Oracle® Java Micro Edition Software Development Kit

Developer's Guide

Release 8 for Windows

E50624-01

April 2014

Describes how to use the Oracle Java Micro Edition Software Development Kit (Java ME SDK) on Windows

ORACLE®

Oracle Java Micro Edition Software Development Kit Developer's Guide, Release 8 for Windows

E50624-01

Contents

Preface

This guide describes how to use the Oracle Java Micro Edition Software Development Kit (Java ME SDK) to develop embedded applications in the NetBeans integrated development environment (IDE). Oracle Java ME SDK 8 contains a complete implementation of the Oracle Java ME Embedded 8 software runtime.

Together, these products support embedded software development on:

- Windows platform in emulation
- Qualcomm Internet of Everything (IoE) platform (for more information, see *Oracle Java ME Embedded Getting Started Guide for the Reference Platform (Qualcomm IoE))*
- Raspberry Pi (for more information, see *Oracle Java ME Embedded Getting Started Guide for the Reference Platform (Raspberry Pi))*

Audience

This document is intended for developers of embedded software who want to develop applications using Oracle Java ME SDK 8 on Windows.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Operating System Commands

This document does not contain information about basic commands and procedures such as opening a terminal window, changing directories, and setting environment variables. See the software documentation that you received with your system for this information.

Shell Prompts

Shell	Prompt
Bourne shell and Korn shell	`$`
Windows	`>`

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Part I

Introduction

Part I provides an introduction to Oracle Java ME SDK 8. It describes how to install the necessary components, create and run a sample application.

Part I contains the following chapters:

1

Before You Begin

Oracle Java ME SDK 8 is a sophisticated and useful tool for programmers who want to develop embedded applications. It can be used with NetBeans IDE 8.0 running on Microsoft Windows 7 (32-bit and 64-bit).

Note: There is no need to install the Oracle Java ME Embedded 8 emulation runtime, because it is implemented in Oracle Java ME SDK 8.

Before you begin, ensure that all the latest Windows 7 updates and service packs are installed.

Installing the Java SE Platform

The Oracle Java ME SDK 8 software requires the Java Platform, Standard Edition Development Kit (JDK) release 7 update 55 or later installed on your computer. This guide assumes you have already installed the JDK. Otherwise, you can download it from http://www.oracle.com/technetwork/java/javase/downloads

Installing the Oracle Java ME SDK 8 Platform

To install Oracle Java ME SDK 8:

1. If you already have the Java ME SDK installed, uninstall the previous version:
   a. If you have Java ME SDK data that you want to save, copy it to a safe location.
   b. In the notification area of the Windows taskbar, right-click the Java ME SDK Device Manager icon and select **Exit**.
   c. Start the Java ME SDK Installer Wizard by opening the Windows Start Menu, selecting **All Programs**, and then **Uninstall** under the **Java ME Platform SDK 8.0** folder. Alternatively, you can use the Programs and Features window in the Windows Control Panel.
   d. On the first step of the wizard, select the option to remove the user data directory.
   e. Follow the other steps of the wizard.
2. Download Oracle Java ME SDK 8 from http://www.oracle.com/technetwork/java/javame/javamobile/download/sdk
3. Double-click the executable file and follow the steps of the installation wizard.

Installing and Starting NetBeans IDE 8.0

To install NetBeans IDE 8.0:

1. Download the version of NetBeans IDE 8.0 that supports Java ME from `https://netbeans.org/downloads/`.
2. Double-click the executable file and follow the steps of the installation wizard.
3. Double-click the icon on the desktop to start NetBeans IDE 8.0.

> **Note:** Ensure that you install the most recent version of NetBeans IDE 8.0 and apply all available updates related to Java ME support.

For more information about working with NetBeans IDE 8.0, see Chapter 2, "Creating a Java ME SDK 8 Sample Project."

2

Creating a Java ME SDK 8 Sample Project

This chapter describes NetBeans IDE 8.0 which provides a rich, visual environment for developing embedded applications and numerous tools to improve the programming process.

Oracle Java ME SDK 8 provides two plugins for working with NetBeans IDE 8.0:

- Java ME SDK Tools plugin
- Java ME SDK Demos plugin (optional)

Installing Java ME SDK Plugins

To install the Java ME SDK plugins for NetBeans:

1. Download the ZIP archive with the plugins from http://www.oracle.com/technetwork/java/javame/javamobile/download/sdk
2. Extract the contents of the archive to a directory on your local machine.
3. Start NetBeans IDE, open the **Tools** menu, and then select **Plugins** to open the Plugins window.
4. Uninstall the previous Java ME SDK plugins:
   - On the **Installed** tab, select the Show Details check box, then select **Java ME SDK Tools** and **Java ME SDK Demos**, and click **Uninstall**.
5. On the **Settings** tab, ensure that **Additional Development Plugins** and **Latest Development Build** are deselected.
6. Click the **Add** button in the lower right of the **Settings** tab.
7. In the Update Center Customizer window, do the following:
   - In the **Name** field, enter **Java ME SDK Update Center**.
   - Select **Check for updates automatically**.
   - In the **URL** field, use the `file` command to point to the location where you extracted your plugins, for example:

     ```
     file:/C:/My_Update_Center_Plugins/updates.xml
     ```

   - Click **OK**.
8. On the **Settings** tab, select the **Java ME SDK Update Center** that you just added.
9. On the **Available Plugins** tab, select **Java ME SDK Tools** and **Java ME SDK Demos**, and click **Install**.

10. Follow the steps in the NetBeans IDE Installer Wizard.
    - Accept the license terms.
    - If additional Validation screens appear, click **Continue**.
11. Click **Finish** to restart NetBeans IDE 8.0.
12. After NetBeans IDE 8.0 restarts, open the **Tools** menu and select **Plugins** to open the Plugins window.
13. On the **Installed** tab, click **Category** to sort the plugins. Ensure that **Java ME SDK Tools** and **Java ME SDK Demos** plugins are active (if they are not, then select them and click **Activate**).
14. When the Java ME SDK plugins are active, click **Close**.

Creating a Sample IMlet File

This section describes how to create a sample IMlet file (`IMletDemo.java`) from the code provided in Example 2–1. This IMlet file is used in the next section, "Creating a New Project". To create a sample IMlet file:

1. Copy the code shown in Example 2–1 into a text file. Use Notepad rather than WordPad, to avoid any unnecessary extra characters.
2. Save the file with the following name: `IMletDemo.java`.

Example 2–1 Code for the Sample IMletDemo.java Project in NetBeans IDE

```
package imletdemo;
import javax.microedition.midlet.MIDlet;

public class IMletDemo extends MIDlet {

    boolean bFirst = false;

    public void startApp() {
        try {
            // Perform startup operations
        } catch (Exception ex) {
            ex.printStackTrace();
            return;
        }

        bFirst = true;
        System.out.println("IMlet Demo is already started...");
        // Start program here

    }

    public void pauseApp() {
         //  Pause the application
    }

    public void destroyApp(boolean unconditional) {
       // Close all resources that have been opened

    }


}
```

Creating a New Project

This section describes how to create a new embedded project using Oracle Java ME SDK 8 and NetBeans IDE 8.0.

1. Open the **File** menu and select **New Project**.
2. In the New Project window, select **Java ME Embedded** from the **Categories** list and **Java ME Embedded Application** from the **Projects** list. Click **Next**.
3. In the New Java ME Embedded Application window, specify **IMletDemo** in the **Project Name** field at the top, and **imletdemo.Midlet** in the **Create Midlet** field at the bottom. Click **Finish**.

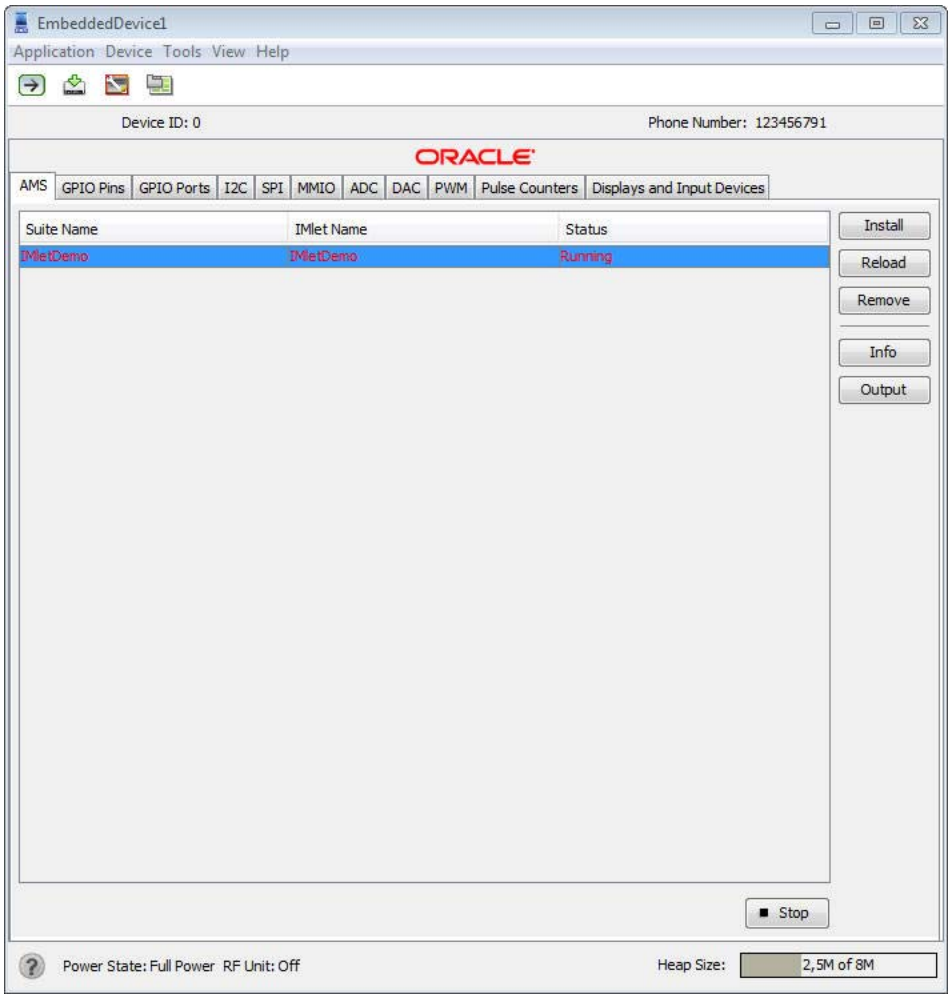
Including Sample IMlet Code and Running the Project

Now you can update the generic project with the sample code that you created earlier in Creating a Sample IMlet File. To include sample IMlet code and run the project:

1. Right-click the **Midlet.java** file in the **Projects** window and select **Properties**.
2. In the Properties window, note the path to the `imletdemo` directory in the **All Files** field.
3. Copy the `IMletDemo.java` file you created in Creating a Sample IMlet File to the `imletdemo` directory.
4. Delete the file `Midlet.java` from the directory.
5. This changes the project tree: instead of `Midlet.java`, the project tree should contain `IMletDemo.java`.
6. Right-click the **IMletDemo** project in the **Projects** window and select **Properties**.
7. In the **Application Descriptor** category, open the **MIDlets** tab.
8. Select the **IMletDemo** row (where the **imletdemo.Midlet** class is flagged in red, because you removed the class file) and click **Edit**.
9. In the **Edit MIDlet** window, select the **imletdemo.IMletDemo** class from the **MIDlet Class** drop-down list. Click **OK**.
10. Click **OK** to close the **Project Properties** window.
11. Clean and build the **IMletDemo** project by selecting it in the **Projects** window and clicking on the hammer-and-broom icon in the NetBeans IDE 8.0 toolbar, or by selecting **Clean and Build Project (IMletDemo)** in the **Run** menu.
12. Run the **IMletDemo** project by selecting the green right-arrow icon in the NetBeans IDE 8.0 toolbar, or by selecting **Run Project (IMletDemo)** in the **Run** menu.

    If successful, the EmbeddedDevice1 emulator starts with the IMletDemo suite running, as shown in Figure 2–1. For more information about the main window of the Java ME Embedded Emulator, see Understanding the Main Window.

Figure 2–1 The EmbeddedDevice1 Emulator Running IMletDemo



Part II

Devices

Part II provides an introduction to devices—the most important and fundamental features of Oracle Java ME SDK 8.

Part II contains the following chapters:

3

Using the Emulators

The Oracle Java ME SDK 8 embedded emulation environment provides you with a platform to test and run Oracle Java ME Embedded Profile (MEEP) IMlet suites without installing those IMlet suites onto an embedded device.

This is done using two default embedded emulators (EmbeddedDevice1 and EmbeddedDevice2). These emulators do not represent a specific device, but provide a correct implementation of the APIs for this platform.

The `Qualcomm_IoE_Device` emulator, which is also described in this chapter, provides an emulation of the Qualcomm Internet-of-Everything (IoE) embedded hardware device. For more information, see *Oracle Java ME Embedded Getting Started Guide for the Reference Platform (Qualcomm IoE)*.

This chapter describes how to run and use the Java ME Embedded Emulator.

Starting the Emulator

If the Embedded emulator is not displayed when you run the sample project described in Chapter 2, "Creating a Java ME SDK 8 Sample Project.", it can be started from the Windows command line. For more information, see Appendix A, "Using the Command-Line Emulator".

Alternatively, click the Windows Start menu, select **All Programs**, open the **Java ME Platform SDK 8.0** folder, and then select **Java ME Embedded Emulator**.

You can also run `emulator.exe` under `bin` in the Java ME SDK installation directory. The default location is `C:\Java_ME_platform_SDK_8.0\bin\emulator.exe`.

Understanding the Main Window

The main window of the Java ME Embedded Emulator is shown in Figure 2–1. The name of the current emulated device is displayed in the title of the main window (for example, EmbeddedDevice1).

The menu bar contains the following menus:

- **Application**: Used to install and run IMlet suites, and to exit the emulator.
- **Device**: Used to view messages addressed to the device through the JSR 120: Wireless Messaging API (WMA).
- **Tools**: Used to manage landmarks, the file system, connectivity, and start the External Events Generator.
- **View**: Used to configure desktop and exit behavior, view output from a running application, and view logging information from the emulated device.

- **Help**: Used to open the context-sensitive help, and view the Java ME SDK release and copyright information.

Below the menu bar is the toolbar with buttons that provide shortcuts for the following operations:

- Run IMlet suite
- Install IMlet suite
- Start the External Events Generator
- Always display the emulator window on top of other windows

> **Note:** When the Device Manager detects an external embedded device, only the **Application**, **View**, and **Help** menus, and the **Run IMlet Suite**, **Install IMlet Suite**, and **Emulator window always on top** buttons in the toolbar are available in the main emulator window.

Below the toolbar, there are two labels:

- **Device ID**: A numerical identifier that is unique for each device
- **Phone Number**: A number used by the emulator to send messages to itself for testing purposes

Below the Oracle logo, there are 11 tabs:

- **AMS**: The Application Management System (AMS) tab displays a table that lists installed IMlet suites, showing the name of the suite and IMlet, and its status. This is the default tab that opens when you start the Java ME Embedded Emulator. You can select a suite and perform one of the following operations by clicking the corresponding button to the right of the table:
  - **Install**: Specify the path or URL to the IMlet suite location and the security domain to load the IMlet suite into the emulator.
  - **Reload**: Reload the selected suite.
  - **Remove**: Remove the selected suite.
  - **Info**: Display information about the selected suite.
  - **Output**: Open the output console of the selected suite.

