. If you want to perform thread operations for groups of threads, collection objects should be used to store the thread objects at the application level.

## The CLDC APIs

The J2SE APIs require several megabytes of memory and therefore they are not all suitable for small devices with limited resources. In designing the APIs for the CLDC the aim was to provide a minimum set of libraries that would be useful for application development and profile definition for a variety of small devices. The CLDC library APIs can be divided into two categories:

1. Classes that are a subset of the J2SE APIs: These classes are located in the java.lang, java.io, and java.util packages. They have been derived from the J2SE 1.3. However, note that not all classes from these packages have been inherited.
2. Classes that are specific to the CLDC: These classes are located in the javax.microedition package and its subpackages.

## MIDP Programming

Anyone who has some hands-on programming with Java can start developing MIDP applications (or MIDlets) right after reading this article. MIDP programming is easier than J2SE programming because the MIDP API is simpler. You need to learn about a few classes before you start writing your own MIDlets. Your MIDlet must inherit from the MIDlet class of the javax.microedition.midlet package, then you simply override some methods; the MIDlet lifecycle methods are: startApp(), pauseApp(), and destroyApp(). To handle events, you must implement the CommandListener interface. Here is a simple MIDlet example:

**Listing 1: LoginMIDlet.java**

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

/**
* This login MIDlet prompts the user for a username and a password. If the
* user enters the correct account information, a list of options is
* displayed, otherwise an error message is displayed.
*
* @author: Qusay H. Mahmoud
*/
public class LoginMIDlet extends MIDlet implements CommandListener {
    private Display display;
    private TextField userName;
    private TextField password;
    private Form form;
    private Command cancel;
    private Command login;

    // Constructor
    public LoginMIDlet() {
        userName = new TextField("LoginID:", "", 10, TextField.ANY);
        password = new TextField("Password", "", 10, TextField.PASSWORD);
```

```
    form = new Form("Sign in");
    cancel = new Command("Cancel", Command.CANCEL, 2);
    login = new Command("Login", Command.OK, 2);
}

// MIDlet lifecycle method: called when the MIDlet is started:
public void startApp() {
    display = Display.getDisplay(this);
    form.append(userName);
    form.append(password);
    form.addCommand(cancel);
    form.addCommand(login);
    form.setCommandListener(this);
    display.setCurrent(form);
}

// MIDlet lifecycle method: called when MIDlet is paused:
public void pauseApp() {
}

// MIDlet lifecycle method: called when the MIDlet is destroyed:
public void destroyApp(boolean unconditional) {
    notifyDestroyed();
    }

// Checks if the user enters the correct account information:
public void validateUser(String name, String password) {
    if (name.equals("qm") && password.equals("guessit")) {
        menu();
    } else {
        tryAgain();
    }
}

// Display a list of services:
public void menu() {
    List services = new List("Choose one", Choice.EXCLUSIVE);
    services.append("Check Email", null);
    services.append("New Message", null);
    services.append("Address Book", null);
    services.append("Customize", null);
    services.append("Sign Out", null);
    services.addCommand(new Command("Back", Command.CANCEL, 2));
    services.addCommand(new Command("Select", Command.OK, 2));
    display.setCurrent(services);
}

// Display an error message if the user enters the incorrect account info:
public void tryAgain() {
    Alert error = new Alert("Login Incorrect", "Please try again", null,
                AlertType.ERROR);
    error.setTimeout(Alert.FOREVER);
    userName.setString("");
    password.setString("");
    display.setCurrent(error, form);
}

// Handle events:
```

```
        public void commandAction(Command c, Displayable d) {
            String label = c.getLabel();
            if(label.equals("Cancel")) {
                destroyApp(true);
            } else if(label.equals("Login")) {
                validateUser(userName.getString(), password.getString());
            }
            // add code to handle user's selection from list of services
        }
    }
```

In Listing 1, the midlet and lcdui packages are imported. The midlet package defines the MIDP, and the lcdui package provides graphical user interface APIs for implementing user interfaces for MIDP applications. It is worth noting that the lcdui package is not a subset of Swing/AWT, simply because the Swing/AWT assumes certain user interaction and provides a rich feature set (such as resizing overlapping windows) not found on mobile devices. The basic unit of interaction on a mobile device is the screen – users' interaction with wireless applications by navigating through screens.

Each MIDlet must extend the MIDlet class, similar to applets, which allows for the orderly starting, stopping, and cleanup of the MIDlet. Therefore, a MIDlet must not have a public static void main() method.

In LoginMIDlet, the Command class is used to encapsulate the semantic information of an action. The command itself contains only information about a command, but not the actual action that happens when a command is activated. The action is defined in a CommandListener associated with the screen. Let's look at the following command statement:

