Oracle® Java Micro Edition Software Development Kit

Developer's Guide

Release 8.1 for Windows

E50624-02

November 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 8.1 for Windows

E50624-02

Contents

Preface

This guide describes how to use the Oracle Java Micro Edition Software Development Kit (Oracle Java ME SDK) to develop Java ME Embedded applications. You can use the standalone Oracle Java ME SDK or install plug-ins for NetBeans IDE or Eclipse IDE and take advantage of the benefits provided by the IDE.

Audience

This document is intended for developers of Java ME Embedded software who want to develop applications using Oracle Java ME SDK 8 on Windows. It assumes basic knowledge of the Java programming language and platform.

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.

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.

1

About Oracle Java ME SDK

Oracle Java ME SDK provides a set of tools for rapid development of embedded software. You can use it to write, edit, compile, package, sign, test, and debug embedded applications. Oracle Java ME SDK 8.1 supports integration with NetBeans IDE 8.0.1 and Eclipse IDE 4.4 by installing a set of plug-ins that enable all features to be used from within the popular IDE.

Java ME Embedded 8.1 is a platform for running intelligent and connected services on resource-constrained devices, such as those found in wireless modules, building and industrial controllers, smart meters, tracking systems, environmental monitors, healthcare equipment, home automation devices, vending machines, and so on. Oracle Java ME SDK 8.1 provides a complete development environment for the Java ME Embedded 8.1 platform.

Oracle Java ME SDK 8.1 includes the Java ME Embedded 8.1 runtime as a device emulation environment for Windows desktop computers. The emulation environment uses the same code base that Oracle licenses to device manufacturers for use on real devices. This enables you to perform extensive testing of your embedded applications on an emulated device before deploying them on a real device.

There are several default emulated embedded devices included with Oracle Java ME SDK. Using the Custom Device Editor, you can create a customized emulated device that mimics the target device. This enables you to start developing and testing your application without the final hardware being available.

At the heart of Oracle Java ME SDK is the Device Manager that registers all available emulated devices and connected external devices. The Device Selector can then be used to switch between target devices on which you want to run the application.

Oracle Java ME SDK provides application management functionality both through a graphical user interface (GUI) and through a command-line interface (CLI). You can use it to install, update, remove, start, and stop applications. You can view the state of applications and control the behavior of the system.

Oracle Java ME SDK can communicate with the *on-device tooling* (ODT) agent running on the Java Virtual Machine (JVM) on the device. The ODT agent provides breakpoint functionality, access to variables and data structures, and other runtime information. This enables live testing, analysis, and debugging of applications at the Java source level on the target device.

Oracle Java ME SDK enables you to emulate realistic network properties and events for testing purposes. You can use the Connectivity Emulation Tool to configure the number and type of network interfaces, switch between network modes, set wireless signal strength, and so on.

Built-in profiling and monitoring utilities enable you to investigate application performance, optimize memory use, which is critical for resource-constrained devices, and minimize the impact of limited network connectivity.

1.1 Supported Application Programming Interfaces

Oracle Java ME SDK 8.1 supports various 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.

You can download the specification documents for all JSRs on the JCP website at https://www.jcp.org.

Required Specifications

The Java ME Embedded 8.1 platform (included with Oracle Java ME SDK 8.1) is an implementation of the following specifications:

- JSR 360: Connected Limited Device Configuration 8 (CLDC 8)
- JSR 361: Java ME Embedded Profile (MEEP 8)

Optional Specifications

The following optional specifications (often referred to as *optional packages*) are also supported by Oracle Java ME SDK 8.1:

- JSR 75: PDA Optional Packages for the J2ME Platform
- JSR 120: Wireless Messaging API
- JSR 172: J2ME Web Services Specification
- JSR 177: Security and Trust Services API for J2ME
- JSR 179: Location API for J2ME
- JSR 280: XML API for Java ME

Oracle APIs

As part of Java ME Embedded 8.1, Oracle also provides the following additional APIs for embedded software development:

- Device I/O API
- JSON API
- HTTP Client API
- OAuth 2.0 API
- Configuration API
- Security API

2

Setting Up the Development Environment

Before you can start developing Java ME Embedded applications, set up the development environment.

2.1 System Requirements

There are no hardware limitations for installing and running Oracle Java ME SDK as long as all software requirements are met.

The following table lists the software requirements for installing Oracle Java ME SDK 8.1.

Table 2–1 Software Requirements for Oracle Java ME SDK 8.1

Component	Requirement
Operating System	Microsoft Windows 7 (32-bit or 64-bit)
Java Platform	Java Platform, Standard Edition (Java SE) Development Kit version 7 or 8 with the latest updates
Integrated Development Environment	■ NetBeans IDE 8.0.1 or later is required if you want to install the Oracle Java ME SDK plug-ins for NetBeans IDE ■ Eclipse IDE 4.4 or later is required if you want to install the Oracle Java ME SDK plug-ins for Eclipse IDE

2.2 Removing Previous Versions of Java ME SDK

It is possible to run several instances of Java ME SDK, but you may want to remove previous versions when installing a new one.

> **Tip:** Before removing Java ME SDK, copy any related data that you want to save to a separate folder.

1. Stop the Device Manager as follows:
   - Right-click the Device Manager icon in the notification area of the Windows taskbar, and select **Exit**.
2. Run the Java ME SDK Installation Wizard in any one of the following ways:
   - Open the Java ME SDK installation directory (by default, it is installed to the root of disk `C:`), and run `remove.exe`.
   - Open the Windows Start menu, select **All Programs**, then open the Java ME SDK folder, and click **Uninstall**.
   - Open the Windows Start menu, select **Control Panel**, then **Programs**, then **Programs and Features**, then select Java ME SDK in the list and click **Uninstall**.
3. On the first step of the wizard, click **Yes** to confirm that you want to uninstall Java ME SDK. Then follow the steps of the wizard.

2.3 Installing the Java SE Development Kit

The Java Platform, Standard Edition (Java SE) Development Kit (JDK) includes a complete Java Runtime Environment (JRE) and tools for developing, debugging, and monitoring Java applications. As an implementation of the Java SE platform, the JDK is required for alignment of Java ME Embedded features and APIs through a unified development model.

Oracle Java ME SDK 8.1 requires JDK 7 or 8 with the latest updates.

To install the JDK:

1. Download the JDK installer from the Java SE Downloads page at

   `http://www.oracle.com/technetwork/java/javase/downloads`

   > **Note:** You must accept the Oracle Technology Network (OTN) License Agreement to download this software.

2. Double-click the executable file and follow the steps of the JDK Installation Wizard.

For information about installing the JDK, refer to the *Java Platform, Standard Edition Installation Guide* at

`http://docs.oracle.com/javase/8/docs/technotes/guides/install`

2.4 Installing Oracle Java ME SDK 8.1

The base Oracle Java ME SDK 8.1 package includes a set of tools for developing Java ME Embedded applications, and the Java ME Embedded 8.1 runtime as a device emulation environment for Windows.

> **Tip:** Before installing Oracle Java ME SDK 8.1, you may want to remove any previous versions. For more information about removing previous versions of Java ME SDK, see Section 2.2.

To install Oracle Java ME SDK 8.1:

1. Make sure that JDK 7 or 8 with the latest updates is installed.

   For more information about how to find the version of Java installed, see the Java Help Center article at

   https://www.java.com/en/download/help/version_manual.xml

   For more information about installing the JDK, see Section 2.3.

2. Download the Oracle Java ME SDK 8.1 installer as an executable file at

   http://www.oracle.com/technetwork/java/javame/javamobile/download/sdk

   Note: You must accept the Oracle Technology Network (OTN) License Agreement to download this software.