  At the bottom of the **AMS** tab, there is a button that can be used to start and stop an IMlet suite. It is labeled either **Start** or **Stop**.
- **GPIO Pins**: The General Purpose Input/Output (GPIO) Pins tab displays which pins are configured for input/output, their values, and which port they belong to.
- **GPIO Ports**: The GPIO Ports tab displays a list of ports, port direction (input/output), and their maximum and current values.
- **I2C**: The Inter-Integrated Circuit (I2C) tab displays information for the selected slave device, data sent to a master device, and data received from a master device.
- **SPI**: The Serial Peripheral Interface (SPI) tab displays information for the selected slave device, data sent to a master device, and data received from a master device. If you have created a custom implementation with the Custom Device Editor, the **Slave** drop-down list might have additional slaves.
- **MMIO**: The Memory-Mapped Input/Output (MMIO) tab displays memory configuration and memory content for the selected device.

- **ADC**: The Analog-to-Digital Converter (ADC) tab displays the current channel, converter number, sampling intervals, minimum and maximum values, and other information.
- **DAC**: The Digital-to-Analog Converter (DAC) tab displays the current channel information, converter characteristics, and a graphic display of signal characteristics with the x-axis showing the digital input and the y-axis showing the analog output.
- **PWM**: The Pulse Width Modulation (PWM) tab displays the amount of electrical power flowing to a device.
- **Pulse Counters**: The Pulse Counters tab displays the identifier, counter name, counter number, counter type, and the pins to which the counters are bound.
- **Displays and Input Devices**: The Displays and Input Devices tab shows information about specific displays and input devices attached to your emulated device, including both primary and auxiliary displays.

Below the tabs, the emulator status bar contains information about the power state, and the memory indicator showing used and total heap memory.

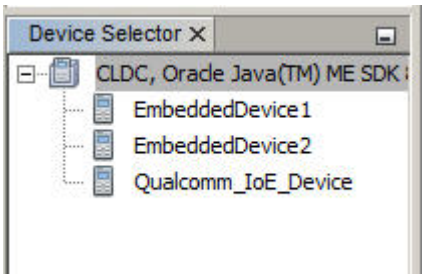
Note: For more information about the Java ME Embedded Emulator GUI, open the **Help** menu and select **Help Contents** to see the help topics. For context-sensitive help, press **F1**. This will open the topic for the window or tab that is currently open.

Running Emulators

Oracle Java ME SDK 8 runs applications on an emulator or an external device. The Device Manager automatically starts detecting external devices when Oracle Java ME SDK 8 is installed. The default emulators are automatically found and displayed in the Device Selector window.

To view the Device Selector in NetBeans IDE, open the **Tools** menu, select **Java ME**, and then **Device Selector**. The Device Selector window is shown in Figure 3–1.

Figure 3–1 Available Devices in the NetBeans IDE 8.0 Device Selector

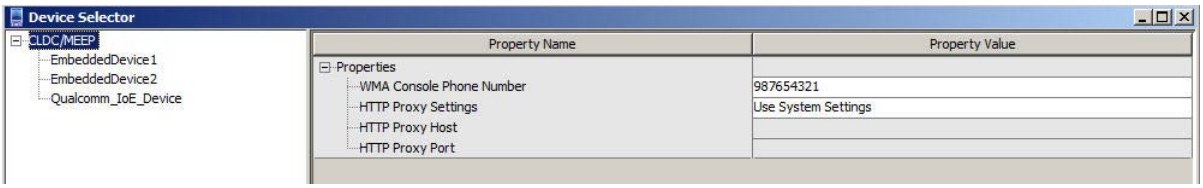


Alternatively, run the `device-selector.exe` file under `bin` in the Java ME SDK installation directory. For example, you can use the following command:

```
C:\> Java_ME_platform_SDK_8.0\bin\device-selector.exe
```

This command displays the standalone Device Selector, as shown in Figure 3–2.

Figure 3–2 The Standalone Device Selector



When an Oracle Java ME SDK 8 project is run from NetBeans IDE 8.0 or from the command line, the default emulator starts (it is defined by the Java ME platform selected for the project). If you do not want to use the default emulator, right-click any emulator in the Device Selector window, and select the project or JAR file with the application you want to run.

To open an emulator without running an application, run the `emulator.exe` file under `bin` in the Java ME SDK installation directory. For example, to run the EmbeddedDevice1 emulator, use the following command:

```
C:\> Java_ME_platform_SDK_8.0\bin\emulator.exe -Xjam -Xdevice:EmbeddedDevice1
```

Alternatively, you can double-click the emulator shortcut installed on your Windows desktop, or use the shortcut in the **Start** menu under **All Programs** in the **Java ME Platform SDK 8.0** folder.

To run an application from the emulator, click the **Application** menu and select **Run IMlet Suite**. Provide the path to the application and any other information, and click **OK**.

Running the Qualcomm_IoE_Device Emulator

The Qualcomm_IoE_Device emulator is based on MEEP, but for a specific embedded platform, the Qualcomm IoE device. Many of the menus and settings are the same as in the MEEP emulator.

To run the Qualcomm_IoE_Device emulator from the Windows command line:

1. Open the `bin` directory under the Oracle Java ME SDK 8 installation directory:

   ```
   C:\> cd Java_ME_platform_SDK_8.0\bin
   ```

2. To start the Qualcomm_IoE_Device emulator without running an application, use the following command:

   ```
   C:\> emulator.exe -Xdevice:Qualcomm_IoE_Device -Xjam
   ```

   To start the Qualcomm_IoE_Device emulator with a running application, use the `-Xdescriptor` option. For example, to run the `DataCollectionDemo` sample application, use the following command:

   ```
   C:\> emulator.exe -Xdevice:Qualcomm_IoE_Device -Xdescriptor:C:\Java_ME_
   platform_SDK_8.0\apps\DataCollectionDemo\DataCollectionDemo.jad
   ```

4

Using the External Events Generator

The External Events Generator enables you to test the capabilities of your device by simulating events on that device. For example, you can send pulses to a pulse counter.

To start the External Events Generator, click the External Events Generator icon in the main emulator window, or open the **Tools** menu and select **External Events Generator**. For information about starting the emulator, see Chapter 3, "Using the Emulators".

Although the External Events Generator functionality is similar for all default emulators (EmbeddedDevice1, EmbeddedDevice2, and Qualcomm_IoE_Device), the tabs are not the same.

The External Events Generator has the following tabs:

- **ADC**: The Analog-to-Digital Converter (ADC) tab can be used to test analog input.
- **GPIO**: The General Purpose Input/Output (GPIO) tab displays input pins for a specific device. You can create a custom device to represent a different GPIO device.

  For information about the GPIO interface, see the Device I/O API docs in the `device-io.zip` file under `\docs\api\` in the Java ME SDK installation directory.
- **I2C**: The Inter-Integrated Circuit (I2C) tab is available only for the Qualcomm_ IoE_Device emulator. It enables to emulate input for all available sensors. You can adjust the settings of the G-sensor, light sensor, and temperature sensor on the device. For information about developing applications that use these sensors, see the documentation for the corresponding sensor. The emulator uses the following sensor models:
  - G-sensor: Bosch BMA150
  - Light sensor: Intersil ISL29011
  - Temperature sensor: ON Semiconductor ADT7481
- **Location**: This tab can be used to set and test the location functionality of the device.
- **MMIO**: The Memory-Mapped Input/Output (MMIO) tab is available only for the EmbeddedDevice1 and EmbeddedDevice2 emulator. If not already selected, select the following default device:
  - **BIG_ENDIAN_DEVICE**: A big endian device that contains all block types (byte, short, int, and block).

Note: If you are using a custom device created with the Custom Device Editor, the MMIO device list might include additional devices. For more information about the Custom Device Editor, see Chapter 5, "Working with Devices"

For information about the MMIO interface, see the Device I/O API documents in the `device-io.zip` file and the Embedded Support API documents in the `embedded-support-api.zip` file under `\docs\api\` in the Java ME SDK installation directory.

- **Power Management**: The Power Management tab enables you to emulate the battery life of an external device, in seconds.
- **Pulse Counters**: The Pulse Counters tab displays the current pulse counters on the device. The default configurations for EmbeddedDevice1 and EmbeddedDevice2 emulators are:
  - COUNTER_PA0
  - COUNTER_PB3
  - COUNTER_PB10
  - COUNTER_PA3

  There is only one default counter for the Qualcomm_IoE_Device emulator.

  You can configure the pulse counters you want and send a signal to the configured pulse counter by clicking **Send Pulse**.
- **SPI**: The Serial Peripheral Interface (SPI) tab is available only for the Qualcomm_IoE_Device emulator. It can be used to configure a sample accelerometer sensor.

  Move the slider of the X, Y, and Z acceleration scale to change the x, y, and z values transmitted over the SPI. The minimum and maximum values for the sliders in the G-sensor are defined by the Java ME application.

  The G-sensor sample duplicates in emulation the functionality of the digital, triaxial acceleration sensor on the Qualcomm IoE embedded device. The acceleration sensor is used to sense tilt, motion, and shock vibration in embedded devices, such as medical instruments, computer peripherals, and monitoring devices. For information about writing applications that use the G-sensor, see the documentation for the sensor. The emulator uses the following G-sensor model: Bosch BMA150.

5

Working with Devices

This chapter describes the Oracle Java ME SDK 8 components that enable you to work effectively with devices, including making device connections, creating, configuring, and customizing devices, and detecting external embedded devices.

The Device Manager is an Oracle Java ME SDK 8 service used to manage both emulated and external devices. As soon as you install the Java ME SDK, the Device Manager automatically starts to detect available devices (including any connected hardware devices).

Alternatively, run the `device-manager.exe` file under `bin` in the Java ME SDK installation directory. For example, you can use the following command:

```
C:\> Java_ME_platform_SDK_8.0\bin\device-manager.exe
```

Oracle Java ME SDK 8 provides several components for working with emulated and external devices. This chapter contains the following sections:

- "Using the Device Connections Manager"
- "Using the Device Selector"
- "Using the Custom Device Editor"
- "Making Device Connections"
- "Additional Peripherals"

Using the Device Connections Manager

The Device Manager is started automatically when Oracle Java ME SDK 8 is installed. However, it is primarily used for external embedded hardware attached to the computer.

To start the Device Connections Manager, right-click the **Oracle Java ME SDK 8.0 Device Manager** icon in the notification area of the Windows taskbar and select **Manage Device Connections**.

To add a new device connection, click **Add**, enter an IP address or host name, and click **OK**. The Java ME SDK automatically detects available devices, listing them in the **IP Address or Host Name** drop-down list. If an embedded hardware device is attached, you can select a COM port for the hardware connection in the **Select COM Port** drop-down list.

If you have an address you no longer want to detect, select the address and click **Remove**. The device will no longer be displayed in the Device Connections Manager.

To see a list of registered devices and their configuration information, right-click the **Oracle Java ME SDK 8.0 Device Manager** icon in the notification area of the Windows taskbar and select **Registered Devices**.

Using the Device Selector

The Device Selector lists the devices detected by the Device Manager, grouped by platform. The Device Selector can be opened as a tab in NetBeans IDE, or as a separate window.

To access the **Device Selector** tab in NetBeans IDE, open the **Tools** menu, select **Java ME**, and then **Device Selector**.

To open the Device Selector in a separate window, run the `device-selector.exe` file under `bin` in the Java ME SDK installation directory. For example, you can use the following command:

```
C:\> Java_ME_platform_SDK_8.0\bin\device-selector.exe
```

The list in the Device Selector matches the list displayed in the Registered Devices window.

Note: When an external hardware device is detected, such as a Raspberry Pi or Qualcomm IoE (MB997B) embedded board, it appears in the Device Selector window with a sequential number on the end, for example, `ExternalEmbeddedDevice1`, `ExternalEmbeddedDevice2`, and so on.

Viewing Platform and Device Properties

To access the **Properties** tab in NetBeans IDE, open the **Window** menu, select **IDE Tools**, and then **Properties**. You can now select the platform or device node in the **Device Selector** tab to view the properties of the platform or device in the **Properties** tab.

To view platform or device properties in a separate window, right-click the platform or device node in the **Device Selector** tab and select **Properties**.

To view the platform or device properties in the standalone Device Selector window, select the platform or device node. The properties are displayed in the right pane of the window.

Changing Platform and Device Properties

Properties displayed in gray font or on a gray background cannot be changed. You can adjust properties displayed in black font on a white background. Only MEEP options can be adjusted.

When viewing properties in NetBeans IDE, if you select a property, a short explanation is displayed in the description area below the **Properties** table.

Device properties consist of the following categories:

- **General**

  The following general properties can be changed:

- **Remove IMlet Suite in execution mode**: If this option is selected, the IMlet suite and resources created by the IMlet are removed when you exit the IMlet (assuming the IMlet was started in execution mode).

  You should select the execution mode for a project by right-clicking a project, selecting **Properties**, opening the **Run** category, and selecting the **Regular Execution** option under **Run Method**.

  To run a standalone application (not a NetBeans project) in execution mode, you should use the `-Xdescriptor` option when running the emulator. For example, you can run the `DataCollectionDemo` application on the `Qualcomm_IoE_Device` emulator using the following command:

  ```
  > emulator.exe -Xdevice:Qualcomm_IoE_Device
  -Xdescriptor:DataCollectionDemo.jad
  ```

- **Phone Number**: You can set the phone number to any appropriate sequence, considering country codes, area codes, and so on. If you change this value, the setting will be applied to future instances. The number is a base value for the selected device.
- **Heapsize**: The heap is the memory allocated on a device to store your application objects. Select the maximum heap size from the drop-down list.
- **Memory Limit Per Suite in KB**: This property enables you to define how much memory in kilobytes an IMlet suite is allocated when started. *Unlimited* means the IMlet can use as much memory as is required to run, but not more than the size of the heap specified by the **Heapsize** value.
- **JAM storage size in KB**: The amount of storage space in kilobytes available for applications installed *over the air*.
- **Locale**: The locale as defined in the ME Embedded Profile specification at `https://jcp.org/aboutJava/communityprocess/edr/jsr361/index.html`

- **Monitor**

  Selecting the **Trace GC**, **Trace Class Loading**, **Trace Exceptions**, and **Trace Method Calls** activates tracing for the corresponding device the next time the emulator is started. The trace output is displayed in the Device Log window.

  Note: **Trace Method Calls** returns a lot of messages, which can affect emulator performance.

- **SATSA**

  Security and Trust Services (SATSA) provide the ability to define security settings. You can define the host name of a Java Card emulator, and port numbers for slots 0 and 1.

- **Location Provider #1** and **Location Provider #2**

  The properties in these categories determine the selection of a location provider. Two providers are available so that your application can be tested matching the location provider criteria.

  For more information about these properties, see the Location API at `http://jcp.org/en/jsr/detail?id=179`

- **Landmark Editor**

This category contains only one property (**Max length of input**) that defines the maximum length of input for landmarks.

Viewing Device Information

To view the device information in NetBeans IDE, double-click a device node in the **Device Selector** tab. The **Device Information** tab opens with detailed information about the device.

To view the device information in the standalone Device Selector window, select a device node. Device information is displayed in the right pane of the window along with the properties.

The Device Information window contains details about the device, such as the supported runtime, supported Java ME extensions, supported optional packages, and other capabilities, such as power management or cellular support.