```
Command infoCommand = new Command("Info", Command.SCREEN, 2);
```

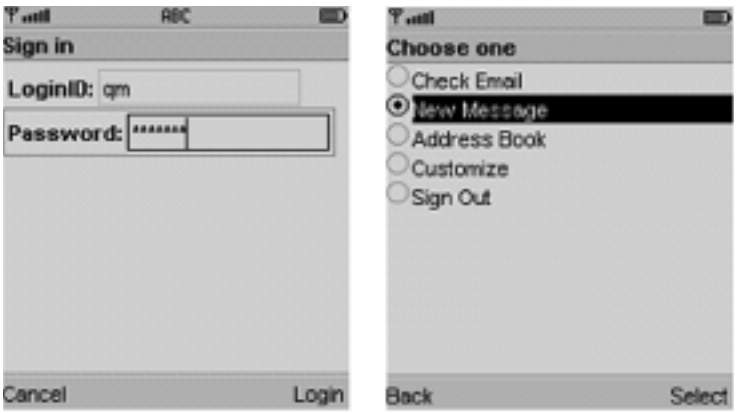


*Figure 2: LoginMIDlet launched and activated*

A command contains three pieces of information: a label, a type, and a priority. The label (which is a string) is used for the visual representation of the command. The type of the command specifies its intent. And the priority value describes the importance of this command relative to other commands on the screen. A priority value of 1 indicates the most important command, and higher priority values indicate commands of lesser importance. When the application is executed, the device chooses the placement of a command based on the type of the command, and places similar commands based on their priorities.

Figure 2 shows the screens when the application is first launched.

## Development Tools

There are several commercial and freely available tools for developing wireless Java applications. My favorite tool is Sun's J2ME Wireless Toolkit (J2ME WTK), which is easy to use and freely available. The J2ME WTK provides a comprehensive tool set and emulators for developing and testing wireless applications, and it is available for the Windows, Linux, and Solaris platforms. It simplifies the development of wireless applications by automating several steps such as preverification and creating Java Archive

(JAR) and Java Application Descriptor (JAD) files (more on this later). Figure 3 shows the interface for the J2ME WTK.

The J2ME WTK can be downloaded from *http://java.sun.com/products/j2mewtoolkit*. To test the LoginMIDlet described above, create a new project, call it Login, and call the MIDlet LoginMIDlet. Then copy my LoginMIDlet.java (the above code) to the apps\Login\src directory of your J2ME WTK installation. The next step is to compile (click on the Build button) and run (click on the Run button) the application. You can choose a device to emulate the application on.

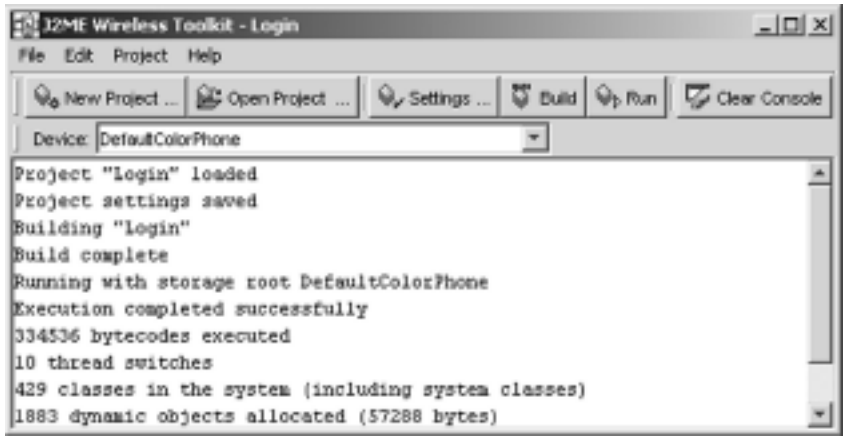


*Figure 3: The J2ME Wireless Toolkit*

## Behind the Scenes

As I mentioned above, the J2ME WTK automates the processes of preverification and packaging of the application. This is done when you click on the Build and Run buttons. So what are preverification and packaging?

### Class Verification

In the J2SE Java virtual machine, the class verifier is responsible for rejecting invalid class files. A JVM-supporting CLDC must be able to reject invalid class files as well, but the class verification process is expensive and time-consuming and, therefore, is not ideal for small, resource-constrained devices. The KVM designers decided to move most of the verification work off the device and onto the desktop, where the class files are compiled or onto a server machine from which applications are being downloaded. This step (off-device class verification) is referred to as preverification. The device is simply responsible for running a few checks on the preverified class file to ensure that it was verified and is still valid.

Therefore, after compiling the .java into .class, the .class file must be preverified using the preverify command (in J2ME WTK). This command preprocesses the program for use by the KVM without changing the name of a class. The preverify command takes a class or a directory of classes and preprocesses them. Luckily, this task is automated by the J2ME WTK.

### Packaging the Application

If an application consists of multiple classes, a JAR file is used to group all the classes together so that the application is easy to distribute and deploy. In the above example, a JAR file (Login.jar) is created – the J2ME WTK automates this using the jar command.

The next step in packaging is creating a manifest file (or application descriptor), which provides information about the contents of the JAR file. The application descriptor is used by the application management software on the device to manage the MIDlet. It is also used by the MIDlet itself to configure specific attributes. The file extension of the application descriptor is jad, which stands for Java Application Descriptor. There is a predefined set of attributes to be used in every application descriptor. One of the attributes is the MIDlet-Jar-Size, which is used by the application management soft-

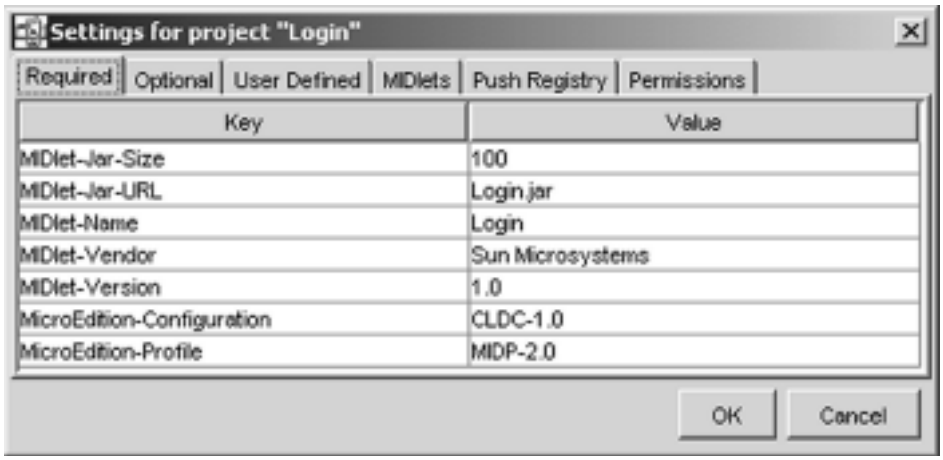


*Figure 4: JAD file for the Login project*

ware to determine whether the device is capable of running the MIDlet before it downloads it (over the air) to the device. Figure 4 shows the JAD file for the Login project (click on the Settings button of the J2EME WTK to see this).

## Deploying Applications

If you have a Java-enabled cell phone from Motorola/Nextel, you can download applications on it either over the air (you'll get billed for the air time) or from the Internet through your PC using a data cable. For more information, visit *http://idenphones.motorola.com*.

Once you have tested the application and you are satisfied with what you see, you can deploy it on a Web server simply by uploading its JAR and JAD files to a Web server. Now your application is downloadable. However, you need to add the following new MIME type to your configuration file and restart the Web server:

```
text/vnd.sun.j2me.app-descriptor  jad
```

## Integrating WAP and J2ME

MIDlets combine excellent online and offline capabilities that are useful for the wireless environment, which suffers from low bandwidth and network disconnection. Integrating WAP and MIDP opens up possibilities for new wireless applications and over-the-air distribution models. Therefore, WAP and MIDP should be viewed as complementary rather than competing technologies. In order to facilitate over-the-air provisioning, there is a need for some kind of an environment on the handset that allows the user to enter a URL for a MIDlet, for example. This environment could very well be a WAP browser. Similar to Java Applets that are integrated into HTML, MIDlets can be integrated into a WML page. The WML page can then be called from a WAP browser, and the embedded MIDlet gets installed on the device. In order to enable this, a WAP browser (with support for over-the-air provisioning) is needed on the device. An alternative approach for over-the-air provisioning is the use of Short Message Service (SMS), as has been done by Siemens, where the installation of MIDlets is accomplished by sending a corresponding SMS. If the SMS contains a URL to a JAD file specifying a MIDlet, then the recipient can install the application simply by confirming the SMS.

## Conclusion

This article introduced the J2ME platform and described the development model for wireless Java applications. The MIDlet provided shows that programming with J2ME is easier than programming with J2SE, because the API is simpler and there are only a dozen classes you need to learn. I hope this article helps you get started with wireless Java application development. Stay tuned for more articles on wireless Java.

### FURTHER READING

Java 2 Micro Edition (J2ME):
*http://java.sun.com/j2me*

The J2ME Wireless Toolkit:
*http://java.sun.com/products/j2mewtoolkit*

*Learning Wireless Java*, by Qusay H. Mahmoud, available from O'Reilly

Motorola iDEN phones:
*http://idenphones.motorola.com*

Sun's Source for Java Developers:
*http://java.sun.com*