3. Double-click the executable file and follow the steps of the Java ME SDK Installation Wizard.

2.5 Installing NetBeans IDE 8.0.1

NetBeans IDE is a free and open source integrated development environment (IDE) that facilitates in the development of Java applications. Oracle Java ME SDK plug-ins for NetBeans IDE enable all features to be used from within the IDE.

NetBeans IDE 8.0.1 or later is required if you want to use the Oracle Java ME SDK plug-ins.

To install NetBeans IDE 8.0.1:

1. Download the NetBeans IDE 8.0.1 installer from the downloads page at

   https://netbeans.org/downloads/

   There are several installers available, depending on the bundle that you need. The Java ME tools pack is included in the full download option, so you should download the installer under the **All** column.

2. Double-click the executable file and follow the steps of the NetBeans IDE Installer Wizard.

After you install NetBeans IDE, start it and check for updates. If updates are available, a corresponding indicator will be available in the status bar at the bottom right of the main window. To start the check manually, open the **Help** menu and select **Check for Updates**.

For detailed information about installing NetBeans IDE, refer to the installation instructions page at

https://netbeans.org/community/releases/80/install.html

2.6 Installing the Java ME SDK Plug-ins for NetBeans IDE

Plug-ins for NetBeans IDE enable all features of Oracle Java ME SDK to be used from within the IDE.

Oracle Java ME SDK 8.1 provides the following plug-ins for working with NetBeans IDE 8.0.1:

- Java ME SDK Tools: Integrates the Oracle Java ME SDK tools into the IDE.

- Java ME SDK Demos: Provides demo Java ME Embedded applications.

The Oracle Java ME SDK plug-ins for NetBeans IDE are distributed as NetBeans module (NBM) files in a ZIP archive. The NBM files are recognized by the NetBeans Plugin Manager. You can configure the Plugin Manager to automatically install the plug-ins by pointing it to the `updates.xml` file that is available in the archive.

To install the Oracle Java ME SDK plug-ins for NetBeans IDE:

1. Download the ZIP archive with the Java ME SDK plug-ins at

   `http://www.oracle.com/technetwork/java/javame/javamobile/download/sdk`

   Note: You must accept the Oracle Technology Network (OTN) License Agreement to download this software.

2. Extract the ZIP archive with the plug-ins to a directory on your computer.
3. Start NetBeans IDE, open the **Tools** menu, and select **Plugins**.
4. If you have a previous version of the Java ME SDK plug-ins installed, remove them as follows:
   a. On the **Installed** tab, select **Show Details** (if available), then select **Java ME SDK Tools** and **Java ME SDK Demos** in the list, and click **Uninstall**.
   b. Follow the steps in the NetBeans IDE Installer Wizard. On the last step of the wizard, select to restart NetBeans IDE now, and click **Finish**.
5. When NetBeans IDE restarts, open the **Tools** menu, select **Plugins**, and add an update center for Java ME SDK plug-ins as follows:
   a. On the **Settings** tab, click **Add**.
   b. In the **Name** field, enter `Java ME SDK Plug-ins Update Center`.
   c. Select **Check for updates automatically**.
   d. In the **URL** field, use the `file` uniform resource locator (URL) scheme to point to the location where you extracted the Java ME SDK plug-ins, for example:

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

   e. Click **OK**.
   f. Ensure that the **Java ME SDK Plug-ins Update Center** is active by selecting the corresponding check box in the list on the **Settings** tab.
6. Install the Java ME SDK plug-ins as follows:
   a. On the **Available Plugins** tab, select **Java ME SDK Tools** and **Java ME SDK Demos** in the list, and click **Install**. The two plug-ins are in the Java ME SDK Tools category.
   b. Follow the steps in the NetBeans IDE Installer Wizard. On the last step of the wizard, select to restart NetBeans IDE now, and click **Finish**.

Note: If the Java ME SDK plug-ins are not in the list on the **Available Plugins** tab, it is likely that you specified the wrong URL to the `updates.xml` file in the update center.

If you are not able to get the Plugins Manager to recognize the Java ME SDK plug-ins, install them manually. To install the plug-ins manually:

1. On the **Downloaded** tab, click **Add Plugins**.
2. Browse to the directory where you extracted the archive with the Java ME SDK plug-ins, select all NBM files and click **Open**.
3. Follow the steps of the NetBeans IDE Installer Wizard. On the last step of the wizard, select to restart NetBeans IDE now, and click **Finish**.

7. When NetBeans IDE restarts, open the **Tools** menu, select **Plugins**, and verify that the Java ME SDK plug-ins are active as follows:
   a. On the **Installed** tab, select **Show Details** (if available), then find the **Java ME SDK Tools** and **Java ME SDK Demos** plug-ins in the list. If they are not active, then select them and click **Activate**.
   b. When the Java ME SDK plug-ins are active, click **Close** to close the Plugins window.

If the plug-ins were installed successfully, the Java ME SDK Start Page tab should be open in NetBeans IDE. To view this tab, select **Java ME SDK Start Page** on the **Help** menu.

2.7 Installing Eclipse IDE 4.4

Eclipse IDE is a free and open source integrated development environment (IDE) that facilitates in the development of Java applications. Oracle Java ME SDK plug-ins for Eclipse IDE enable all features to be used from within the IDE.

Eclipse IDE 4.4 or later is required if you want to use the Oracle Java ME SDK plug-ins.

To install Eclipse IDE 4.4:

1. Download the ZIP file with the Eclipse IDE 4.4 package from the downloads page at

   `https://www.eclipse.org/downloads/`

   There are several packages available, depending on the bundle that you need. You should download the Eclipse Standard package.

2. Extract the downloaded ZIP file to a directory of your choice (for example, `C:\Program Files\Eclipse`). You can optionally create a shortcut to the executable file (`eclipse.exe`) in this directory.

For detailed information about installing Eclipse IDE, refer to the installation instructions page at

`https://wiki.eclipse.org/Eclipse/Installation`

2.8 Installing the Java ME SDK Plug-ins for Eclipse IDE

Plug-ins for Eclipse IDE enable all features of Oracle Java ME SDK to be used from within the IDE.

Oracle Java ME SDK 8.1 provides the following plug-ins for working with Eclipse IDE 4.4:

- Java ME SDK Tools: Integrates the Oracle Java ME SDK tools into the IDE.
- Java ME SDK Demos: Provides demo Java ME Embedded applications.

The Java ME SDK plug-ins are distributed as JAR files archived inside a ZIP file. The JAR files contain platform extensions for Eclipse IDE that are recognized by the Install New Software Wizard.

In order to install the Java ME SDK plug-ins, you need the Mobile Tools for Java (MTJ) extensions. They are also distributed as JAR files, but in a separate ZIP file. When you install the Java ME SDK plug-ins, MTJ extensions should install automatically. However, you can download and install them separately. For more information, see Installing Mobile Tools for Java Extensions.

To install the Java ME SDK plug-ins for Eclipse IDE:

1. Download the ZIP files with the Java ME SDK plug-ins from

   `http://www.oracle.com/technetwork/java/javame/javamobile/download/sdk`

   Note: You must accept the Oracle Technology Network (OTN) License Agreement to download this software.

2. Start Eclipse IDE and uninstall the previous versions of Java ME SDK plug-ins as follows:
   a. Open the **Help** menu and click **Installation Details**.
   b. On the **Installed Software** tab, select **Java ME SDK Tools**, and **Java ME SDK Demos** in the list, and click **Uninstall**.
   c. On the **Uninstall Details** window, click **Next**, then **Finish**.
   d. When prompted, click **Yes** to restart Eclipse IDE.
3. Install the Java ME SDK plug-ins as follows:
   a. Open the **Help** menu and select **Install New Software**.
   b. At the top of the **Available Software** window, click **Add**.
   c. In the **Add Repository** dialog, click **Archive**.
   d. In the file-system explorer window, browse to the ZIP file with the Java ME SDK plug-ins and click **Open**.
   e. In the **Add Repository** dialog, click **OK**.
   f. On the **Available Software** window, select **Java ME SDK Tools** and **Java ME SDK Demos** in the list, and click **Next**.
   g. On the **Install Details** window, click **Next**.
   h. Accept the terms of the license agreement and click **Finish**.
   i. When the installation process completes, restart Eclipse IDE.