Editing the Security Configuration

The security configuration for a device is set at the time the device is created using the Client Security Model (CSM). For more information about CSM, see the JSR 361: Java ME Embedded Profile at `https://jcp.org/en/jsr/detail?id=361`. For more information about how CSM is implemented for Java ME SDK, see Chapter 14, "Security and IMlet Signing".

To edit the security configuration for a device:

1. Right-click the device node in the Device Selector window and select **Security Configuration**.
2. At the top of the Security Configuration window, specify the custom security providers implementation JAR file, and the class names for the custom authentication and security policy providers. For more information about creating custom providers, see Chapter 15, "Custom Security Policy and Authentication Providers".
3. Select a client in the **Clients** pane and view its permissions and certificates in the panes to the right.
4. To add a client, click **Add** in the **Client** pane, specify the name, and then click **OK**. To remove a client, select it and click **Remove**.
5. To add a permission, click **Add** in the **Permissions** pane, select a permission from the drop-down list, specify a protected resource name, and then click **OK**. To edit or remove a permission, select it and click **Edit** or **Remove**.
6. To add a certificate, click **Add** in the **Certificates** pane, select a certificate from the list, and click **OK**. To remove a certificate, select it and click **Remove**.
7. Click **OK** to apply changes to the security configuration.

Using the Custom Device Editor

The Custom Device Editor can be used to create custom devices. The appearance of a custom device is generic, but the functionality can be configured according to your specifications.

To run the Custom Device Editor in NetBeans IDE, open the **Tools** menu, select **Java ME**, and then **Custom Device Editor**.

To run the Custom Device Editor in a separate window, run the `device-editor.exe` file under `bin` in the Java ME SDK installation directory. For example, you can use the following command:

```
C:\> Java_ME_platform_SDK_8.0\bin\device-editor.exe
```

Creating a Custom Device

To create a custom device:

1. In the Custom Device Editor, select a platform (for example, MEEP) and click **New**.
2. Specify a name and description of the device.
3. Ensure that the **Device Configuration** and **Device Profile** match the specifications for your new device.
4. Select optional packages that provide additional functionality for your device, corresponding to the required configuration.
5. Define properties for the interfaces and protocols supported by your custom device under the corresponding tabs.
6. Click **OK** to create and add the device to the custom device tree.

The created device appears in the Device Selector, and the device definition is saved under `\toolkit-lib\devices` in the Java ME SDK installation directory. You can run projects from the NetBeans IDE or from the command line on this device.

Setting Custom Device Properties

When you create a new embedded device using the Custom Device Editor, you can use the default implementation or create your own custom implementation for the interfaces discussed in this section.

You can set device properties when you create the custom device, or define device properties later, by selecting your custom device in the Custom Device Editor and clicking **Edit**.

This section briefly describes the tabs in the custom device properties. For more information about custom device protocols and property settings, see the corresponding help topic by pressing **F1** with the necessary tab open.

For each custom device, you can configure properties in the following tabs:

- **GPIO**: The General Purpose Input/Output (GPIO) ports are groupings of GPIO pins that can be configured for output, input, or bidirectional.
- **I2C** and **SPI**: The Inter-Integrated Circuit (I2C) and the Serial Peripheral Interface (SPI) tabs are similar. Each one can be used to simulate a simple peripheral slave device that echoes back data sent to it.
- **MMIO**: The Memory-Mapped Input/Output (MMIO) tab can be used to emulate the MMIO interface bus. It facilitates I/O between the CPU, memory, and peripheral devices.
- **ADC**: The Analog-to-Digital Converter (ADC) tab can be used to set up the conversion of an input analog stream to a sequence of digital numbers.
- **DAC**: The Digital-to-Analog Converter (DAC) tab can be used to set up the conversion of a sequence of digital numbers to an analog stream.

- **PWM**: The Pulse Width Modulation (PWM) tab can be used to set up the conformance of a digital signal's width based on its duration.
- **Pulse Counters**: The Pulse Counters tab can be used to set up pulse counters that track the number of pulses sent to a device.
- **Line-Oriented Displays**: The Line-Oriented Displays tab can be used to configure a line-oriented display that provides a simple user interface in emulation.
- **Headless Input Devices**: The Headless Input Devices tab can be used to configure input devices not connected to a monitor, such as buttons.

Managing Custom Devices

Custom devices should be managed using the Custom Device Editor. Using the tool ensures that your device can be detected and integrated with Oracle Java ME SDK 8.

To clone a device, select it and click **Clone**. Provide a unique name to prevent confusion.

To remove a device, select it and click **Remove**. This completely deletes the device.

To save the device configuration as a ZIP file, select it and click **Export**. Specify the path and name in the file system explorer.

To load a device configuration from a previously exported ZIP file, click **Import** and select the file in the file system explorer.

Making Device Connections

The configuration of all peripherals, except Universal Asynchronous Receiver/Transmitter (UART), ATCommands devices, and WatchDog timers, can be inspected in the emulator main window. The configuration of UART is defined by the hardware configuration of the COM ports on your PC.

For more information about connecting to a UART Device, see Connecting to a UART Device.

To open a serial port, such as COM1 or COM2, in Windows, use the Device Connections Manager. You can use the following code in the application:

```
Connector.open("comm:COM1")
```

Connecting to a UART Device

To use the UART functionality, you must configure the `daapi_config.json` file located under `runtimes\meep\lib` in the Java ME SDK installation directory. For example, the default location is:

```
C:\Java_ME_platform_SDK_8.0\runtimes\meep\lib\daapi_config.json
```

The configuration of the hardware device name for the UART ports in Windows is made up of two settings:

- The `deviceNumber` property of `UARTDeviceConfiguration`
- The system property of `device.uart.prefix`

For example, if `UARTDeviceConfiguration.deviceNumber = 1` and `device.uart.prefix = COM`, then the hardware device name is `COM1`.

For an already opened UART, you can add it to the peripheral manager by calling `DeviceManager.open` with the configuration object.

Additional Peripherals

The `EmbeddedDevice1` emulator provides support for additional peripherals: ATDevice and Watchdog timers. Use the following configuration settings to open them using the device manager:

- There are two Watchdog timers on the `EmbeddedDevice1` emulator with the following configurations:
  - **Device Name**: WDG, ID: 30, Hardware Timer's Number: 1
    (This is a regular watchdog timer)
  - **Device Name**: WWDG, ID: 31, Hardware Timer's Number: 2
    (This is a windowed watchdog timer)

  Watchdog timers provide the services to track how your application functions. Because no hardware is available on Windows, all running applications are stopped when the Watchdog event is started.
- ATDevice is a simple AT commands-based device on the `EmbeddedDevice1` emulator that responds with `OK` to any command. It has the following configuration:
  - **Device Name**: EMUL, ID: 13, Controller Number: 1, Hardware Channel's Number: 1

Part III

NetBeans IDE

Part III provides an introduction to more specialized developer operations, such as tooling, monitoring, and debugging, and information about file structure, logging, properties, and projects.

Part III contains the following chapters:

6

Creating Projects

A project is a group of files comprising a single application, including source files, resource files, XML configuration files, automatically generated Apache Ant build files, and a properties file.

When a project is created, the Oracle Java ME SDK 8 performs the following tasks:

- Creates a source tree
- Sets the emulator platform for the project
- Sets the project run and compile-time classpaths
- Creates a build script that contains actions for running, compiling, debugging, and building Javadoc

The Oracle Java ME SDK 8 and NetBeans IDE 8.0 create their project infrastructure directly on top of Apache Ant. An Oracle Java ME SDK 8 project can be opened and edited in NetBeans IDE 8.0. With the Ant infrastructure in place, you can build and run your projects within the Oracle Java ME SDK 8 or from the command line.

NetBeans IDE provides two views of the project:

- The **Projects** tab provides a logical view of the project.
- The **Files** tab displays a physical view of the project.

Project settings are controlled in the project Properties window. To open the Properties window, right-click on an item or subitem in the project tree and select **Properties**.

Creating a Java ME Project

The project provides a basic infrastructure for development. You provide source files, resource files, and project settings as needed. Most project properties can be edited later.

To create a Java ME project:

1. Open the **File** menu and select **New Project**.
2. In the Choose Project window, select the **Java ME Embedded** category, and the **Java ME Embedded Application** project type. Click **Next**.
3. In the Name and Location window, specify a project name and any other settings. The default values are usually fine. Click **Finish**.

Create a New IMlet

By default, the new project contains an IMlet class that extends the `javax.microedition.midlet.MIDlet` class. This class can be used as the main IMlet. To add another IMlet to the project:

1. Right-click the project, select **New**, and then **MIDlet**.
2. Specify the name of the IMlet, its location within the selected project, and other settings. Click **Finish**.

When the new IMlet is created, the Oracle Java ME SDK 8 automatically adds it to the project's Application Descriptor File.

Debugging Java ME Projects

Oracle Java ME SDK 8 projects use standard NetBeans IDE 8.0 debugging utilities. See the NetBeans help topic, *Debugging Tasks: Quick Reference*. This topic includes links to a variety of debugging procedures.

If you have an external device that runs a supported runtime you can perform on-device debugging. The device must be detected by the Device Manager and be present on the Device Selector, as described in Chapter 5, "Working with Devices."

7

Viewing and Editing Project Properties

All projects have properties. Some properties, such as the project's name and location cannot be changed, but other properties can be edited.

To view or edit a project's properties, right-click the project node and select **Properties**. In the Project Properties window, you can view and customize the project properties.

Configuring Project Sources

Use the **Sources** category in the Project Properties window to view the project folder location and configure a list of source folders with the corresponding labels used in the **Projects** view.

You can add and remove folders to the list of sources, and order the list.

Below the list, use the drop-down lists to select the source format and encoding.

Selecting the Platform for the Project

The Java ME platform emulates the execution of an application on one or more target devices. Use the **Platform** category in the Project Properties window to select and configure the platform used in your project.

Ensure that **Oracle Java Platform Micro Edition SDK 8.0** is selected as the platform.

By default, the devices in the device menu are also suitable for the platform type and emulator platform. The device you select is the default device for this project. It is used whenever you run the project. The selected device influences the device's **Configuration** and **Profile** options, and the available optional packages.

Configuring Project Libraries

Use the **Libraries** category in the Project Properties window to configure the list of libraries used for compiling, running, and testing the project.

You can add, edit, and remove projects, libraries, JAR files, and folders to the lists in corresponding tabs. You an also reorder the lists as required.

Configuring Application Descriptor Attributes

Use the **Application Descriptor** category in the Project Properties window to configure the project attributes.

Under each tab, you can add, edit, remove, and reorder the general attributes for JAR manifests, MIDlets in the suite, the push registry, and permissions requested by the API.

Note: Do not begin user-defined attribute keys with `MIDlet-` or `MicroEdition-`.

To use the push registry, you must configure the permission to access the Push Registry API (`javax.microedition.io.PushRegistry`) in the **API Permissions** tab.

Configuring the Build Process

When you build a project, the Oracle Java ME SDK 8 compiles the source files and generates the packaged build output (a JAR file) for your project. You can build the main project and all of its required projects, or build any project individually.

In general, you do not need to build the project or compile individual classes to run the project. Use the **Build** category and its subcategories in the Project Properties window to configure the following build tasks:

- **Compiling**: Use this category to define how your project is compiled.
- **Signing**: Use this category to enable signing and assign key pairs to the project.
- **Obfuscating**: Use this category to install an obfuscator library and set the level of obfuscation for project files.
- **Documenting**: Use this category to define how your project is documented.

Configuring Project Running Properties

Use the **Run** category in the Project Properties window to set up the configuration for running the project.

You can set command-line options, the debugger timeout, and the run method.

Building a Project from the Command Line

In NetBeans IDE, you click one button to build and run the project. Behind the scenes, however, there are two steps. First, Java source files are compiled into Java class files. Next, the class files are *preverified*, which means they are prepared for the CLDC virtual machine.

To build the project manually from the command line, do the following:

1. Run the `jar` command to verify that it is in your `PATH` environment variable.

   Check the version of the JDK by running `java -version` on the command line.

2. Use the `javac` compiler from the JDK to compile the source files. You can use the existing Oracle Java ME SDK 8 project directory structure. Use the `-bootclasspath` option to tell the compiler to use the MEEP APIs, and use the `-d` option to tell the compiler where to put the compiled class files.

   The following example demonstrates how you might compile an application, taking source files from the `src` directory and placing the class files in the `tmpclasses` directory:

```
C:\> javac -bootclasspath ..\..\lib\cldc_1.8.jar;..\..\lib\meep_8.0.jar
     -d tmpclasses
     src\*.java
```

For more information about `javac`, see the *JDK Command Reference*.

Packaging an IMlet Suite (JAR and JAD)

To package an IMlet suite manually, you must create a manifest file, an application JAR file, and an IMlet descriptor (also known as a Java Application Descriptor or JAD).

Create a manifest file that contains the appropriate attributes as specified in the MEEP specification. You can use any text editor to create the manifest file. For example, a manifest might have the following contents:

```
MIDlet-1: My MIDlet, MyMIDlet.png, MyMIDlet
MIDlet-Name: MyMIDlet
MIDlet-Vendor: My Organization
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.8
MicroEdition-Profile: MEEP 8.0
```

Create a JAR file that contains the manifest and the suite's classes and resource files. To create the JAR file, use the `jar` tool that comes with the Java SE software development kit. The syntax is as follows:

`jar cfm` *`file manifest`* `-C` *`class-directory`* `. -C` *`resource-directory`* `.`

The arguments are as follows:

- *file*: JAR file to create
- *manifest*: Manifest file for the MIDlets
- *class-directory*: Directory containing the application's classes
- *resource-directory*: Directory containing the application's resources

For example, to create a JAR file named `MyApp.jar` whose classes are in the `classes` directory and resources are in the `res` directory, use the following command:

```
jar cfm MyApp.jar MANIFEST.MF -C classes . -C res .
```

Create a JAD file that contains the appropriate attributes as specified in the MEEP specification. You can use any text editor to create the JAD file. This file must have the extension `.jad`.

> **Note:** You must set the `MIDlet-Jar-Size` entry to the size in bytes of the JAR file created in the previous step.

For example, a JAD file might have the following contents:

```
MIDlet-Name: MyIMlet
MIDlet-Vendor: My Organization
MIDlet-Version: 1.0
MIDlet-Jar-URL: MyApp.jar
MIDlet-Jar-Size: 24601
```

8

Finding Files in the Multiple User Environment

All users with an account on the host machine can access Oracle Java ME SDK 8. This feature is called Multiple User Environment (MUE).

> **Note:** A MUE supports access from several accounts. It does not support multiple users accessing Oracle Java ME SDK 8 simultaneously. For more information, see Switching Users.

MUE in Oracle Java ME SDK 8 requires an installation directory to be used as a source for copying. Each user's personal files are located in the user's folder under `C:\Users`. This document uses the variable *username* to refer to the user's personal directory. The personal Java ME SDK configuration files are maintained in a subdirectory named `javame-sdk` that has subdirectories for each version installed. For example, the Java ME SDK 8 configuration files for a user named `johns` are located under `C:\Users\johns\javame-sdk\8.0`.

For information about log files, see Chapter 9, "Logs."

Switching Users

Multiple users cannot run Oracle Java ME SDK 8 simultaneously, but you can run Oracle Java ME SDK 8 from different user accounts on the host machine. When you switch users, you must close Oracle Java ME SDK 8 and exit the Device Manager. A different user can then start Oracle Java ME SDK 8 as the owner of all processes.