When you start Eclipse IDE, to develop Java ME Embedded applications, activate the Java ME perspective as follows:

1. Open the **Window** menu, select **Open Perspective**, then **Other**.
2. In the **Open Perspective** window, select **Java ME** and click **OK**.

2.9 Installing Mobile Tools for Java Extensions

When you install the Java ME SDK plug-ins, Mobile Tools for Java (MTJ) extensions should install automatically. However, you can download and install them separately as follows:

1. Download the ZIP files with the MTJ extensions from

   `https://projects.eclipse.org/projects/tools.sequoyah.mtj/downloads`

2. Start Eclipse IDE and uninstall the previous versions of MTJ extensions as follows:
   a. Open the **Help** menu and click **Installation Details**.
   b. On the **Installed Software** tab, select **Mobile Tools for Java**, and click **Uninstall**.
   c. On the **Uninstall Details** window, click **Next**, then **Finish**.
   d. When prompted, click **Yes** to restart Eclipse IDE.
3. Install the MTJ plug-ins as follows:
   a. Open the **Help** menu and select **Install New Software**.
   b. At the top of the **Available Software** window, click **Add**.
   c. In the **Add Repository** dialog, click **Archive**.
   d. In the file-system explorer window, browse to the ZIP file with the MTJ plug-ins and click **Open**.
   e. In the **Add Repository** dialog, click **OK**.
   f. On the **Available Software** window, select **Mobile Tools for Java** in the list, and click **Next**.
   g. On the **Install Details** window, click **Next**.
   h. Accept the terms of the license agreement and click **Finish**.
   i. When the installation process completes, restart Eclipse IDE.

2.10 Updating Oracle Java ME SDK

Oracle Java ME SDK is constantly being developed. New releases may include new features, support for new APIs, fixes of known issues, and so on. You should always use the latest available version of Oracle Java ME SDK.

For minor releases, it is possible to update your instance of Oracle Java ME SDK. However, in case of a major release, you have to install the new version of Oracle Java ME SDK.

The Java ME SDK Update Center provides notifications when updates to the core Oracle Java ME SDK components, tools, or the Java ME Embedded runtime become available. This ensures that developers are working with the latest version of Oracle Java ME SDK.

The Java ME SDK Update Center is a standalone tool, but it can also be started from the NetBeans IDE or Eclipse IDE if you have the Java ME SDK plug-ins installed.

To start the standalone Java ME SDK Update Center:

- Launch `update-center.exe` under `bin` in the Oracle Java ME SDK installation directory.

To start the Java ME SDK Update Center from NetBeans IDE:

- Open the **Tools** menu, select **Java ME**, and then **Java ME SDK Update Center**.

To start the Java ME SDK Update Center from Eclipse IDE:

- Open the **Help** menu and select **Java ME SDK Update Center**.

The Java ME SDK Update Center window is separated into the following tabs:

- **Available**: Contains a list of Java ME SDK packages available on the update server that you can install.
- **Installed**: Contains a list of installed Java ME SDK packages that you can uninstall.
- **Updates**: Contains a list of installed Java ME SDK packages for which updates are available on the update server.
- **Settings**: Contains a list of update servers that you can customize.

Note: You cannot remove or edit the default Java ME SDK Update Server.

3

Installation and Configuration Directories

When using Oracle Java ME SDK, you need to understand the structure of directories that are installed. Knowing the location of specific files can help you with troubleshooting, maintenance, and advanced configuration.

You can run Oracle Java ME SDK from different user accounts on the host machine. This feature is called Multiple User Environment (MUE). MUE does not support multiple users accessing Oracle Java ME SDK simultaneously. When you switch users, you must close Oracle Java ME SDK and exit the Device Manager. A different user can then start Oracle Java ME SDK as the owner of all processes.

During installation of Oracle Java ME SDK, a user-specified location is used as the main distribution directory, where various executable, source, and documentation files reside. A separate location, which is fixed relative to the user's home directory, is used for setup, configuration, and log files.

The following sections describe the structure of each directory:

- The Oracle Java ME SDK Installation Directory Structure
- The Oracle Java ME SDK Configuration Directory Structure

3.1 The Oracle Java ME SDK Installation Directory Structure

The location of the Oracle Java ME SDK installation directory is specified by the user during the installation process. This directory is referred to as `JAVAME_SDK_HOME`. By default, Oracle Java ME SDK 8.1 is installed to the `C:\Java_ME_platform_SDK_8.1` directory.

The Oracle Java ME SDK installation directory structure conforms to the *Unified Emulator Interface (UEI) Specification* version 1.0.2 available at

`http://www.oracle.com/technetwork/java/javame/documentation/ueispecs-187994.pdf`

This structure is recognized by all IDEs and other tools that work with the UEI.

The root of the `JAVAME_SDK_HOME` directory contains the `remove.exe` file that starts the Java ME SDK Installer Wizard for removing Oracle Java ME SDK. It also contains subdirectories listed in Table 3–1.

Table 3–1 The Structure of the Oracle Java ME SDK Installation Directory

Folder	Description
`bin`	Contains executable files and dynamic-link libraries for running various Oracle Java ME SDK utilities

Table 3–1 (Cont.) The Structure of the Oracle Java ME SDK Installation Directory

Folder	Description
`docs`	Contains documentation for standard JSR APIs and Oracle APIs supported by Oracle Java ME SDK
flash	Contains runtime binaries that can be flashed to a device.
`legal`	Contains a text file with legal information about third-party software
`lib`	Contains JAR files with standard JSR APIs and Oracle APIs required for compilation of Java ME Embedded applications
`runtimes`	Contains the Java ME Embedded Profile (MEEP) runtime files
`toolkit-lib`	Contains files for configuration and definition of devices and UI elements, executable and configuration files for the Device Manager and other Oracle Java ME SDK services and utilities

3.2 The Oracle Java ME SDK Configuration Directory Structure

The location of the Oracle Java ME SDK configuration directory is relative to the user's home directory. This location is referred to as `JAVAME_SDK_USER`. For a user account named `username`, the Oracle Java ME SDK 8.1 configuration directory is located in the `C:\Users\username\javame-sdk\8.1` directory. If you delete this directory, it is re-created automatically when the Device Manager is restarted.

The root of the `JAVAME_SDK_USER` directory is used for temporary lock files and the `rmi-registry.port` file that defines the remote method invocation (RMI) registry port number. It also contains subdirectories listed in Table 3–2.

Table 3–2 The Structure of the Oracle Java ME SDK Configuration Directory

Folder	Description
`device-detection`	Contains information about the devices detected by the Device Connections Manager
`help-cache`	Contains cached Oracle Java ME SDK help pages (for NetBeans IDE plug-ins, Eclipse IDE plug-ins, and the Java ME Embedded Emulator)
`log`	Contains log files of all Oracle Java ME SDK tools (such as Device Manager, Device Selector, Custom Device Editor, and so on)
`updates`	Contains downloaded updates for Oracle Java ME SDK
`work`	Contains folders with configuration and log files for all connected devices (both emulated and real devices)

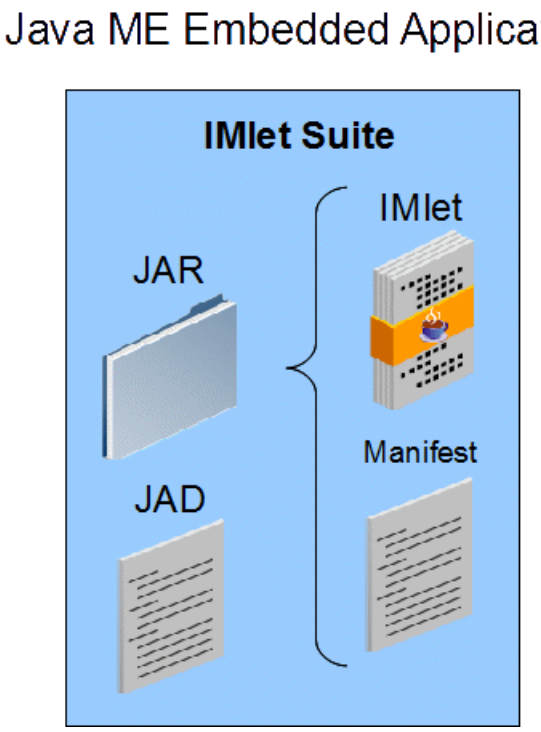
4

About Java ME Embedded Applications

Java ME Embedded applications run on small devices, with either a simple or no display at all, with low power consumption, and with limited network connectivity. Target devices include wireless modules, smart meters, industrial controllers, home automation systems, and so on.

Each Java ME Embedded application consists of an *IMlet* and other classes and resources as may be needed by the application. An IMlet is a class that extends the `javax.microedition.midlet.MIDlet` class and conforms to the Java ME Embedded Profile 8 (MEEP 8) specification. The methods of this class enable the *application management software* (AMS) on an embedded device to create, start, pause, and destroy an IMlet. Several IMlets can be packaged into a Java Archive (JAR) file along with a manifest file to form an *IMlet suite*. A Java Application Descriptor (JAD) file is used to describe the IMlet suite. Figure 4–1 shows the structure of a Java ME Embedded Application.

Figure 4–1 Structure of a Java ME Embedded Application



For more information about the `javax.microedition.midlet.MIDlet` class, see the API documentation at

http://docs.oracle.com/javame/8.0/api/meep/api/javax/microedition/midlet/MIDlet.html

> **Note:** Although an IMlet is a type of MIDlet, in the context of the Java ME Embedded platform, both terms are used interchangeably. They both refer to an application that conforms to the MEEP 8 specification.

The general procedure for developing a Java ME Embedded application includes the following steps:

1. Create an IMlet source code file that extends the `javax.microedition.midlet.MIDlet` class.
2. Compile the IMlet into a binary class file.
3. Create a manifest file.
4. Package the IMlet class and manifest into a JAR file.
5. Create a JAD file associated with the JAR.

The JAD file and the associated JAR file form an IMlet suite that can be deployed on an embedded device.

An Integrated Development Environment (IDE) automates most tasks involved in the development of applications. For instance, an IDE manages source code and resource files, parses code to highlight syntax errors, and configures the necessary settings to build, package, run, and debug an application. Source files and settings are combined by the IDE into a project.

Developing a Java ME Embedded application using an IDE includes the following steps:

1. Create a Java ME Embedded Application project.
2. Add one or more IMlet source files along with any other necessary resource files to the project.
3. Build the project.

   The IDE automatically compiles the IMlet class and other necessary classes, creates the manifest file, packages it all into a JAR file, and creates a JAD file to complete the IMlet suite.

Oracle Java ME SDK provides plug-ins for NetBeans IDE and Eclipse IDE that enable you to use all features of the SDK from within the IDE.

Creating and running a sample Java ME Embedded application is a good way to learn the basics of Oracle Java ME SDK. You can create an application either with or without an Integrated Development Environment (IDE), run it on a real connected device or an emulated device.

Example 4–1 shows the source code for a sample IMlet. You can use this source code to create a sample Java ME Embedded application.

Example 4–1 Sample IMlet Source Code (IMletDemo.java)

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

```
public class IMletDemo extends MIDlet {

    public void startApp() {
        try {
            // Add startup operations here
        } catch (Exception ex) {
            ex.printStackTrace();
            return;
        }
        System.out.println("IMletDemo is started...");
        // Add application code here
    }

    public void destroyApp(boolean unconditional) {
       // Add operations to close all resources that have been opened
    }
}
```

The following options are available for creating a sample Java ME Embedded application using Oracle Java ME SDK:

- Developing a Sample Java ME Embedded Application in NetBeans IDE
- Developing a Sample Java ME Embedded Application in Eclipse IDE
- Developing a Sample Java ME Embedded Application Without an IDE

4.1 Developing a Sample Java ME Embedded Application in NetBeans IDE

NetBeans IDE automates the process of building, packaging and running Java ME Embedded applications by providing a standard Java ME Embedded Application project. A Java ME Embedded Application project contains a fully functional template IMlet, and is configured in such a way that you only have to click one button to run it.

You can edit the provided IMlet source code, add other resource files to the project, and configure the project settings as needed. NetBeans IDE will ensure that all source files, resources, and settings are managed properly.

When you run a Java ME Embedded Application project, NetBeans IDE automatically builds, packages, and runs the application on the default emulated EmbeddedDevice1 or on another available device. You can choose to run it on another emulated device, or a real connected device.

Developing a sample Java ME Embedded application in NetBeans IDE using Oracle Java ME SDK involves the following procedures:

1. "Creating the IMletDemo Project in NetBeans IDE"
2. "Running the IMletDemo Project in NetBeans IDE"

4.1.1 Creating the IMletDemo Project in NetBeans IDE

A project combines source files and settings that are necessary to build, run, and debug an application. Without an IDE, there are a lot more manual actions required to manage all project files and settings.

NetBeans IDE includes various project types that are preconfigured for developing Java SE, JavaFX, Java ME, and many other types of applications. To create a Java ME Embedded Application project in NetBeans IDE:

1. On the **File** menu, select **New Project**.
2. On the **Choose Project** step, select **Java ME Embedded** from the **Categories** list and **Java ME Embedded Application** from the **Projects** list. Click **Next**.
3. On the **Name and Location** step, enter `IMletDemo` in the **Project Name** field. Click **Finish**.

You should see the created `IMletDemo` project in the **Projects** tab of NetBeans IDE. The `IMletDemo.java` source file from the `imletdemo` package should be open in a separate tab.

If everything is correct, you should be able to build and run the `IMletDemo` project generated from the standard Java ME Embedded Application template in NetBeans IDE. However, the application will not do anything, because methods do not contain any code.

Copy the code shown in Example 4–1 into the `IMletDemo.java` file. This will make the application print `IMletDemo is started...` to the output console when you run the application.

4.1.2 Running the IMletDemo Project in NetBeans IDE

Oracle Java ME SDK 8.1 provides the Java ME Embedded 8.1 emulation environment that enables you to duplicate (or *emulate*) an embedded device and run the application without the actual device. By default, when you run a Java ME Embedded Application project in NetBeans IDE, it is started on the emulated device EmbeddedDevice1.

To run the `IMletDemo` project, do one of the following:

- Select the `IMletDemo` project in the **Projects** tab and click the green right-arrow icon in the NetBeans IDE toolbar.
- Select the `IMletDemo` project in the **Projects** tab and press F6 on the keyboard.
- Select the `IMletDemo` project in the **Projects** tab, open the **Run** menu and select **Run Project (IMletDemo)**.
- Right-click the `IMletDemo` project name in the **Projects** tab and select **Run**.