Installation Directories

The Oracle Java ME SDK 8 directory structure conforms to the Unified Emulator Interface (UEI) Specification (`http://www.oracle.com/technetwork/java/javame/documentation/ueispecs-187994.pdf`), version 1.0.2. This structure is recognized by all IDEs and other tools that work with the UEI.

By default, Oracle Java ME SDK 8 is installed under `C:\Java_ME_platform_SDK_8.0`. The installation directory has the following structure:

- `bin`: Contains the following command-line tools:
  - `cref`: The Java Card Platform Simulator tool, which is used to simulate smart cards in the emulator. It is used for testing SATSA (JSR 177) applications with the Oracle Java ME SDK 8. For more information about SATSA, see

Chapter 20, "JSR 177: Smart Card Security (SATSA)".

  - `device-editor`: Tool for creating new custom devices.
  - `device-manager`: The Device Manager is a component that must be running when you work with Oracle Java ME SDK 8. After installation, it starts as a service, and it automatically restarts every time your computer restarts.
  - `device-selector`: Tool for viewing and editing device and platform properties and information. It can also be used to run applications on a specific device.
  - `emulator`: UEI-compliant emulator. For more information, see Chapter 3, "Using the Emulators."
  - `jadtool`: Tool for signing IMlets. For more information, see Chapter 14, "Security and IMlet Signing."
  - `mekeytool`: Tool for managing Java ME keystores. For more information, see Chapter 14, "Security and IMlet Signing."
  - `update-center`: The Oracle Java ME SDK 8 Update Center.
  - `wma-tool`: A command-line tool for sending and receiving SMS and CBS messages. For more information, see "Running the WMA Tool."
  - `wscompile`: Tool for compiling stubs and skeletons for JSR 172.
- `docs`: Release documentation.
- `legal`: License and copyright files.
- `lib`: JSR JAR files for compilation.
- `runtimes`: MEEP runtime files.
- `toolkit-lib`: Oracle Java ME SDK 8 files for configuration and definition of devices and UI elements. Executables and configuration files for the Device Manager and other Oracle Java ME SDK 8 services and utilities.

NetBeans IDE 8.0 User Directories

These are the default NetBeans IDE 8.0 user directories:

- NetBeans IDE 8.0 default project location:

  username`\My Documents\NetBeansProjects`

- To see the NetBeans IDE 8.0 user directory, click the **Help** menu and select **About** in the main window. The default location:

  username`\.netbeans\8.0`

Oracle Java ME SDK 8 User Directories

The `javame-sdk` directory contains device instances and session information. If you delete this directory, it is re-created automatically when the Device Manager is restarted. It contains subdirectories for each available release. For example, the directory for Oracle Java ME SDK 8 is:

username`\javame-sdk\8.0`

Device working directories are located under `work`, for example:

username`\javame-sdk\8.0\work\EmbeddedDevice1`

Any detected external devices are also added to this directory.

9

Logs

Each Java ME SDK tool (Device Manager, Device Selector, Custom Device Editor, and so on) saves its log to the user configuration directory under `log`. The default location is `userdir\javame-sdk\8.0\log\`

Each device (or emulator) instance writes its own log to the `device.log` file in the device's configuration directory. For example, the EmbeddedDevice1 log is located in `userdir\javame-sdk\8.0\work\EmbeddedDevice1\device.log`

10 Profiling Applications

Oracle Java ME SDK 8 supports performance profiling for Java ME applications. The profiler keeps track of every method in your application. For a particular emulation session, the profiler figures out how much time was spent in each method.

Oracle Java ME SDK 8 supports offline profiling. Data is collected during the emulation session. After you close the emulator, you can export the data to an NPS file you can load and view later. As you view the snapshot you, can investigate particular methods, classes, and packages, and save a customized snapshot (a PNG file) for future reference.

You can start a profiling session from NetBeans IDE 8.0, as described in Collecting and Saving Profiler Data in the IDE or from the command line. It is important to understand that profiling data produced from the command line has a different format (PROF) than data produced from the NetBeans IDE 8.0 profiler (NPS).

> **Note:** This feature might slow down the execution of your application.

Profiling data from Oracle Java ME SDK 8 projects is displayed in NetBeans IDE on a tab labeled **cpu** with the name of the file or the time the data was displayed. The **cpu** tab opens when a profiled application stops. Because only performance profiling is supported, the Profiler window has limited usefulness for MEEP applications.

Collecting and Saving Profiler Data in the IDE

This procedure describes interactive profiling.

> **Note:** The profiler maintains a large amount of data, so profiled IMlets place greater demands on the heap. To increase the `Heapsize` property, see Chapter 5, "Working with Devices."

To run the profiler:

1. In the Projects widow, right-click the project you want to profile and select **Profile**.

   If this is the first time profiling this project, you are prompted to integrate the profiler. Click **Yes** to perform the integration.

2. Select **CPU Profiler** and click **Run**.

3. Profiling data is displayed in a tab labeled **cpu** with the time the data was displayed.

4. To export the profile data to an NPS file, click the **Export to** button and specify the file name and location. This data can be reloaded at a later time. See Loading an NPS File.
5. To save the current view to a PNG file, click the **Save current view to image** button and specify a file name and location.

Loading an NPS File

To retrieve profile data from a previously exported NPS file (see Collecting and Saving Profiler Data in the IDE):

1. Open the **Profile** menu and select **Load Snapshot**.
2. Select the NPS file.

The Profiler opens in its own tab labeled **cpu** with the name of the file.

Note: The profiling values obtained from the emulator do not reflect the actual values on an external device.

Importing PROF File

A PROF file created from the command line can be loaded in NetBeans IDE 8.0. The following example shows what a profiling session command might look like:

```
emulator.exe -Xdevice:EmbeddedDevice1
-Xdescriptor:"C:\Documents and Settings\user\My Documents\NetBeansProjects\SomeDem
o\dist\Games.jad" -Xprofile:file=C:\temp\SomeDemo.prof
```

Files created from the command line are formatted differently from the NPS files created as described in Collecting and Saving Profiler Data in the IDE

To retrieve command-line profile data in NetBeans:

1. Open the **File** menu and select **Open File**.
2. Select the PROF file and click **Open**.

   The Profiler displays the data as a text file.

11

Network Monitoring

The network monitor provides a convenient way to see the information your application is sending and receiving on the network. This is helpful if you are debugging network interactions or looking for ways to optimize network traffic.

MEEP applications can connect through HTTP and other protocols.

Monitoring Network Traffic

To activate network activity monitoring for an application:

1. In the Projects window, right-click a project and select **Profile**.
2. If this is the first time profiling this application, you are prompted to integrate the profile with the project. Click **Yes** to perform the integration.
3. In the Profile window, select **Network Monitor**, and click **Run**.
4. Start your application.

When the application makes a network connection, information about the connection is captured and displayed in the **Network Monitor** tab.

The top frame displays a list of connections. Click a connection to display its details in the bottom frame.

Under **Hex View**, raw connection data is shown as raw hexadecimal values with the equivalent text.

Filtering and Sorting Messages

Filters are used to examine a subset of the total network traffic.

- In the **Select Devices** list, select only the devices you want to view.
- In the **Select Protocols** list, select only the protocols you want to view. The protocols listed reflect what is installed on the device.
- In the **URL Filter** field, you can specify the URL for which you want to view connection data.

Click on a table header to sort the message data:

- **No.**: Connections are sorted by phone number.
- **Protocol**: Connections are sorted by protocol name.
- **Device**: Connections are sorted by device name.
- **URL**: Connections are sorted by URL.

- **Time**: Connections are sorted in chronological order.
- **Size**: Connections are sorted by the size of data.
- **Active**: Connections are sorted based on whether it is currently active.

Saving and Loading Network Monitor Information

To save your network monitor session, click the **Save Snapshot** button in the Network Monitor toolbar. Select a file name. The default file extension is NMD (network monitor data).

To load a network monitor session:

1. Open the **File** menu and select **Open File**.
2. Browse to find the necessary NMD file.

Note: To avoid memory issues, the Hex view display is currently limited to 16 kilobytes of data.

Searching the Connection Data

To specify a string that you want to find in the available connection data, click the button with the magnifying glass labeled **Find in results** (Ctrl + F) above the table in the Network Monitor toolbar.

You can search the records up to the specified string or back to a previous string occurrence by clicking the **Find Next Occurrence** (F3) or **Find Previous Occurrence** (Shift+F3) buttons in the toolbar.

Clearing the Connection List

To remove all inactive protocol records from the network monitor, click the **Clear** button (the broom icon in the right part of the Network Monitor toolbar).

12

Memory Monitoring and Runtime Tracing

This chapter describes how to use tracing and the Memory Monitor to examine an application's memory use on a particular device.

Activating tracing for a particular device enables you to see low-level information as an application runs.

The Memory Monitor shows memory use as an application runs. It displays a dynamic detailed listing of the memory usage per object in table form, and a graphical representation of the memory use over time. You can take a snapshot of the memory monitor data. Snapshots can be loaded and examined later.

> **Note:** The memory usage that you observe with the emulator is not the same as the memory usage on an external device. Remember, the emulator does not represent an external device. It is one possible implementation of its supported APIs.

Enabling Tracing

To enable tracing:

1. In the Device Selector window, right-click a device and select **Properties**.
2. In the device's Properties window, expand the **Monitor** node and select the desired trace options:
   - **Trace GC**: Monitoring garbage collection can help you determine object health. The garbage collector cannot delete objects that do not have a null reference. Null objects will be garbage collected and not reported as active.
   - **Trace Class Loading**: Observing class initialization and loading is useful for determining dependencies among classes.
   - **Trace Exceptions**: Displaying exceptions caught by the Java ME SDK can be used for debugging.
   - **Trace Method Calls**: Reporting called and returned methods is useful to understand the operational sequence of the application. The output for this option is very verbose, and it can affect performance.
3. (Optional) Verbose tracing output might cause you to run out of memory on the device before the application is fully tested. To increase the device memory, right-click a device and select **Properties**, expand the **General** node, and then specify the value for the **Heapsize** option.

To display tracing data, in the emulator, open the **View** menu and select **Device Log** when an application is running. Each device (or emulator) instance writes its own log

to the `device.log` file in the device's configuration directory. For example, the EmbeddedDevice1 log is located in
`userdir\javame-sdk\8.0\work\EmbeddedDevice1\device.log`

Using the Memory Monitor

To examine an application's memory usage:

1. In the Projects view, right-click the project and select **Profile**.

   If the profiler is not yet integrated, you are prompted to enable profiling for the project. Click **Yes** to continue.

2. Select **Memory Monitor**, and click **Run**.

   The **Memory Monitor** tab opens in the main working area of NetBeans IDE.

The top part of the **Memory Monitor** tab contains a graph, while the bottom part of the tab contains an object table. To the left of the graph is the current memory usage in bytes. The green line plots these values. The red line is the maximum amount of memory used since program execution, corresponding to the maximum size in bytes on the left.

Beneath the object table you see counters that display the total number of objects, the amount of memory used, the amount of free memory, and the total amount of memory on the device.

Click a column header in the object table to sort the data. Sorting is case sensitive.

Click a row to display the call stack tree in a window to the right of the table.

Click a folder to browse the call stack tree to see the methods that created the object.

To find a particular method in the call stack tree, click **Find** and enter a search string.

Click **Refresh** to update the call stack tree as data is gathered. It is not refreshed automatically.

Because the data changes rapidly, it is convenient to take several snapshots of the memory monitor data and review them later. Open the **File** menu and select **Save As**, then specify an MMS (memory monitor snapshot) file name and location for the monitor data snapshot.

When you exit an IMlet, object table data is cleared, but graph data is not cleared. The graph data you see is cumulative for this emulator session. The Memory Monitor plots session data for any IMlet run on the current emulator until you exit the application and close the emulator.

Viewing a Session Snapshot

To load a memory monitor snapshot.

1. Open the **File** menu and select **Open File**.
2. Browse to find the MMS file you saved.

   The Memory Monitor opens in its own tab in the main window. Note that the tab displays the time when the snapshot was saved.

13

Application Debugging

This chapter discusses the debugging process for a MEEP application.

Before starting this procedure, ensure that your environment is properly configured:

- Ensure that the device is connected and that it appears in the Device Selector window.
- Right-click the project and select **Properties**. In the Properties window, select the **Platform** category and set the device currently available.

To debug a project, right-click the project name and select **Debug**.

Part IV

Security

Part IV provides information about the security model in Oracle Java ME SDK 8.

Part IV contains the following chapters:

- Chapter 14, "Security and IMlet Signing"
- Chapter 15, "Custom Security Policy and Authentication Providers"

14

Security and IMlet Signing

This chapter describes how the security architecture is organized in Oracle Java ME SDK 8.

Applications are installed, run, closed, and restarted according to the IMlet life cycle described in the *Java ME Embedded Profile* specification. You can find the specification in the `meep-8.0.zip` file located under `docs\api` in the Java ME SDK installation directory. The default location is `C:\Java_ME_platform_SDK_8.0\docs\api\meep-8.0.zip`

In particular, the following chapters in the specification are the most relevant for understanding the security model:

- Security for Applications
- Security Authentication Providers
- Security Policy Providers

The following is the general process for creating a cryptographically signed IMlet suite:

1. The IMlet author, probably a software company, buys a signing key pair from a certificate authority (CA).
2. The author signs the IMlet suite with the signing key pair and distributes the company's certificate with the IMlet suite.
3. When the IMlet suite is installed on the emulator or on a device, the implementation verifies the author's certificate using its own copy of the CA's root certificate. Then the implementation uses the author's certificate to verify the signature on the IMlet suite.
4. After verification, the device or emulator assigns the IMlet suite to one of the clients defined by the security policy. The default authentication scheme (X.509-based certificate) uses the certificate DN to determine to which client an application must be bound.

Security Policy Provider Clients

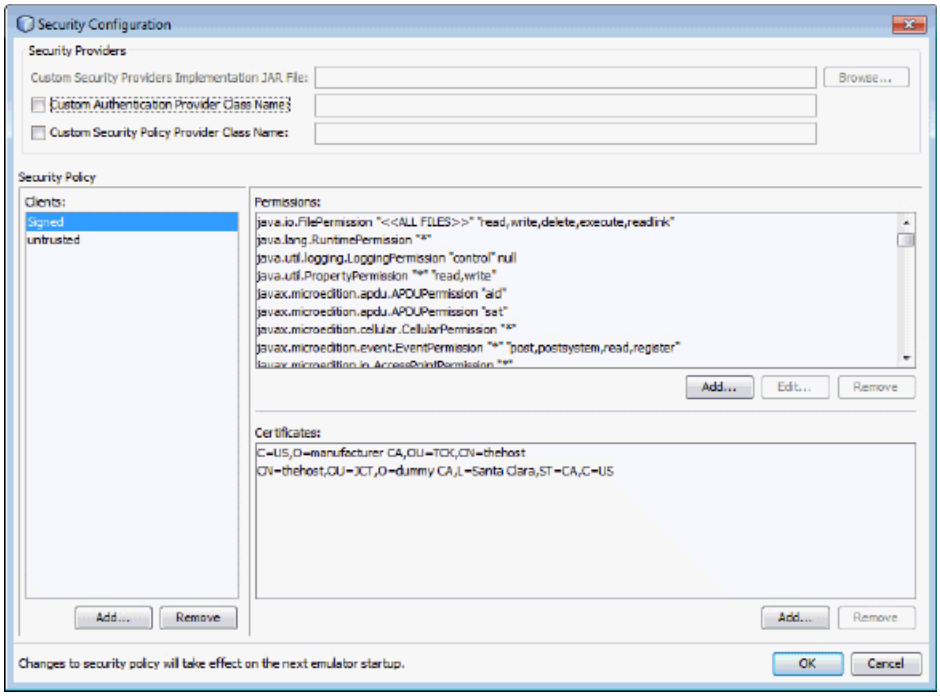
Oracle Java ME SDK 8 supports the following clients by default:

- `Signed`: An example client with all permissions granted.
- `untrusted`: Defines the security policy for untrusted applications. According to the X.509 authentication scheme, unsigned applications are bound to the untrusted client.