If successful, the EmbeddedDevice1 emulator starts with the IMletDemo suite running. If you used the code from Example 4–1, you should see the following line in the **Output** tab of NetBeans IDE:

```
IMletDemo is started...
```

To open the **Output** tab in NetBeans IDE, select **Output** on the **Window** menu, or press Ctrl+4 on the keyboard.

You can run the `IMletDemo` project on a device other than the default emulated EmbeddedDevice1 (for example, another emulated device, or a real connected device). To run the `IMletDemo` project on a specific device:

1. Open the **Device Selector** tab in NetBeans IDE as follows:
   - On the **Tools** menu, select **Java ME**, and then **Device Selector**.
2. On the **Device Selector** tab, right-click the device on which you want to run the `IMletDemo` project, select **Run Project**, and then **IMletDemo**.

4.2 Developing a Sample Java ME Embedded Application in Eclipse IDE

Eclipse IDE automates the process of building, packaging and running Java ME Embedded applications by providing standard Java ME Project and Java ME MIDlet templates. A Java ME project is configured in such a way that you only have to click one button to run it.

The standard MIDlet template can be used to create the IMlet source code. You can also add other resource files to the project, and configure the project settings as needed. Eclipse IDE will ensure that all source files, resources, and settings are managed properly.

When you run a Java ME project, Eclipse IDE automatically builds, packages, and runs the application on the default emulated EmbeddedDevice1 or on another available device. You can choose to run it on another emulated device, or a real connected device.

Developing a sample Java ME Embedded application in Eclipse IDE using Oracle Java ME SDK involves the following procedures:

1. "Creating the IMletDemo Project in Eclipse IDE"
2. "Running the IMletDemo Project in Eclipse IDE"

4.2.1 Creating the IMletDemo Project in Eclipse IDE

A project combines source files and settings that are necessary to build, run, and debug an application. Without an IDE, there are a lot more manual actions required to manage all project files and settings.

To create a Java ME project in Eclipse IDE:

1. Ensure that the Java ME perspective is active as follows:
   a. Open the **Window** menu, select **Open Perspective**, then **Other**.
   b. In the **Open Perspective** window, select **Java ME** and click **OK**.
2. On the **File** menu, select **New** and then **Java ME Project**.
3. In the **New Java ME Project** dialog, enter `IMletDemo` in the **Project name** field. Click **Finish**.

   You should see the created `IMletDemo` project in the **Package Explorer** tab of Eclipse IDE.
4. Right-click the `IMletDemo` project in the **Package Explorer** tab, select **New** and then **Java ME MIDlet**.
5. In the **New Java ME MIDlet** dialog, enter `imletdemo` in the **Package** field and `IMletDemo` in the **Name** field. Click **Finish**.

   You should see the created `IMletDemo.java` source file open in a separate tab in Eclipse IDE. The source file should be located in the `imletdemo` package under `src` in the `IMletDemo` project in the **Package Explorer** tab of Eclipse IDE.

If everything is correct, you should be able to build and run the `IMletDemo` project in Eclipse IDE. However, the application will not do anything, because methods do not contain any code.

Copy the code shown in Example 4–1 into the `IMletDemo.java` file. This will make the application print `IMletDemo is started...` to the output console when you run the application.

4.2.2 Running the IMletDemo Project in Eclipse IDE

Oracle Java ME SDK 8.1 provides the Java ME Embedded 8.1 emulation environment that enables you to duplicate (or *emulate*) an embedded device and run the application without the actual device. By default, when you run a Java ME project in Eclipse IDE, it is started on the emulated device EmbeddedDevice1.

To run the `IMletDemo` project, do one of the following:

- Select the `IMletDemo` project in the **Projects** tab and press Ctrl+F11 on the keyboard.
- Select the `IMletDemo` project in the **Projects** tab, open the **Run** menu and select **Run**.
- Right-click the `IMletDemo` project name in the **Projects** tab and select **Run As**, then **Emulated Java ME JAD**.

If successful, the EmbeddedDevice1 emulator starts with the `IMletDemo` suite running. If you used the code from Example 4–1, you should see the following line in the **Console** tab of Eclipse IDE:

```
IMletDemo is started...
```

To open the **Console** tab in Eclipse IDE, open the **Window** menu and select **Show View**, then **Console**, or press Alt+Shift+Q and then C on the keyboard.

You can run the `IMletDemo` project on a device other than the default emulated EmbeddedDevice1 (for example, another emulated device, or a real connected device). To run the `IMletDemo` project on a specific device:

1. Open the **Device Selector** tab in Eclipse IDE as follows:
   - On the **Window** menu, select **Show View**, and then **Device Selector**.
2. On the **Device Selector** tab, right-click the device on which you want to run the `IMletDemo` project, select **Run Project**, and then **IMletDemo**.

4.3 Developing a Sample Java ME Embedded Application Without an IDE

An IDE automates most tasks involved in the development of applications to increase speed and efficiency. However, understanding how to develop a Java ME Embedded application without an IDE can help you realize what those tasks are, how and why they are performed.

To develop a sample Java ME Embedded application without an IDE:

1. Create a sample IMlet source code file that extends the `javax.microedition.midlet.MIDlet` class using any text editor.

   See "Creating the IMletDemo Source Code File".

2. Build the sample application by compiling the sample IMlet source code file using the `javac` command-line tool.

   See "Building the IMletDemo Class File From the Command Line".

3. Package the sample application by creating a Java Archive (JAR) file with the compiled sample IMlet class file and manifest file using the `jar` command-line tool, and a Java Application Descriptor (JAD) file with the description of the JAR file using any text editor.

   See "Packaging the IMletDemo Application From the Command Line".

4. Run the sample application by starting the Java ME Embedded Emulator using the `emulator` command-line tool with the `-Xdescriptor` command that specifies the JAD file.

   See "Running the IMletDemo Application From the Command Line".

4.3.1 Creating the IMletDemo Source Code File

An IMlet is a class that extends the `javax.microedition.midlet.MIDlet` class and conforms to the Java ME Embedded Profile 8 (MEEP 8) specification. You can use any text editor to create the IMlet source code file.

To create a sample IMlet source code file:

1. Create an empty text file using any text editor.
2. Copy the code shown in Example 4–1 into this file.
3. Save the file as `IMletDemo.java`.

4.3.2 Building the IMletDemo Class File From the Command Line

Building an application involves compiling source code files into bytecode class files. This is done using the standard Java SE Development Kit `javac` compiler.

Use the `-bootclasspath` option to specify the location of the CLDC and MEEP 8 APIs that are necessary to compile an IMlet. You can also use the `-d` option to specify where to place the compiled class files.

For example, if the `IMletDemo.java` source file is located in the `C:\meApp\src` directory, the Oracle Java ME SDK installation directory is set to the `JAVAME_SDK_HOME` environment variable, and you would like to place the compiled class files to the `C:\meApp\classes` directory, then on the Windows Command Prompt, change to the `C:\meApp` directory, and run the following command:

```
C:\meApp>%JAVA_HOME%\javac -bootclasspath %JAVAME_SDK_HOME%\lib\cldc_
1.8.jar;%JAVAME_SDK_HOME%\lib\meep_8.0.jar -d classes src\IMletDemo.java
```

As a result of this command, the following class file should be created:
`C:\meApp\classes\imletdemo\IMletDemo.class`

For more information about the `javac` compiler, see the corresponding section of the *JDK Tools Reference* at

`http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html`

4.3.3 Packaging the IMletDemo Application From the Command Line

Java ME Embedded applications are deployed as IMlet suites. An IMlet suite contains at least one IMlet class file, any number of additional resource files, and a manifest file packaged in a Java Archive (JAR). It also includes a separate Java Application Descriptor (JAD) file to describe the IMlet suite.

To package the sample IMletDemo application:

1. Create a manifest file as follows:
   a. Create an empty text file using any text editor.
   b. Copy the following text into the file:

      ```
      MIDlet-Name: IMletDemo
      MIDlet-Version: 1.0
      ```

```
MIDlet-Vendor: Company Inc.
MIDlet-1: IMletDemo,,imletdemo.IMletDemo
MicroEdition-Configuration: CLDC-1.8
MicroEdition-Profile: MEEP-8.0
```

These are the required attributes for any IMlet suite JAR manifest file as specified by the MEEP 8 specification.

c. Save the file as `manifest.mf`.

2. Create a JAR file that contains the manifest and the `IMletDemo.class` file using the standard Java SE Development Kit `jar` tool. Use the `c` option to create a new JAR, the `f` option to specify the name of the JAR, and the `m` option to specify the name of the manifest file to include.

For example, if the `IMletDemo.class` file is located in the `C:\meApp\classes\imletdemo` directory, the `manifest.mf` file is located in the `C:\meApp` directory, and you want to create the `IMletDemo.jar` file in the `C:\meApp\dist` directory, then on the Windows Command Prompt, change to the `C:\meApp` directory, and run the following command:

```
C:\meApp>%JAVA_HOME%\jar cfm dist\IMletDemo.jar manifest.mf -C classes .
```

As a result of this command, the following JAR file should be created: `C:\meApp\dist\IMletDemo.jar`

For more information about the `jar` tool, see the corresponding section of the *JDK Tools Reference* at

http://docs.oracle.com/javase/8/docs/technotes/tools/windows/jar.html

3. Create a JAD file as follows:

a. Create an empty text file using any text editor.

b. Copy the following text into the file:

```
MIDlet-Name: IMletDemo
MIDlet-Version: 1.0
MIDlet-Vendor: Company Inc.
MIDlet-1: IMletDemo,,imletdemo.IMletDemo
MIDlet-Jar-Size: 933
MIDlet-Jar-URL: IMletDemo.jar
```

These are the required attributes for any IMlet suite JAD file as specified by the MEEP 8 specification.

Note: You must set the `MIDlet-Jar-Size` attribute value to the size of the JAR file in bytes.

---

c. Save the file as `IMletDemo.jad` to the `C:\meApp\dist` directory.

4.3.4 Running the IMletDemo Application From the Command Line

Oracle Java ME SDK 8.1 provides the Java ME Embedded 8.1 emulation environment that enables you to duplicate (or *emulate*) an embedded device and run the application without the actual device using the Java ME Embedded Emulator.

The Java ME Embedded Emulator executable (`emulator.exe`) is located under `bin` in the Oracle Java ME SDK installation directory `JAVAME_SDK_HOME`.

You should use the `-Xdescriptor` command to specify the location of the JAD file associated with the Java ME Embedded application (IMlet suite) that you want to run. By default, the application starts on the EmbeddedDevice1 emulator, unless you use the `-Xdevice` option that enables you to specify the name of the emulated device on which you want to run the application.

For example, if you want to run the IMletDemo application packaged in the `IMletDemo.jar` file with the `IMletDemo.jad` file in the `C:\meApp\dist` directory, then on the Windows Command Prompt, change to the `bin` directory in the Oracle Java ME SDK installation directory (`JAVAME_SDK_HOME`), and run the following command:

```
JAVAME_SDK_HOME>emulator -Xdescriptor:C:\meApp\dist\IMletDemo.jad
```

If successful, the EmbeddedDevice1 emulator starts with the IMletDemo suite running.

5

About Java ME Embedded Application Projects

An Integrated Development Environment (IDE) uses projects to combine source files and settings that are necessary to build, run, and debug applications. Without an IDE, there are a lot more manual actions required to manage all the files and settings.

The development and configuration of an application in the IDE takes place in the context of the project. It is the highest level of organization for the application that you are developing.

When you create a project in an IDE, it generates an Ant script to build the application. Alternatively, IDEs also support Maven. For more information about Ant and Maven, see their respective official web sites:

- http://ant.apache.org/
- http://maven.apache.org/

For more information about NetBeans IDE projects, see *Developing Applications with NetBeans IDE* at

http://docs.oracle.com/cd/E50453_01/doc.80/e50452/toc.htm

For more information about Eclipse IDE projects, see *Eclipse Documentation* at

http://help.eclipse.org

5.1 Managing Java ME Embedded Application Projects in NetBeans IDE

NetBeans IDE includes various project types that are preconfigured for developing Java SE, JavaFX, Java ME, and many other types of applications. Each type includes template source files and settings that are specific to the development platform.

You can initially define some of the more important settings when the project is created. Other settings are preconfigured with default values, however, you can change them at any time.

To create a Java ME Embedded Application project in NetBeans IDE:

1. On the **File** menu, select **New Project**.
2. On the **Choose Project** step, select **Java ME Embedded** from the **Categories** list and **Java ME Embedded Application** from the **Projects** list. Click **Next**.
3. On the **Name and Location** step, specify initial settings as necessary and click **Finish**.

When you create a project in NetBeans IDE, you can view it in one of two ways:

- The **Projects** tab provides a logical view of the project

- The **Files** tab provides a physical view of the project

To rename, move, copy, or delete an existing project in NetBeans IDE, right-click the project on the **Projects** tab, and select **Rename**, **Move**, **Copy**, or **Delete**.

To add an IMlet to a Java ME Embedded Application project:

1. Right-click the project on the **Projects** tab, select **New**, then **MIDlet**.
2. On the **Name and Location** step of the New MIDlet window, specify the name and location as necessary and click **Finish**.

To configure the settings of an existing project in NetBeans IDE, right-click the project on the **Projects** tab, and select **Properties**.

5.1.1 Managing Java ME Embedded Application Project Sources in NetBeans IDE

When creating a project in NetBeans IDE, you specify the location and name of the project folder. By default, the `src` folder is created inside the project folder for all source packages. You can add other folders with source code files and packages that you want to be part of the project.

To manage the project sources in NetBeans IDE, right-click the project, select **Properties**, and then open the **Sources** category.

Managing the List of Source Package Folders

The sources for a project are maintained in the form of a list of folders with source code files and packages. To add a folder with sources that you want to be part of the project, click **Add Folder**. To remove a folder, select it in the list and click **Remove**. If there are multiple folders used by a project, you can define the order in which folders are processed (that is, source files are compiled) using the **Move Up** and **Move Down** buttons.

For each source package folder, you can define a label that is displayed in the **Projects** view. For example, the default `src` folder is labeled `Source Packages`. By default, all added folders are labeled with the name of the folder. To change a label, double-click it, enter a name, and press Enter.

Setting the Source and Target Versions

If you are developing an application that must be compatible with previous versions of Java, you can set the version of the source code that the compiler should expect, and the target runtime version for which you want to compile. To set the source and target version, select it in the **Source/Binary Format** drop-down list under the list of source package folders. This setting defines the `-source` and `-target` options of the `javac` Java compiler.

Setting the Encoding of Source Files

If development occurs in multiple countries, encoding of source files in projects may vary. You need to make sure that the Java compiler knows the encoding. To set the encoding of source files in a project, select it in the **Encoding** drop-down list under the list of source package folders. This setting defines the `-encoding` option of the `javac` Java compiler.

Including and Excluding Source Files

The defined source package folders may contain files that you want to exclude from the project. To configure which files to included and exclude, click the **Includes/Excludes** button under the list of source package folders. Specify regular

expressions to filter out files you want to include and files you want to exclude. Check the lists of included and excluded files based on the regular expressions to make sure that the expressions are correct and click **OK**.