Configuring the Security Policy

To configure the security policy for a device, right-click the device in the Device Selector and select **Security Configuration**. Figure 14–1 shows the Security Configuration window.

Figure 14–1 The Security Configuration Window



The options in the **Security Providers** group at the top of the Security Configuration window can be used if you want to specify a custom security provider implementation JAR file, and class names of your custom authentication provider and security policy provider. For information about creating custom providers, see Chapter 15, "Custom Security Policy and Authentication Providers".

To add a client, click **Add** under the **Clients** list, specify a name and click **OK**. To remove a client, select it in the list and click **Remove**.

When you select a client from the list, you can add, edit, and remove permissions and certificates for the selected client.

To add a permission, select the necessary client, and click **Add** under the Permissions list. Then select the permission from the list, specify the name of the protected resource (you can use wildcards) and the requested actions separated by commas (for example, `read,write`), and click **OK**. To edit a permission, select it from the list of permissions, and click **Edit** under the Permissions list. To remove a permission, select it in the list, and click **Remove**.

To add a certificate, select the necessary client, and click **Add** under the Certificates list. Then select the certificate from the list of available certificates and click **OK**. To remove a certificate, select it in the list, and click **Remove**.

Signing a Project

Devices use signing information to verify an application's source and validity before allowing it to access protected APIs.

Oracle Java ME SDK 8 provides a default built-in keystore, but you can also create any number of key pairs using the Keystores Manager as described in Managing Keystores and Key Pairs.

The key pair consists of the following keys:

- A private key that is used to create a digital signature.
- A public key that anyone can use to verify the authenticity of the digital signature.

To sign a project with a key pair:

1. Right-click a project and select **Properties**.
2. In the **Signing** category, select **Sign JAR**.
3. Select an existing keystore or click **Open Keystores Manager** to create another keystore. For information about managing keystores, see Managing Keystores and Key Pairs.
4. Select a key pair alias.

   A keystore might be accessed by several key pairs, each with a different alias. If you prefer to use a unique key pair, click **Open Keystores Manager** and create a new key pair.

   The **Certificate Details** area displays the subject, issuer, and validity dates for the selected keystore.
5. Click **OK**.

It is also necessary to export the certificate to the device. For more information, see Managing Root Certificates.

Managing Keystores and Key Pairs

For test purposes, you can create a signing key pair to sign an IMlet. The Keystores Manager administers this task. The keystores known to the Keystores Manager are listed when you sign a project.

To deploy an IMlet on a device, you must obtain a signing key pair from a certificate authority recognized by the device. You can also import keys from an existing Java SE platform keystore.

To create a keystore:

1. Open the **Tools** menu and select **Keystore Management**.
2. Click **Add Keystore**.
3. Select **Create a New Keystore** and specify a name, location, and password.
4. Click **OK**.

To add an existing keystore:

1. Open the **Tools** menu and select **Keystore Management**.
2. Click **Add Keystore**.
3. Select **Add Existing Keystore** and specify the path to the keystore file. The default location for user-defined keystores is the user's folder under `C:\Users`.

4. Click **OK**.

   You might have to unlock this keystore and each key pair within it.

To create a new key pair:

1. Open the **Tools** menu and select **Keystore Management**.
2. Select a keystore.

   Note: You cannot create key pairs in the default built-in keystore.

3. Click **New**.
4. Specify an alias used to refer to this key pair and at least one field under **Certificate Details**. Optionally, you can also provide a password.
5. Click **OK**.

To remove a key pair, select it in the list and click **Delete**.

Managing Root Certificates

The Oracle Java ME SDK 8 command-line tools manage the emulator's list of root certificates.

External devices have similar lists of root certificates. When you deploy your application on an external device, you must use signing keys issued by a certificate authority whose root certificate is on the device. This makes it possible for the device to verify your application.

Each emulator instance has its own keystore. The keystore file is named `_main.ks` and located under `appdb\certs` in the device's configuration directory. For example, the default keystore for EmbeddedDevice1 is `userdir\javame-sdk\8.0\work\EmbeddedDevice1\appdb\certs\_main.ks`

You can use the `-import` option to import certificates from these keystores as described in Manage Certificates (mekeytool).

To export a certificate to an emulated device:

1. Open the **Tools** menu and select **Keystore Management**.
2. Select a keystore, and then select a key.
3. Click **Export**.
4. Select and emulator and a certificate, and click **Export**.

   Note: Before exporting, you can modify the list of registered keys by selecting any key and clicking **Delete Key** to delete it from the list.

5. Click **Close** when you are done.

Command-Line Security Features

The full spectrum of the Oracle Java ME SDK 8 security features are also available from the command line. You can adjust the emulator's default protection domain, sign IMlet suites, and manage certificates.

Sign IMlet Suites (jadtool)

`jadtool` is a command-line interface for signing IMlet suites using public key cryptography according to the MEEP specification. Signing an IMlet suite is the process of adding the signer certificates and the digital signature of the JAR file to a JAD file. `jadtool` is also capable of signing payment update (JPP) files.

`jadtool` only uses certificates and keys from Java ME platform keystores. Java SE software provides `keytool`, the command-line tool to manage Java SE platform keystores.

`jadtool.exe` is located under `bin` in the Java ME SDK installation directory.

The following options can be used with the `jadtool` command:

-help
Prints usage instructions for `jadtool`.

-addcert
Adds the certificate of the key pair from the given keystore to the JAD file or JPP file. This option has the following syntax:

-addcert -alias <key_alias> [-storepass <password>] [-keystore <keystore>] [-certnum <number>] [-chainnum <number>] [-encoding <encoding>] -inputjad <filename> -outputjad <filename>

-addjarsig
Adds a digital signature of the input JPP file to the specified output JPP file. This option has the following syntax:

-addjarsig [-jarfile <filename>] -keypass <password> -alias <key_alias> -storepass <password> [-keystore <keystore>] [-chainnum <number>] [-encoding <encoding>] -inputjad <filename> -outputjad <filename>

-showcert
Displays information about certificates in JAD files. This option has the following syntax:

-showcert [[-certnum <number>] [-chainnum <number>] | [-all]] [-encoding <encoding>] -inputjad <filename>

Manage Certificates (mekeytool)

`mekeytool` manages the public keys of certificate authorities (CAs). It is functionally similar to the `keytool` utility that comes with the Java SE Development Kit (JDK). The purpose of the public keys is to facilitate secure HTTP communication over SSL (HTTPS).

Before using `mekeytool`, you must have access to a Java Cryptography Extension keystore. You can create one using the Java SE `keytool` utility (found in the `bin` directory under the JDK installation location).

Oracle Java ME SDK 8 provides a default Java ME keystore, which is located in the Java ME SDK installation directory under `runtimes\cldc-hi\appdb\certs`. This keystore contains an index file named `_main.ks` and a set of certificate files.

Each emulator instance has its own keystore located in the device folder, for example: `userdir\javame-sdk\8.0\work\EmbeddedDevice1\appdb\certs`. If you do not specify a value for `-keystore`, the default keystore is used.

The -Xdevice option can be used with any command to run it on the specified device. Note that not every device supports all of the mekeytool commands. Specify the device name after a colon. For example, to list the keys in the keystore of EmbeddedDevice1, run the following command:

```
> mekeytool.exe -Xdevice:EmbeddedDevice1 -list
```

The following commands can be used with the `mekeytool` utility:

-help
Prints usage instructions for `mekeytool`.

-import
Imports a public key from the source keystore to the device's keystore. This command has the following syntax:

-import [-keystore <filename>] [-storepass <password>] [-keypass <password>] [-alias <key_alias>]

Option	Description	Default
`-keystore`	Path to the JCA keystore file or file that contains the certificate	`%HOME%\.keystore.ks`
`-storepass`	Password to unlock the input JCA keystore	N/A
`-keypass`	Private key password for the JKS or PKCS12 keystore	N/A
`-alias`	The key pair alias in the input JCA keystore	N/A

-list
Lists the keys in the Java ME keystore, including the owner and validity period for each.

-delete
Deletes a key from the given Java ME keystore with the given owner. This command has the following syntax:

-delete {-owner <owner> | -number <number>}

Option	Description	Default
-number	The key number in the keystore. Keys are numbered starting from 1. To view the key number, use the -list option.	N/A
-owner	The key owner.	N/A

-export

Exports the key from the keystore. This command has the following syntax:

-export -number <number> -out <filename>

Option	Description	Default
-number	The key number in the keystore. Keys are numbered starting from 1. To view the key number, use the -list option.	N/A
-out	Name of the output file.	N/A

15

Custom Security Policy and Authentication Providers

This chapter describes how you can create custom security policy and authentication providers, as defined in the MEEP specification. Oracle Java ME SDK 8 is bundled with default providers that can be used without any modification or configured to your needs, as described in Configuring the Security Policy.

The classes necessary to create custom security policy and authentication providers are defined in the `com.oracle.meep.security` package. You can find a detailed Javadoc of this package in the `security_api_javadoc.zip` file located under `docs\api` in the Java ME SDK installation directory. The default location is `C:\Java_ME_platform_SDK_8.0\docs\api\security_api_javadoc.zip`

This chapter contains the following sections:

- Section 15.1, "Creating a Security Policy Provider"
- Section 15.2, "Creating an Authentication Provider"
- Section 15.3, "Installing Custom Providers"

15.1 Creating a Security Policy Provider

The purpose of a security policy provider is to define the list of clients and their *protection domains*. A protection domain of a client is a set of permissions that can be granted to the application bound to this client.

A custom security policy provider must extend the `Policy` class and implement the `Policy.initialize()` abstract method. This method is called by the security framework and is responsible for security policy initialization. During initialization, the custom security policy provider must use the `Policy.addClient(com.oracle.meep.security.Client)` helper method to create the list of clients.

Example 15–1 shows how to create a custom security policy provider that defines two clients with different protection domains and specifies a separate protection domain for the virtual `untrusted` client.

Example 15–1 Custom Security Policy Provider

```
package com.company.security;

import com.oracle.meep.security.Client;
import com.oracle.meep.security.Policy;

public class PolicyProvider extends Policy {
```

```
    public void initialize() {
        Client clientA = new Client("clientA");
        clientA.addPermissions(new
javax.microedition.io.HttpProtocolPermission("http://locahost:80/"),
            new javax.microedition.io.SSLProtocolPermission("ssl://:*"));
        addClient(clientA);

        Client clientB = new Client("clientB");
        clientB.addPermissions(new
javax.microedition.io.PushRegistryPermission("*", "static,dynamic,alarm"));
        addClient(clientB);

        getUntrustedClient().addPermissions(new
javax.microedition.location.LocationPermission("location", "location"));
    }
}
```

15.2 Creating an Authentication Provider

The purpose of an authentication provider is to verify an application or LIBlet and return the list of appropriate clients. A custom authentication provider must extend the `AuthenticationProvider` class and implement the following abstract methods:

- `AuthenticationProvider.initialize()`
- `AuthenticationProvider.authenticateApplication(com.oracle.meep.security.MIDletProperties, java.io.InputStream)`

The `authenticateApplication()` method should either return the list of clients to which an application or LIBlet is bound, or report an authentication error by throwing `AuthenticationProviderException`.

Application properties from JAD and JAR files can be used for authentication purposes. To access the list of clients defined by the security policy, use the following methods:

- `Policy.getPolicy()`: Access the security policy provider instance.
- `Policy.getClients()`: Get the list of all clients except for virtual clients.
- `Policy.getClient(java.lang.String)`: Get the client by name.
- `Policy.getRootClient()`: Get the virtual root client.
- `Policy.getUntrustedClient()`: Get the virtual untrusted client.

Example 15–2 shows how to create a custom authentication provider that selects clients depending on the application vendor property.

Example 15–2 Custom Authentication Provider

```
package com.company.security;

import com.oracle.meep.security.AuthenticationProvider;
import com.oracle.meep.security.AuthenticationProviderException;
import com.oracle.meep.security.Client;
import com.oracle.meep.security.MIDletProperties;
import com.oracle.meep.security.Policy;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.List;

public class AuthProvider extends AuthenticationProvider {
```

```
    public List<Client> authenticateApplication(MIDletProperties props,
InputStream in) throws AuthenticationProviderException {
        List<Client> result = new ArrayList<>();
        String vendor = props.getProperty("MIDlet-Vendor");

        switch (vendor) {
            case "Manufacturer":
                result.add(Policy.getPolicy().getRootClient());
                break;
            case "TrustedCompany":
                result.add(Policy.getPolicy().getClient("clientA"));
                result.add(Policy.getPolicy().getClient("clientB"));
                break;
            case "UntrustedCompany":
                result.add(Policy.getPolicy().getUntrustedClient());
                break;
            default:
                throw new
AuthenticationProviderException(AuthenticationProviderException.ErrorCode.AUTHENTI
CATION_FAILURE);
        }

        return result;
    }

    public void initialize() {
    }
}
```

15.3 Installing Custom Providers

To install a custom security policy or authentication provider on an emulated device:

1. Build the provider into a single JAR file. You can find API stub files in the `security_api.jar` archive under `lib\ext` in the Java ME SDK installation directory. The default location is `C:\Java_ME_platform_SDK_8.0\lib\ext\security_api.jar`
2. In NetBeans IDE, right-click an emulated device in the Device Selector and select **Security Configuration**.
3. Specify the path to the custom security provider implementation JAR file, and the class names of the authentication and security policy providers. For more information about using the Security Configuration window, see "Configuring the Security Policy".

To install custom security providers on a physical external device, see the documentation for the device.

Part V

Optional Packages

Part V provides information about supported Java Specification Requests (JSRs) in Oracle Java ME SDK 8.

Part V contains the following chapters:

16

API Support

Oracle Java ME SDK 8 supports many standard application programming interfaces (APIs) defined through the Java Community Process (JCP) program. JCP APIs are often referred to as JSRs, named after the Java Specification Request process. JSRs that are not part of the platform are referred to as *optional packages*.

The MEEP platform is based on JSR 228: Information Module Profile - Next Generation (IMP-NG).

For a full list of supported JCP APIs, see Table 16–1. Oracle Java ME SDK 8 provides documentation that describes how certain APIs are implemented. Many supported APIs do not require special implementation considerations, so they are not described. For information about Oracle APIs provided to support the MEEP platform, see Oracle APIs.

For convenience, Javadocs that are the intellectual property of Oracle are located under `docs` in the Java ME SDK installation directory. The remainder can be downloaded from `http://jcp.org`.

Table 16–1 Supported JCP APIs

JSR	Supported APIs	Name and URL
JSR 75	File Connection	*PDA Optional Packages for the J2ME Platform* `http://jcp.org/en/jsr/detail?id=75`
JSR 120	WMA 1.1	*Wireless Messaging API* `http://jcp.org/en/jsr/detail?id=120`
JSR 172	Web Services	*J2ME Web Services Specification* `http://jcp.org/en/jsr/detail?id=172`
JSR 177	APDU and CRYPTO	*Security and Trust Services API for Java ME* `http://jcp.org/en/jsr/detail?id=177`
JSR 179	Location	*Location API for Java ME* `http://jcp.org/en/jsr/detail?id=179`
JSR 280	XML API	*XML API for Java ME* `http://jcp.org/en/jsr/detail?id=280`
JSR 360	CLDC 8	*Connected Limited Device Configuration 8* `http://jcp.org/en/jsr/detail?id=360`
JSR 361	MEEP	*Java ME Embedded Profile* `http://jcp.org/en/jsr/detail?id=361`

Oracle APIs

The MEEP project type supports developing applications for the Oracle Java ME Embedded 8.0 runtime. The Java ME Embedded 8.0 runtime includes several standard JSR APIs and additional Oracle APIs for embedded use cases. These new APIs are:

- Device I/O API: Provides interfaces and classes for communicating with and controlling peripheral devices.
- Configuration API: Provides read and write access to the Java ME runtime configuration.
- HTTP API Client: Provides a Java-based framework for communication with Web Services.
- JSON API: Provides an object model to process the JavaScript Object Notation (JSON) format.
- OAuth 2.0 API: Provides access to the OAuth 2.0 authorization framework.

Javadocs for these APIs are located under `docs` in the Java ME SDK installation directory.

17

JSR 75: PDA Optional Packages

Oracle Java ME SDK 8 supports JSR 75: PDA Optional Packages for the J2ME Platform. JSR 75 includes the `FileConnection` optional package that enables IMlets to access a local device's file system.

This chapter describes how Oracle Java ME SDK 8 implements the `FileConnection` API.

FileConnection API

On an external device, the `FileConnection` API typically provides access to files stored in the device's memory or on a memory card.

In the Oracle Java ME SDK 8 emulator, the `FileConnection` API enables IMlets to access files stored on your computer's hard disk.

The files that can be accessed using the `FileConnection` optional package are stored in the device's directory, for example:

```
userdir\javame-sdk\8.0\work\EmbeddedDevice1\appdb\filesystem
```

Each subdirectory of `filesystem` is called a *root*. Oracle Java ME SDK 8 provides a mechanism for managing roots in the Java ME Embedded Emulator.

While the emulator is running, open the **Tools** menu and select **Manage File System**. In the Manage File System window, you can mount, unmount, or unmount and delete file system roots. Mounted roots are displayed in the top list, and unmounted roots are displayed in the bottom list. You can remount or delete a selected directory. Mounted root directories and their subdirectories are available to applications using the `FileConnection` API. Unmounted roots can be remounted in the future.

- To add a new empty file system root directory, click **Mount Empty** and enter a name for the directory.
- To mount a copy of an existing directory, click **Mount Copy** and browse to select a directory you want to copy. When the File System Root Entry dialog box opens, specify the name for this root. A deep copy of the selected directory is placed into the emulator's file system with the specified root name.
- To make a directory inaccessible to the `FileConnection` API, select it in the list and click **Unmount**. The selected root is unmounted and moved to the unmounted roots list.
- To completely remove a mounted directory, select it and click **Unmount & Delete**.
- To remount an unmounted directory, select it and click **Remount**. The root is moved to the mounted roots list.

- To delete an unmounted directory, select it and click **Delete**. The selected root is removed from the list.

Running PDAPDemo

`PDAPDemo` shows you how to use the `FileConnection` API that is part of the JSR 75 specification.

Browsing Files

The default emulators have one directory, `root1`. This directory is located at:

username`\javame-sdk\8.0\work\`*devicename*`\appdb\filesystem\root1`

For test purposes, copy files or directories into the `root1` directory of the default emulator. You can also add other directories at the same level as `root1`.

Open and run the `PDAPDemo` project:

- Start the `FileBrowser` IMlet. You see a directory listing, and you can browse through the directories and files you placed there.
- Select a directory and click the **View** button to open it.
- Using the **Menu** commands, you can view a file or see its properties. Try selecting the file and choosing **Properties** or **View** from the menu.

  You can view the content of text files in the browser.
- Try using the Java ME Embedded Emulator to unmount and mount directories. Unmounted directories are not visible in the application running on the emulator.

18

JSR 120: Wireless Messaging

Oracle Java ME SDK 8 supports the Wireless Messaging API (WMA) with a sophisticated simulation environment. WMA 1.1 (JSR 120) enables IMlets to send and receive Short Message Service (SMS) or Cell Broadcast Service (CBS) messages.

This chapter describes the tools you can use to develop WMA applications. It begins by showing you how to configure the emulator's support of WMA. Next, it describes the WMA console, a tool for testing WMA applications.

Many of the tasks in this topic can also be accomplished from the command line. For more information, see Running the WMA Tool.

Using the WMA Console to Send and Receive Messages

The WMA console is a tool that enables you to send messages to and receive messages from applications. You can, for example, use the WMA console to send SMS messages to an IMlet running on the emulator.

To start the WMA console in NetBeans IDE 8.0, open the **Tools** menu, select **Java ME**, and then **WMA Console**. Messages can be sent from the WMA Console to an emulator instance.

The console opens as a tab in NetBeans IDE 8.0. The console phone number is displayed as part of the WMA Console tab label (for example, 987654321).

The WMA console phone number is an editable CLDC property. In the Device Selector, right-click the **CLDC, Java(TM) ME Platform SDK 8.0** node and select **Properties**. Enter a new value in the **WMA Console Phone Number** field. If the number is available, it is assigned to the console. If the number is in use, it is assigned to the console the next time you restart NetBeans IDE 8.0.

Each instance of the emulator has a simulated phone number that is shown in the emulator window. The phone numbers are important because they are used as addresses for WMA messages. The phone number is a device property, and it can be changed. In the Device Selector, right-click a device and select **Properties**. Enter a new value in the **Phone Number** field.

Sending a Text or Binary SMS Message

To send a text SMS message, open the WMA Console and click **Send SMS**.

- The **To Clients** list contains phone numbers of all running emulator instances. Select one or more destinations and enter a port number.
- To send a text message, select the **Text Message** tab, enter your message and click **Send**.

- To send the contents of a file as a binary message, click the **Binary Message** tab. Enter the path of a file directly, or click **Browse** to select it in the file system explorer.

> **Note:** The maximum message length for text and binary messages is 4096 bytes.

Sending Text or Binary CBS Messages

To send a text or binary CBS message, click **Send CBS** in the WMA Console. Sending CBS messages is similar to sending SMS messages, except that recipients are unnecessary because it is a broadcast. Specify a message identifier and enter the text or binary content of your message. The maximum message length for text and binary messages is 4096 bytes.

> **Note:** The application running on the emulator receives only the first 160 symbols of the CBS message.

Receiving Messages in the WMA Console

To access the WMA Output Window, open the **Window** menu, select **Java ME** and then **WMA Console Output**. The WMA Output Window has its own phone number displayed in the label. You can send messages from your applications running on the emulator to the WMA console.

Received messages are displayed in the WMA Output Window.

Running the WMA Tool

`wma-tool` is the command-line version of the WMA Console. It is located under `bin` in the Java ME SDK installation directory. The Device Manager must be running before you start `wma-tool`. When the tool is started, it outputs the phone number it is using.

`wma-tool` has the following syntax:

```
wma-tool <command> [options]
```

Each protocol has send and receive commands. The requested command is passed to the tool as the first argument. The following commands are available:

- `receive`: Receive any message
- `smsreceive`: Receives SMS messages
- `cbsreceive`: Receives CBS messages
- `smssend`: Sends SMS message
- `cbssend`: Sends CBS message

All `*send` commands send the specified message and exit. All `*receive` commands print incoming messages until they are explicitly stopped.

The following options are available:

-o
Stores binary content to the output directory specified after a space.

-t
After starting in non-interactive mode, waits for a message for the number of seconds specified after a space.

-f
Stores text content as files instead of printing it.

-q
Runs in quite mode.

Examples of smsreceive and cbsreceive

The syntax for receiving a message is similar for both protocols:

smsreceive [-o <output_dir>] [-t <timeout>] [-q]

cbsreceive [-o <output_dir>] [-t <timeout>] [-q]

The following example demonstrates how to receive a message from an emulator:

1. Start the Java ME Embedded Emulator. Click the Windows **Start** menu, open **All Programs**, select **Java ME Platform SDK 8.0**, and then **Java ME Embedded Emulator**.
2. Use the following command to run `wma-tool`:

```
C:\Java_ME_platform_SDK_8.0\bin> wma-tool smsreceive

WMA tool started with phone number: 987654321
press <Enter> to exit.
```

3. In the emulator, run the SMS Send IMlet and send a message to the WMA Console. Enter the console phone number.

   The console receives the message as follows:

```
SMS Received:
          From: 123456789
Timestamp: Thu Aug 23 23:31:26 PDT 2012
          Port: 50000
  Content type: Text
      Encoding: GSM7BIT
       Content: A message from EmbeddedDevice1 to wma-tool
Waiting for another message, press <Enter> to exit.
```

Example of smssend

The following syntax is used for sending SMS messages:

```
wma-tool smssend <target_phone> <target_port> <message_content>
```

The message content can be specified either as text or using the `-f` option with the name of the file that you want to send as a binary message.

For example, to send a text message to phone number 123456789 on port number 50000, use the following command:

```
C:\Java_ME_platform_SDK_8.0\bin> wma-tool smssend 123456789 50000 "smssend message
from wma-tool"
```

Example cbssend

The following syntax is used for sending CBS messages:

```
wma-tool cbssend <message_id> <message_content>
```

The message content can be specified either as text or using the `-f` option with the name of the file that you want to send as a binary message.

For example, to send a text message with message identifier 50001, use the following command:

```
C:\Java_ME_platform_SDK_8.0\bin> wma-tool cbssend 50001 "cbssend message from
wma-tool"
```

19

JSR 172: Web Services Support

The Oracle Java ME SDK 8 emulator supports JSR 172: J2ME Web Services Specification. JSR 172 provides APIs for accessing web services from mobile applications. It also includes an API for parsing XML documents.

NetBeans IDE 8.0 provides a stub generator that automates creating source code for accessing web services that conform to the J2ME Web Services Specification.

Generating Stub Files from WSDL Descriptors

You can add stub files to any MEEP application.

> **Note:** If you are using NetBeans IDE 8.0, the SOAP Web Services plugin must be installed and activated.

To add a stub file:

1. In the Projects window, expand the tree for a project, right-click the **Source Packages** node and select **New**, and then **Other**. Select the **Java ME Embedded** category and then **Java ME Web Service Client**.
2. In the New Java ME Webservice Client page, you can do one of the following:
   - Click **Running Web Service** and enter the URL for the WSDL, and then click **Retrieve WSDL**.
   - Click **Specify the Local filename for the retrieved WSDL** and browse to find a file on your system.

   In either case, you must enter a package name (if it is not supplied), and then click **Finish**. The new package appears in the project and includes an interface file and a stub file.
3. You can now edit your source files to call the content that the stub file provides, then build and run the project.

   For additional information about how to generate stub files and other supporting files from the command line, see Generating Stub Files from the Command Line.

Generating Stub Files from the Command Line

Mobile clients can use the Stub Generator to access web services. The `wscompile` tool generates stub files, ties, serializers, and WSDL files used in Java API for XML (JAX) RPC clients and services. The tool reads a configuration file that specifies a WSDL file, a model file, or a compiled service endpoint interface.

The wscompile.exe file is located under bin in the Java ME SDK installation directory. The syntax for the stub generator command is as follows:

```
wscompile <command> [options] <config_file>
```

The following commands for the wscompile tool are available:

Command	Description
-gen	Same as -gen:client
-gen:client	Generates client artifacts (stubs, etc.)
-import	Generates interfaces and value types only

The following options for the wscompile tool are available:

Option	Description
-d <output_dir>	Specifies where to place generated output files
-f:<features>	Enables the given features
-g	Generates debugging information
-features:<features>	Same as -f
-httpproxy:<host>:<port>	Specifies a HTTP proxy server (port defaults to 8080)
-model <file>	Writes the internal model to the given file
-O	Optimizes generated code
-s <dir>	Specifies where to place generated source files
-verbose	Outputs messages about what the compiler is doing
-version	Prints version information
-cldc1.0	Sets the CLDC version to 1.0 (default). Float and double become String.
-cldc1.1	Sets the CLDC version to 1.1. Float and double are OK.
-cldc1.0info	Shows all CLDC 1.0 information and warning messages.

Note: Only one -gen option must be specified. The -f option requires a comma-separated list of features.

The wscompile tool can read WSDL files, compiled service endpoint interface (SEI) files, or model files as input. The following table lists features that can be specified with the -f option for the wscompile command, and the type of files that can be provided when the feature is specified. Multiple features must be separated by commas.

Option	Description
explicitcontext	Turns on explicit service context mapping
nodatabinding	Turns off data binding for literal encoding
noencodedtypes	Turns off encoding type information

Option	Description
`nomultirefs`	Turns off support for multiple references
`novalidation`	Turns off full validation of imported WSDL documents
`searchschema`	Searches schema for subtypes
`serializeinterfaces`	Turns on direct serialization of interface types
`wsi`	Enables WSI-Basic Profile features (default)
`resolveidref`	Resolves `xsd:IDREF`
`donotunwrap`	No unwrap

The following are examples of using the `wscompile` command:

```
wscompile -gen -d generated config.xml
wscompile -gen -f:nounwrap -O -cldc1.1 -d generated config.xml
```

20

JSR 177: Smart Card Security (SATSA)

This topic describes how you can use Oracle Java ME SDK 8 to work with SATSA in your applications. JSR 177: Security and Trust Services API (SATSA) for J2ME provides smart card access and cryptographic capabilities to applications running on small devices. The SATSA specification defines several APIs as optional packages. The Oracle Java ME SDK 8 emulator supports the following SATSA packages:

- **SATSA-APDU**: Enables applications to communicate with smart card applications using a low-level protocol.
- **SATSA-CRYPTO**: A general-purpose cryptographic API that supports message digests, digital signatures, and ciphers.

For a more general introduction to SATSA and using smart cards with small devices, see the *SATSA Developer's Guide* at `http://www.oracle.com/technetwork/index.html`.

To develop your own Java Card applications, download the Java Card Development Kit at `http://www.oracle.com/technetwork/java/javame/index.html`.

Card Slots in the Emulator

SATSA devices are likely to have one or more slots that hold smart cards. Applications that use SATSA to communicate with smart cards must specify a slot and a card application.

The Oracle Java ME SDK 8 emulator is not an external device and, therefore, does not have physical slots for smart cards. Instead, it communicates with a smart card application using a socket protocol. The other end of the socket might be a smart card simulator, or it might be a proxy that communicates with smart card hardware.

The Oracle Java ME SDK 8 emulator includes two simulated smart card slots. Each slot has an associated socket that represents one end of the protocol that is used to communicate with smart card applications.

The default card emulator host name is `localhost`, and the default ports are 9025 for slot 0 and 9026 for slot 1. These port defaults are a property of the device. To change the port numbers, right-click the device in the Device Selector, and select **Properties**.

Adjusting Access Control

Access control permissions and PIN properties can be specified in text files. When the first APDU connection is established, the implementation reads the ACL and PIN data from a file named `acl_` followed by a slot number, located in the device's

configuration directory. For example, the access control file for slot 0 on the EmbeddedDevice1 device is:

```
userdir\javame-sdk\8.0\work\EmbeddedDevice1\appdb\acl_0
```

If the file is absent or contains errors, the access control verification for this slot is disabled.