Related Topics

For more information about the `javac` Java compiler, see the *Java Platform, Standard Edition Tools Reference* at

`http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html`

5.1.2 Selecting Java ME Embedded Application Project Platform in NetBeans IDE

The Java SE Development Kit (JDK) is used to compile and package a Java ME Embedded application, while the Java ME Embedded platform included with Oracle Java ME SDK provides an emulation environment for Windows to run the application on an emulated device.

You can select both the JDK and the Java ME platform for a Java ME Embedded Application project when creating the project. To select the JDK and the Java ME platform for an existing Java ME Embedded Application project, right-click the project, select **Properties**, and then open the **Platform** category.

The **JDK Path** and **Java ME Platform** drop-down lists contain only those JDK versions and Java ME platforms that are registered with NetBeans IDE.

Managing Platforms in NetBeans IDE

When you install NetBeans IDE, it automatically detects and registers all Java platforms on your computer. You can use the Java Platform Manager to manually register other platforms in NetBeans IDE as necessary. To access the Java Platform Manager do one of the following:

- On the **Tools** menu, select **Java Platforms**.
- When creating a project, click **Manage Platforms**.
- In the **Platform** category of the **Project Properties** window for an existing project, click **Manage Platforms**.

Configuring the Emulation Environment

The Java ME platform includes an emulation environment that provides implementations of the device's APIs. For example, Oracle Java ME SDK 8.1 includes the Java ME Embedded 8.1 runtime as an emulation environment for Windows.

To select the default device on which to run the Java ME Embedded Application project, use the **Device** drop-down list. In case of Oracle Java ME SDK 8.1, `EmbeddedDevice1` is selected by default.

Because Java ME Embedded 8.1 implements only JSR 360: Connected Limited Device Configuration 8 (CLDC 8) and JSR 361: Java ME Embedded Profile (MEEP 8), the **Configuration** and **Profile** options do not provide any alternatives.

Various devices may implement optional application programming interfaces (APIs) to provide specific functionality (for example, wireless communication or physical location tracking). By default, all optional packages available to the platform are selected for a project. However, if a device does not support some of the APIs, you can exclude corresponding packages to reduce the size of the application.

5.1.3 Managing Java ME Embedded Application Project Libraries in NetBeans IDE

A project may depend on classes, associated source files, annotation processors, and Javadoc documentation from another project, library, Java Archive (JAR), or any other location. These dependencies (also known as *libraries*) are added to the class path so that they can be accessed during compilation. The list of libraries defines the `-classpath`, `-sourcepath`, and `-processorpath` options of the `javac` Java compiler.

To manage these libraries for an existing Java ME Embedded Application project, right-click the project, select **Properties**, and then open the **Libraries** category. The following tabs are available in the **Libraries** category:

- **Compile**: This tab is used to manage the list of compile-time libraries that define the location of general dependencies required during compilation. These are propagated to other library types.
- **Processor**: This tab is used to manage the list of processor-path libraries that define the location of the annotation processors used in the project. If no libraries are specified or the processor is not available in the specified libraries, then the general compile-time class path is searched for annotation processors.
- **LIBlets**: This tab is used to manage the list of shareable software components that a Java ME Embedded application may use at runtime. To customize LIBlet-specific options in this tab, add LIBlets of type `liblet` as compile-time libraries on the **Compile** tab.

A project can have dependencies in another project, in a library, in a JAR file, or in a folder. To add the dependency, click the corresponding button (**Add Project**, **Add Library**, or **Add JAR/Folder**).

To edit a library, select it in the list and click **Edit**. To remove a library, select it in the list and click **Remove**. If there are multiple libraries on which a project depends, you can define the order in which they are searched for the corresponding dependencies using the **Move Up** and **Move Down** buttons.

If there are source code files in the libraries that are associated with a project, they have to be built to be used. The **Build Projects on Classpath** check box is selected by default to enable all project dependencies to be built if they are on the class path. If you know that your project does not require any of the source files in the dependent libraries, you can deselect this check box to decrease the time of compilation.

Related Topics

For more information about the `javac` Java compiler, see the *Java Platform, Standard Edition Tools Reference* at

`http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html`

5.1.4 Managing Java ME Embedded Application Descriptor Attributes in NetBeans IDE

The basic components of a Java ME Embedded application are the Java Application Descriptor (JAD) file and the Java Archive (JAR) file. Together, these two files form an IMlet suite. Application descriptor attributes define metadata that represents the application's properties and configuration.

Attributes are contained in a JAD file, and include information, such as the name, vendor, and version of the IMlet suite, the location and size of the JAR file, and the configuration and profile requirements. The JAD file may contain any number of attributes defined by JSR 361: Java ME Embedded Profile (MEEP 8), as well as custom attributes defined by the developer of the application.

A manifest file contained in a JAR file has the same syntax as the JAD file and it may share the same attributes. Attributes in the JAD file must agree with those in the manifest file.

For more information about the application descriptor attributes, see the Application Packaging document of the *Java ME Embedded Profile Specification* at

`http://docs.oracle.com/javame/config/cldc/opt-pkgs/api/meep/api/doc-files/packaging.html`

To manage application descriptor attributes for an existing Java ME Embedded Application project in NetBeans IDE, right-click the project, select **Properties**, and then open the **Application Descriptor** category. The following tabs are available in the **Application Descriptor** category:

- **Attributes**: This tab is used to manage the list of name and value pairs for general attributes. If you are creating a Java ME Embedded application, select **MIDlet Suite** as the packaging model. If you are creating a shareable software component that an application will use during runtime, select **LIBlet** as the packaging model.

  There are three default attributes defined that are required and cannot be removed from the list. For a MIDlet, the default attributes are `MIDlet-Name`, `MIDlet-Vendor`, and `MIDlet-Version`. For a LIBlet, the default attributes are `LIBlet-Name`, `LIBlet-Vendor`, and `LIBlet-Version`.

  By default, the JAR file is located in the same folder as the JAD file. To specify a different location for the JAR file that is specified in the JAD file, select **Override JAR URL in JAD** under the list of general attributes, and enter an absolute or relative URL that will be used as the value for the `MIDlet-Jar-URL` or `LIBlet-Jar-URL` attribute.

  Caution: Attributes beginning with `MIDlet-`, `LIBlet-`, or `MicroEdition-` are reserved for use by the *application management software* (AMS) on the device. Do not use these for custom user-defined attributes!

- **MIDlets**: This tab is used to manage the list of MIDlets in the suite, with a class name, a displayed name, and a displayed icon for each. At least one entry is necessary. By default, it is the main IMlet class.

  The first entry in the list defines the `MIDlet-1` attribute, the second entry defines the `MIDlet-2` attribute, and so on. You can change the order using the **Move Up** and **Move Down** buttons.

- **Push Registry**: This tab is used to manage the list of MIDlets that are registered for push notifications, with a class name, an IP address of the sender, and a connection string that identifies the protocol and port number. When you install your application on a device, the *application management software* (AMS) listens for incoming connections from MIDlets specified in the push registry. If a registered MIDlet connects from a matching IP address over the specified protocol to the specified port number, the AMS launches the application.

  The first entry in the list defines the `MIDlet-Push-1` attribute, the second entry defines the `MIDlet-Push-2` attribute, and so on. You can change the order using the **Move Up** and **Move Down** buttons.

  For more information about the push registry, see the Javadoc for the `javax.microedition.io.PushRegistry` class in the *Java ME Embedded Profile Specification* at

http://docs.oracle.com/javame/config/cldc/opt-pkgs/api/meep/api/javax/microedition/io/PushRegistry.html

> **Note:** To use the push registry in your application, you must set the `javax.microedition.io.PushRegistryPermission` attribute on the **API Permissions** tab.

- **API Permissions**: This tab is used to manage the list of permission attributes for protected APIs that the application uses. When you install your application on a device, the AMS compares the permissions requested with the permissions in the destination protection domain. If a required permission is denied, the installation terminates and an exception is returned. If an optional permission is denied, the application may install, but will run with limited functionality.