The file can contain information about PIN properties and application permissions.

Specifying PIN Properties

PIN properties are represented by a pin_data record in the access control file. The record has the following format:

Example 20–1 PIN Properties Example

```
pin_data {
    id number
    label string
    type       bcd | ascii | utf | half-nibble | iso
    min        minLength
    max        maxLength
    stored     storedLength
    reference  byte
    pad        byte - optional
    flag       case-sensitive | change-disabled | unblock-disabled
               needs-padding | disable-allowed | unblockingPIN
   }
```

Specifying Application Permissions

Application permissions are defined in access control file (acf) records. The record has the following format:

Example 20–2 Access Control File Record Format

```
acf AID fnumbers separated by blanks {
    ace {
         root CA name
         ...
         apdu {
                eight numbers separated by blanks
                ...
         }
         ...
         pin_apdu {
                 id number
            verify | change | disable | enable | unblock
            four hexadecimal numbers
                 ...
         }
         ...
    }
    ...
}
```

The acf record is an *access control file*. The AID after acf identifies the application. A missing AID indicates that the entry applies to all applications. The acf record can

contain ace records. If there are no ace records, then access to an application is restricted by this acf.

The ace record is an *access control entry*. It can contain root, apdu, and pin_apdu records.

The root record contains one certification authority (CA) name. If the IMlet suite was authorized using a certificate issued by this CA, the ace grants access to this IMlet. A missing root field indicates that the ace applies to all identified parties. One principal is described by one line. This line must contain only the word root and the principal name, for example:

```
root CN=thehost;OU=JCT;O=dummy CA;L=Santa Clara;ST=CA;C=US
```

The apdu record describes an APDU permission. A missing permission record indicates that all operations are allowed.

An APDU permission contains one or more sequences of 8 hexadecimal values, separated by blanks. The first 4 bytes describe the APDU command, and the other 4 bytes are the mask, for example:

```
apdu {
     0 20  0 82  0 20  0 82
    80 20  0  0 ff ff  0  0
}
```

All the numbers are hexadecimal. Tabulation, blank, CR, and LF symbols are used as separators. Separators can be omitted before and after braces ({ }).

The pin_apdu records contain information necessary for PIN entry methods, which is the PIN identifier and APDU command headers, or remote method names.

Access Control File Example

A sample control file is provided in Example 20–3.

Example 20–3 Access Control File Example

```
pin_data {
  label   Unblock pin
  id        44
  type      utf
  min       4
  stored    8
  max       8
  reference 33
  pad       ff
  flag      needs-padding
  yflag     unblockingPIN
}
pin_data {
  label     Main pin
  id        55
  type      half-nibble
  min       4
  stored    8
  max       8
  reference 12
  pad       ff
  flag      disable-allowed
  flag      needs-padding
```

```
}

acf a0 0 0 0 62 ee 1 {
  ace {
    root CN=thehost;OU=JCT;O=dummy CA;L=Santa Clara;ST=CA;C=US

    pin_apdu {
            id 55
            verify 1 2 3 1
            change 4 3 2 2
            disable 1 1 1 3
            enable 5 5 5 4
            unblock 7 7 7 5
        }
  }
}

acf a0 00 00 00 62 03 01 0c 02 01 {
  ace {
        root CN=thehost;OU=JCT;O=dummy CA;L=Santa Clara;ST=CA;C=US
        apdu {
               0 20  0 82  0 20  0 82
               80 20  0  0 ff ff  0  0
             }
        apdu {
               80 22  0  0 ff ff  0  0
             }
        }
  }
acf a0 00 00 00 62 03 01 0c 02 01 {

  ace {
    apdu {
        0 20 0 82 ff ff ff ff
    }
  }
}

acf a0 00 00 00 62 03 01 0c 06 01 {

  ace {
    apdu {
        0 20 0 82 ff ff ff ff
    }
  }
}
```

21

JSR 179: Location API Support

JSR 179: Location API for J2ME gives applications the opportunity to use a device's location capabilities. For example, some devices include Global Positioning System (GPS) hardware. Other devices might be able to receive location information from the wireless network. The Location API provides a standard interface to location information, regardless of the underlying technique.

In the Location API, a *location provider* encapsulates a positioning method and supplies information about the device's location. The application requests a provider by specifying required criteria, such as the desired accuracy and response time. If an appropriate implementation is available, then the application can use it to obtain information about the device's physical location.

Oracle Java ME SDK 8 includes a simulated location provider. You can use the emulator's External Events Generator to specify where the emulator should think it is located. In addition, you can configure the properties of the provider itself, and you can manage a database of landmarks.

Setting the Emulator's Location at Runtime

To specify the simulated location of the emulator while it is running:

1. In the emulator, open the **Tools** menu and select **External Events Generator**. Click the **Location** tab.
2. Under the **Location** group, specify values for latitude, longitude, altitude, speed, and course. Applications that use the Location API can retrieve these values as the location of the emulator.

For more elaborate testing, you can set up a location script that describes motion over time. Location scripts are XML files that consist of a list of locations, called *waypoints*, and associated times. Oracle Java ME SDK 8 determines the current location of the emulator by interpolating between the points in the location script. Here, for example, is a simple location script that specifies a starting point (`time="0"`) and moves to a new point in 10 seconds (`time="10000"`):

Example 21–1 Location Script Example

```
<waypoints>
  <waypoint time="0"
            latitude="14" longitude="50" altitude="310" />
  <waypoint time="10000"
            latitude="14.5" longitude="50.1" altitude="215" />
</waypoints>
```

The altitude measurement is in meters, and the time values are in milliseconds.

Use a text editor to create your location script. You can point the External Events Generator to this script by clicking **Browse** next to the **Script** field. Below the **Script** field are controls for playing, pausing, stopping, and moving to the beginning and end of the location script. You can also drag the time slider to a particular point.

Some devices are also capable of measuring their orientation. To make this kind of information available to your application, change the **State** field in the **Orientation** group to **Supported**, and specify values for azimuth, pitch, and roll. The **Magnetic Orientation** check box defines whether the azimuth and pitch measurements are relative to the Earth's magnetic field or relative to true north and gravity.

To test how your application handles unexpected conditions, try changing the **State** field in the **Location Provider** group to **Temporarily Unavailable** or **Out of Service**. When your application attempts to retrieve the emulator's location, an exception is thrown, and you can see how your application responds.

Part VI

Sample Applications

Part VI provides information about working with sample programs in Oracle Java ME SDK 8.

Part VI contains the following chapters:

22

Using Sample Applications

The Oracle Java ME SDK 8 sample applications introduce you to the emulator's API features and the Oracle Java ME SDK 8 features, tools, and utilities that support the various APIs.

> **Note:** Before using the Oracle Java ME SDK 8 sample applications, carefully read Appendix B, "Installation and Runtime Security Guidelines." Some demonstrations use network access and open ports, and do not include protection against malicious intrusion. If you run the sample projects, ensure that your environment is secure.

Installing Sample Applications

Sample applications are installed using an Oracle Java ME SDK 8 ZIP file that has the following name format: `jmesdk-8_0-samples-<build_number>-<date>.zip`. The default location for installing the sample applications is in the Oracle Java ME SDK 8 installation directory.

To install sample applications:

1. Download the sample applications file from the Oracle Technology Network (OTN).
2. Move the sample applications file to the Oracle Java ME SDK 8 installation directory. By default, it is `C:\Java_ME_platform_SDK_8.0`.
3. Extract the sample applications file and the file contained inside (`com.oracle.javame.sdk.sample.applications.zip`) to the `apps` directory.
4. Ensure that the following subdirectories are available in the `apps` directory:
   - `DataCollectionDemo`
   - `GPIODemo`
   - `I2CDemo`
   - `LightTrackDemo`
   - `NetworkDemo`
   - `PDAPDemo`
   - `SystemControllerDemo`

Configuring the Web Browser and Proxy Settings

If you are behind a firewall, you can configure the sample applications to use proxy server settings that you define.

The sample application proxy server settings typically match the proxy server settings used in your web browser. To manually set the proxy server settings for your sample applications, do the following:

1. In NetBeans IDE 8.0, open the **Tools** menu, select **JavaME**, and then **Device Selector**.
2. Right-click **CLDC, Oracle Java ME SDK 8.0** and select **Properties**.
3. Specify the **HTTP Proxy Settings**, **HTTP Proxy Host**, and **HTTP Proxy Port** fields to match your network and browser settings.

Running Sample Applications

This section describes how to use sample applications created specifically for Oracle Java ME SDK 8. Because these sample applications are headless, you must observe the application status in the emulator's External Events Generator, in the Output window, or in the console window if you execute the demo straight from the command line.

Running the DataCollectionDemo

The `DataCollectionDemo` demonstrates the following functionality:

- Multiple virtual machines (MVM)
- Inter-IMlet communication using local datagrams
- Device I/O API pulse counter
- Device I/O API serial peripheral interface
- Logging API

In the `DataCollectionDemo`, several data collector IMlets read data from peripheral devices using the Device I/O API and send the data to a data processor.

For more information about the setup and behavior of the `DataCollectionDemo`, see the `readme.txt` file, under `apps\DataCollectionDemo` in the Java ME SDK installation directory.

For more information about the Qualcomm IoE platform, see *Oracle Java ME Embedded Getting Started Guide for the Reference Platform (Qualcomm IoE)*.

Running the GPIODemo

The `GPIODemo` can run on an emulator. The implementations are different, because the emulator uses the External Events Generator, and the external device supports direct interaction.

To run `GPIODemo` on the emulator:

1. Run `GPIODemo` on the **EmbeddedDevice1** emulator.
2. Click the **GPIO Pins** tab. This view approximates the device actions.
3. Open the **Tools** menu and select **External Events Generator**. Open the **GPIO** tab.
4. Click **BUTTON 1** in the External Events Generator to toggle the state of the pin named BUTTON 1 in the EmbeddedDevice1 emulator. If the button value in the

External Events Generator is changed to **High**, then the button value in the EmbeddedDevice1 emulator is also changed to High.

Running the I2CDemo

The `I2CDemo` is designed to work with Oracle Java ME SDK 8. It has no user interaction.

To run `I2CDemo` on the emulator:

1. Run `I2CDemo` on the **EmbeddedDevice1** emulator.
2. Click the **I2C** tab.
3. Run `I2CDemo.jad` located under `apps\I2CDemo` in the Java ME SDK installation directory.
4. The `I2CDemo` acquires a slave named **I2C_Joystick**, writes data to the slave, and retrieves it. The `I2CDemo` is successful if the Sent Data and Received Data match.

Running the NetworkDemo

You can configure the `NetworkDemo` as a server or as a client by editing the application descriptor. You start two instances of `NetworkDemo`; the first one acts as a server and the second one acts as a client. The client instance attempts to connect to the server instance and if the connection is successful they exchange a message.

To run `NetworkDemo` on the emulator:

1. Create two instance projects of the `NetworkDemo` sample project.
2. Right-click the first project and select **Properties**.
3. In the **Platform** category, select the device **EmbeddedDevice1**. In the **Application Descriptor** category, set the value of the following property:

   ```
   Oracle-Demo-Network-Mode:Server
   ```

4. Click **OK**.
5. Start the first project. It opens on the emulator `EmbeddedDevice1` and waits for a connection.
6. Right-click the second project and select **Properties**.
7. In the **Platform** category, select the device **EmbeddedDevice2**. In the **Application Descriptor** category, set the value of the following property:

   ```
   Oracle-Demo-Network-Mode:Client
   ```

8. Click **OK**.
9. Start the second project. It opens on the emulator `EmbeddedDevice2`.
10. The client attempts to connect to the server. If successful, the following is displayed in the output for the first project (the server):

    ```
    Waiting for connection on port 5000
    Connection accepted
    Message received - Client messages
    ```

    The following is displayed in the output of the second project (the client):

    ```
    Connected to server localhost on port 5000
    Message received - Server string
    ```

Running the NetworkDemo on the Reference Board

You can run one of the instance projects on the board and the other in one of the emulators.

> **Note:** Applications that are run on an embedded platform, such as the Qualcomm IoE or Raspberry Pi, must be signed. For more information, see the Oracle Java ME SDK 8 *Getting Started Guide* for your embedded platform.

To run the client on the board and the server in one of the emulators:

1. Right-click the first project (the server) and select **Properties**. In the **Platform** category, select the device **EmbeddedDevice1** (the emulator). In the **Application Descriptor** category, set the value of the property **Oracle-Demo-Network-Mode** to **Server** and click **OK**.
2. Start the first project (the server). It runs on the emulator and waits for a connection.
3. Right-click the second project (the client) and select **Properties**. In the **Platform** category, select the device **EmbeddedExternalDevice1** (the board). In the **Application Descriptor** category, set the value of the property **Oracle-Demo-Network-Mode** to **Client** and the value of the property **Oracle-Demo-Network-Address** to the IP address of the computer where NetBeans IDE 8.0 is running and click **OK**.
4. Start the second project (the client). It runs on the board and attempts to connect to the server. If successful, the following is displayed in the output tab of the first project (the server):

   ```
   Connection accepted
   Message received - Client messages
   ```

   The following is displayed in the TCP log of the board (the client):

   ```
   Connected to server 10.0.0.10 on port 5000
   Message received - Server String
   ```

Running the PDAPDemo

To run `PDAPDemo` on the emulator:

1. Create test files and directories inside the emulator's file system. The file system is located in the Java ME SDK configuration directory. For example, for EmbeddedDevice1, the file system is located under `userdir\javame-sdk\8.0\work\EmbeddedDevice1\appdb\filesystem\root1`
2. Open the project in NetBeans IDE 8.0, right-click the project and select **Properties**.
3. In the **Platform** category, select the device **EmbeddedDevice1** and click **OK**.
4. In the Device Selector window, right-click an **EmbeddedDevice1** emulator, select **Run Project** and then **PDAPDemo**.
5. Start the project.

6. On the **EmbeddedDevice1** emulator, open the **Tools** menu and select **Manage File System** to see a list of mounted file systems.
7. Open a terminal emulator and create a Telnet connection to **localhost** on port **5001**.

> **Note:** The Telnet negotiation mode must be set to **Passive**. The negotiation mode can be set inside a Telnet client application (for example, PuTTY), by selecting **Category**, then **Connection**, then **Telnet**, and then **Passive**.

8. A command line opens where you can browse the emulator's file system. You can use the following commands:
   - `cd`: Change directory
   - `ls`: List information about the files for the current directory
   - `new`: Create a file or directory
   - `prop`: Show properties of a file
   - `rm`: Remove a file
   - `view`: View a file's content

Running the PDAPDemo on the Reference Board

To run `PDAPDemo` on the reference board:

1. Right-click the project and select **Properties**. In the **Platform** category, select the device **EmbeddedExternalDevice1** and click **OK**.
2. Start the project. It runs on the reference board.
3. Open a terminal emulator and create a raw connection to the IP address of the board on port **5001**.
4. The command line that opens is the same as the one you use when you run the `PDAPDemo` on the emulator.

The file system of the demo is stored in the `java` directory on the SD card. In that directory, there are a subdirectories that are named using the number identifier that the AMS assigns to an IMlet during its installation.

Running the LightTrackDemo

The `LightTrackerDemo` is specifically aimed at demonstrating functionality on an embedded device.

In the `LightTrackerDemo`, a certain number of LEDs are turned on and turned off on the board, in a sequence that you can control. It makes use of the Device I/O API and the GPIO port to demonstrate its functionality. It requires connection of an ADC channel to an on-board potentiometer.

For more information about the setup and behavior of the Light Tracker demo, see the `readme.txt` file located under `apps\LightTrackDemo` in the Java ME SDK installation directory.

Running the SystemControllerDemo

The `SystemControllerDemo` is specifically aimed at showing off functionality on an embedded device, such as the Qualcomm IoE reference platform.

The purpose of the `SystemControllerDemo` is to control the life cycle of IMlets on the reference platform. It makes use of the following functionalities:

- Multitasking Virtual Machine (MVM)
- IMlet auto-start
- Application Management System (AMS) API
- Logging API
- General Purpose Input/Output (GPIO)
- Watchdog timer

For more information about the setup and behavior of the System Controller demo, see the `readme.txt` file located under `apps\SystemControllerDemo` in the Java ME SDK installation directory.

For more information about the Qualcomm IoE platform, see *Oracle Java ME Embedded Getting Started Guide for the Reference Platform (Qualcomm IoE)*.

Troubleshooting

Sometimes a sample application does not run successfully. Often, the problem is your environment.

- Some demonstrations require specific setup and instructions. For example, if a sample uses web services and you are behind a firewall, you must configure the emulator's proxy server settings. See "Configuring the Web Browser and Proxy Settings."
- Because sample programs can be started remotely, virus checking software can sometimes prevent them from running. In the console, you see warnings that the emulator cannot connect.

  Consider configuring your antivirus software to allow access to sample application directories and components.

Part VII

Appendixes

Part VII provides supporting information, such as running the emulator using the Windows command line, and security considerations when using sample applications.

Part VII contains the following appendixes:

- Appendix A, "Using the Command-Line Emulator"
- Chapter B, "Installation and Runtime Security Guidelines"

A

Using the Command-Line Emulator

The Oracle Java ME SDK 8 Embedded Emulator can be started from the Windows command line. After the emulator starts, it runs and behaves the same as it does when started from NetBeans IDE 8.0.

Starting the emulator from the Windows command line enables you to use a number of emulator options. For more information, see Useful Emulator Command Options.

You can find the Oracle Java ME SDK 8 command-line emulator under `bin` in the Oracle Java ME SDK 8 installation directory.

Using the Oracle Java ME SDK 8 Emulator

To start the emulator from the Windows command line:

1. Open the Windows command prompt. There are several ways to do this:
   - Press Win+R on the keyboard, or open the **Start** menu and select **Run** to open the Run window. Now, type `cmd` and click **OK**.
   - Open the **Start** menu, search for `cmd.exe` and run it.
   - Double-click `C:\Windows\System32\cmd.exe`.
2. Change to the `bin` directory in the Oracle Java ME SDK 8 installation directory. For the default location, use the following command:

   ```
   > cd C:\Java_ME_platform_SDK_8.0\bin
   ```

3. Run the `emulator.exe` command. Use the `-Xdevice` option to specify the device you would like to run, and the `-Xdescriptor` option to specify the JAD file you would like to run on the device. For example, to run the `sample_imlet.jad` file on EmbeddedDevice1, use the following command:

   ```
   C:\Java_ME_platform_SDK_8.0\bin> emulator.exe -Xdevice:EmbeddedDevice1
   -Xdescriptor:C:\Java_ME_platform_SDK_8.0\apps\sample\sample_imlet.jad
   ```

> **Note:** You can run the `emulator` command without the `.exe` extension.

Useful Emulator Command Options

The `emulator` command can be used with the following command-line options:

-help
Displays usage information with a complete list of command-line options.

-Xjam:list
Displays a list of installed IMlets.

-Xquery
Displays a list of supported devices.

-Xjam:install=*URL*
Installs a JAD file over the air and executes the IMlet. You must specify a URL of the JAD file, for example:

```
> emulator -Xjam:install=http://www.example.com/TestJAD.jad
```

-Xjam:run={*storage_name*|*storage_number*}
Runs an IMlet. You must specify either the storage name or storage number, which can be displayed using the `-Xjam:list` command.

-Xjam:remove={*storage_name*|*storage_number*|all}
Removes an IMlet. To remove a specific IMlet, you must specify either the storage name or storage number, which can be displayed using the `-Xjam:list` command. To remove all IMlets, specify `all` as the parameter.

-Xdescriptor:*path*
Installs a JAD file, executes the IMlet locally, and removes it after completion. You must specify the path and name of the JAD file, for example:

```
> emulator -Xdescriptor:C:\imlets\sample.jad
```

-Xautotest:*URL*
Runs an IMlet in autotest mode. You must specify the URL of the JAD file, for example:

```
> emulator -Xautotest:http://127.0.0.1:8080/getNextApp.jad
```

B

Installation and Runtime Security Guidelines

Oracle Java ME SDK 8 requires an execution model that makes certain network resources are available for emulator execution. These required resources might include (but are not limited to) a variety of communication capabilities between product components.

> **Note:** The Oracle Java ME SDK 8 installation and runtime system is a developer system. It is not designed to guard against any malicious attacks from outside intruders.

During execution, the Oracle Java ME SDK 8 architecture can present an insecure operating environment to the platform's installation file system, and its runtime environment. For this reason, it is critically important to observe the precautions outlined in these guidelines when you install and run Oracle Java ME SDK 8.

Maintaining Optimum Network Security

To maintain optimum network security, Oracle Java ME SDK 8 can be installed and run in an isolated network environment, where the Oracle Java ME SDK 8 system is not connected directly to the Internet. It can also be connected to a secure company intranet environment, which will reduce unwanted exposure to malicious intrusion.

An example of an Oracle Java ME SDK 8 requirement for an Internet connection is when wireless functionality requires a connection to the Internet to support communications with the wireless network infrastructure that is part of an Oracle Java ME SDK 8 application execution process. Whether or not an Internet connection is required depends on the particular application running on Oracle Java ME SDK 8. For example, some applications can use an HTTP connection.

If Oracle Java ME SDK 8 is open to any network access, then you must take the following precautions to protect valuable resources from malicious intrusion:

- Installing the Java ME Demos plugin is optional. Some sample projects use network access and open ports. Because the sample code does not include protection against malicious intrusion, ensure that your environment is secure if you install and run the sample projects.
- Install Oracle Java ME SDK 8 behind a secure firewall that strictly limits unauthorized network access to the Oracle Java ME SDK 8 file system and services. Limit access privileges to those that are required for Oracle Java ME SDK 8 usage while allowing all the bidirectional local network communications that are necessary for Oracle Java ME SDK 8 functionality. The firewall configuration must support these requirements to run the Oracle Java ME SDK 8 while also

addressing them from a security standpoint.

- Follow the principle of *least privileged* by assigning the minimum set of system access permissions required to install and execute Oracle Java ME SDK 8.
- Do not store any sensitive information on the same file system that is hosting Oracle Java ME SDK 8.
- To maintain the maximum level of security, ensure that all the latest updates for the operating system are installed.

Glossary

access point

A network-connectivity configuration that is predefined on a device. An access point can represent different network profiles for the same bearer type, or for different bearer types that may be available on a device, such as Wi-Fi or Bluetooth.

ADC

Analog-to-digital converter. A hardware device that converts analog signals (time and amplitude) into a stream of binary numbers that can be processed by a digital device.

AMS

Application management system. The system functionality that completes tasks such as installing applications, updating applications, and managing applications between the foreground and background.

APDU

Application Protocol Data Unit. A communication mechanism used by SIM cards and smart cards to communicate with card reader software or a card reader device.

API

Application programming interface. A set of classes used by programmers to write applications that provide standard methods and interfaces and eliminate the need for programmers to reinvent commonly used code.

ARM

Advanced RISC Machine. A family of computer processors that use reduced instruction set (RISC) CPU technology, developed by ARM Holdings. ARM is a licensable instruction set architecture (ISA) and is used in the majority of embedded platforms.

AT commands

A set of commands developed to facilitate modem communications, such as dialing, hanging up, and changing the parameters of a connection. Also known as the Hayes command set, AT means *attention.*

CLDC

Connected Limited Device Configuration. A Java ME platform configuration for devices with limited memory and network connectivity. It uses a low-footprint Java virtual machine such as the CLDC HotSpot Implementation, and several minimalist Java platform APIs for application services.

configuration

Defines the minimum Java runtime environment (for example, the combination of a Java Virtual Machine and a core set of Java platform APIs) for a family of Java ME platform devices.

DAC

Digital-to-analog converter. A hardware device that converts a stream of binary numbers into an analog signal (time and amplitude), such as audio playback.

GPIO pins

The general purpose input/output pins are unassigned pins on an embedded platform that can be assigned or configured as needed by a developer.

GPIO port

A group of GPIO pins (typically 8 pins) arranged in a group and treated as a single port.

HTTP

HyperText Transfer Protocol. The most commonly used Internet protocol, based on TCP/IP that is used to fetch documents and other hypertext objects from remote hosts.

HTTPS

Secure HyperText Transfer Protocol. A protocol for transferring encrypted hypertext data using Secure Socket Layer (SSL) technology.

IMlet

Similar to a MIDP 2.0 MIDlet, an IMlet is a small application specifically for running in an embedded environment. An IMlet uses classes defined by the MEEP 8.0 and CLDC 1.8 specifications.

IMlet suite

A way of packaging one or more IMlets for easy distribution and use. Similar to a MIDlet suite, but for smaller applications running in an embedded environment. Each IMlet suite contains a Java application descriptor file (`.jad`), which lists the class names and files names for each IMlet, and a Java Archive file (`.jar`), which contains the class files and resource files for each IMlet

I2C

Inter-Integrated Circuit. A multimaster, serial computer bus used to attach low-speed peripherals to an embedded platform

JAD file

Java Application Descriptor file. A file provided in a MIDlet or IMlet suite that contains attributes used by application management software (AMS) to manage the MIDlet or IMlet life cycle, and other application-specific attributes used by the MIDlet or IMlet suite itself.

JAR file

Java Archive file. A platform-independent file format that aggregates many files into one. Multiple applications written in the Java programming language and their required components (class files, images, sounds, and other resource files) can be bundled in a JAR file and provided as part of a MIDlet or IMlet suite.

JCP

Java Community Process. The global standards body guiding the development of the Java programming language.

Java ME platform

Java Platform, Micro Edition. A group of specifications and technologies that pertain to running the Java platform on small devices, such as cell phones, pagers, set-top boxes, and embedded devices. More specifically, the Java ME platform consists of a configuration (such as CLDC) and a profile (such as MEEP) tailored to a specific class of device.

JSR

Java Specification Request. A proposal for developing new Java platform technology, which is reviewed, developed, and finalized into a formal specification by the JCP program.

Java Virtual Machine

A software execution engine that safely and compatibly executes the byte codes in Java class files on a microprocessor.

MEEP

Oracle Java ME Embedded Profile. A profile for embedded (headless) devices, the MEEP specification (JSR 361) includes APIs for security, networking, connectivity, concurrency, and other functionality, and but not graphics and user interface APIs.

MVM

Multiple virtual machines. A software mode that can run more than one MIDlet or IMlet at a time.

obfuscation

A technique used to complicate code by making it harder to understand when it is decompiled. Obfuscation makes it harder to reverse-engineer applications and therefore, steal them.

optional package

A set of Java ME platform APIs that provides additional functionality by extending the runtime capabilities of an existing configuration and profile.

profile

A set of APIs added to a configuration to support specific uses of an embedded or mobile device. Along with its underlying configuration, a profile defines a complete and self-contained application environment.

provisioning

A mechanism for providing services, data, or both to an embedded or mobile device over a network.

pulse counter

A hardware or software component that counts electronic pulses, or events, on a digital input line, for example, a GPIO pin.

push registry

The list of inbound connections, across which entities can push data. Each item in the list contains the URL (protocol, host, and port) for the connection, the entity permitted to push data through the connection, and the application that receives the connection.

RISC

Reduced Instruction Set Computing. A CPU design based on simplified instruction sets that provide higher performance and faster execution of individual instructions. The ARM architecture is based on RISC design principles.

RMI

Remote Method Invocation. A feature of Java SE technology that enables Java technology objects running in one virtual machine to seamlessly invoke objects running in another virtual machine.

SD card

Secure Digital card. A non-volatile memory card format for use in portable devices, such as mobile phones and digital cameras, and embedded systems. SD cards come in three different sizes, with several storage capacities and speeds.

SIM

Subscriber identity module. An integrated circuit embedded into a removable SIM card that securely stores the International Mobile Subscriber Identity (IMSI) and the related key used to identify and authenticate subscribers on mobile and embedded devices.

slave mode

Describes the relationship between a master and one or more devices in a Serial Peripheral Interface (SPI) bus arrangement. Data transmission in an SPI bus is initiated by the master device and received by one or more slave devices, which cannot initiate data transmissions on their own.

smart card

A card that stores and processes information through the electronic circuits embedded in silicon in the substrate of its body. Smart cards carry both processing power and information. A SIM card is a special kind of smart card for use in a mobile device.

SMS

Short Message Service. A protocol that allows transmission of short text-based messages over a wireless network. SMS messaging is the most widely used data application in the world.

SOAP

Simple Object Access Protocol. An XML-based protocol that enables objects of any type to communicate in a distributed environment. It is most commonly used to develop web services.

SPI

Serial Peripheral Interface. A synchronous bus commonly used in embedded systems that allows full duplex communication between a master device and one or more slave devices.

SSL

Secure Sockets Layer. A protocol for transmitting data over the Internet using encryption and authentication, including the use of digital certificates and both public and private keys.

SVM

Single Virtual Machine. A software mode that can run only one MIDlet or IMlet at a time.

task

At the platform level, each separate application that runs within a single Java Virtual Machine is called a task. The API used to instantiate each task is a stripped-down version of the Isolate API defined in JSR 121.

TCP/IP

Transmission Control Protocol/Internet Protocol. A fundamental Internet protocol that provides for reliable delivery of streams of data from one host to another.

terminal profile

Device characteristics of a terminal (mobile or embedded device) passed to the SIM card along with the IMEI during SIM card initialization. The terminal profile tells the SIM card what values are supported by the device.

UART

Universal Asynchronous Receiver/Transmitter. A piece of computer hardware that translates data between serial and parallel formats. It is used to facilitate communication between different kinds of peripheral devices, input/output streams, and embedded systems, to ensure universal communication between devices.

URI

Uniform Resource Identifier. A compact string of characters used to identify or name an abstract or physical resource. A URI can be further classified as a uniform resource locator (URL), a uniform resource name (URN), or both.

USB

Universal Serial Bus. An industry standard that defines the cables, connectors, and protocols used in a bus for connection, communication, and power supply between computers and electronic devices, such as embedded platforms and mobile phones.

USIM

Universal Subscriber Identity Module. An updated version of a SIM designed for use over 3G networks. USIM is able to process small applications securely using better cryptographic authentication and stronger keys. Larger memory on USIM enables the addition of thousands of contact details including subscriber information, contact details, and other custom settings.

WAP

Wireless Application Protocol. A protocol for transmitting data between a server and a client (such as a cell phone or embedded device) over a wireless network. WAP in the wireless world is analogous to HTTP in the World Wide Web.

watchdog timer

A dedicated piece of hardware or software that watches an embedded system for a fault condition by continually polling for a response. If the system goes offline and no response is received, the watchdog timer initiates a reboot procedure or takes other steps to return the system to a running state.

WMA

Wireless Messaging API. A set of classes for sending and receiving Short Message Service (SMS) messages.

XML schema

A set of rules to which an XML document must conform to be considered valid.