  For a MIDlet, the first required permission entry in the list defines the `MIDlet-Permission-1` attribute, the second required permission entry defines the `MIDlet-Permission-2` attribute, and so on. Optional permission entries define the `MIDlet-Permission-Opt-1` attribute, the `MIDlet-Permission-Opt-2` attribute, and so on. For a LIBlet, corresponding attributes begin with the `LIBlet-Permission-1` attribute and the `LIBlet-Permission-Opt-1` attribute.

  For more information about permissions, see the Security for Applications document of the *Java ME Embedded Profile Specification* at

  http://docs.oracle.com/javame/config/cldc/opt-pkgs/api/meep/api/doc-files/security_framework.html

On each tab, to add an attribute, click **Add**, provide the necessary data, and click **OK**. To edit an attribute, select it and click **Edit**. To remove an attribute, select it and click **Remove**.

5.1.5 Configuring Java Compiler Settings in NetBeans IDE

When building a Java ME Embedded Application project, NetBeans IDE automatically compiles the main IMlet class and any other necessary classes. To do this, NetBeans IDE uses the `javac` Java compiler from the Java SE Development Kit (JDK).

The `javac` command has many options that allows you to configure how the Java compiler produces bytecode class files. These options can be configured in the NetBeans IDE project properties. To configure Java compiler settings for an existing Java ME Embedded Application project in NetBeans IDE, right-click the project, select **Properties**, and then open the **Compiling** category.

Generating Debugging Information

The Java compiler can generate debugging information into the output class files. This information can then be used by debugging tools during run time.

By default, a project is configured to generate all debugging information, which is defined by the `-g` option of the `javac` command. This includes information about line numbers, source files, and local variables. However, once your application is fully debugged, you should recompile it without any debugging information to make the class files smaller and harder to reverse engineer. To build your project without any debugging information, deselect **Generate Debugging Info**. This sets the `-g:none` option for the `javac` command.

Reporting Deprecated API Usage

As Java classes are updated, their APIs change. New methods, constructors, and fields are added, existing ones can sometimes be renamed for consistency. Some classes and interfaces can replace existing ones when a better approach is found.

Java supports a deprecation mechanism to let the developers know when an API they are using is deprecated. To deprecate a class, method, or member field, an annotation is added to it, as well as a Javadoc tag with comments. The comment is generated in the Javadoc for the API, warning the user and suggesting alternatives. The annotation causes the `javac` Java compiler to produce a warning, although existing calls to deprecated APIs continue to work, and classes are still compiled.

To see the exact class, method, or member field that is deprecated, select **Report Uses of Deprecated APIs**. This behavior is defined by the `-deprecation` option of the `javac` command. Without this option, the Java compiler shows only a summary of the source file names that use or override deprecated classes, methods, or fields.

Tracking Java Dependencies

If your project depends on external classes, it is important to track modifications to the dependencies. The ability to track this is a feature of the build system used by NetBeans IDE.

To automatically recompile any class in your project that depends on a class that has been modified, select **Track Java Dependencies**. This ensures that the latest version of any project dependency is used by NetBeans IDE when you build and run the project.

Processing Annotations

Annotations are metadata in the source code that provide information about the code and do not affect the operation of the application. Some annotations are used by the Java compiler to detect errors or suppress warnings. Other annotations are processed at the beginning of compilation to generate additional source code files, XML files, and so on. And there are certain annotations that are accessible at run time.

To enable annotation processing during compilation, select **Enable Annotation Processing**. This is the default behavior of the Java compiler. If you deselect this option, annotations will not be processed, which is defined by the `-proc:none` option of the `javac` command. To see the results of annotation processing directly in the Java Editor in NetBeans IDE, select **Enable Annotation Processing in Editor**.

You can specify custom annotation processors that you want to use for building your project in the **Annotation Processors** list. To add a processor, click **Add** next to the list, enter the fully qualified name (FQN) of the processor, and click **OK**. This list defines the `-processor` option of the `javac` command.

If the annotation processor associated with your project excepts command-line options, you can specify the ones that you want to pass to it in the **Processor Options** list. To add an option, click **Add** next to the list, enter a key and its value, and click **OK**. This list defines the `-A` option of the `javac` command. It is specified in the form **`-A`***`key`*`[=`*`value`*`]`.

Other Java Compiler Settings

To configure additional Java compiler settings, enter the corresponding `javac` command options in the **Additional Compiler Options** field. Use the exact syntax that you would use when entering them after the `javac` command.

Related Topics

For more information about the `javac` Java compiler, see the *Java Platform, Standard Edition Tools Reference* at

`http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html`

5.1.6 Signing Java ME Embedded Applications in NetBeans IDE

Signing a Java ME Embedded application allows MEEP devices to verify the integrity and origin of the IMlet suite. Signing information is used to check an application's source and validity before allowing it to access certain protected APIs. The certificate that is used to sign your application designates a security domain that defines the permitted protected APIs.

You should sign your applications for security reasons, to protect them from being tampered by malicious third parties, and to increase their acceptance by distribution channels. Security aware vendors always have more credibility in the industry.

To enable signing for an existing Java ME Embedded Application project in NetBeans IDE, right-click the project, select **Properties**, open the **Signing** category, and select **Sign JAR**.

By default, the built-in keystore provided by Oracle Java ME SDK is selected, with the `minimal` certificate, which denies all permissions to protected APIs. The certificates in the built-in keystore can be used for internal testing purposes. When you are ready to distribute your application, you should buy a signing key pair from a reputable certificate authority (CA) and use it to sign the application.

To be able to select a non-default key pair with which to sign your application, import it to an existing keystore or create a new keystore. This can be done in the Keystores Manager. To open the Keystores Manager window, click **Open Keystores Manager**.

Besides selecting the certificate for signing your application, you have to make sure that the corresponding key is registered on the device. To export the selected key to a specific device registered with Oracle Java ME SDK:

1. In the **Signing** category of the project properties, click **Export Key into Java ME SDK Platform Emulator**.

   Alternatively, you can click **Export** in the Keystores Manager window.
2. Select the device and one of the security clients available on the device.
3. Click **Export**.

5.1.7 Obfuscating Java ME Embedded Applications in NetBeans IDE

Obfuscation refers to deliberately making program code harder to understand, decompile, and reverse-engineer. Obfuscators are programs that transform readable code into obfuscated code.

Oracle Java ME SDK includes ProGuard, which is a Java bytecode obfuscator. It first shrinks, optimizes, and preverifies Java class files to make them more compact. Then it transforms the bytecode to make it almost impossible to reverse-engineer. This is an important security measure, because raw bytecode produced by the Java compiler contains much of the source code information, which is your intellectual property. Size optimization is also crucial for Java ME Embedded applications, which are designed for resource-constrained devices.

To enable obfuscation for an existing Java ME Embedded Application project in NetBeans IDE, right-click the project, select **Properties**, open the **Obfuscating** category, and click **Install ProGuard Obfuscator**.

When ProGuard is installed, select the level of obfuscation using the **Obfuscation Level** slider. You can see the impact of the selected obfuscation level on bytecode in the **Level Description** field. This field also lists the arguments passed to ProGuard when it is launched. To configure additional settings for ProGuard, specify the corresponding options in the **Additional Obfuscation Settings** field.

For more information about command-line options of ProGuard, see the *ProGuard Reference Card* at

`http://proguard.sourceforge.net/#manual/refcard.html`

5.1.8 Configuring Project Documentation Settings in NetBeans IDE

For each of your applications, you can produce a set of Javadoc HTML pages that describe the project's classes, inner classes, interfaces, constructors, methods, and fields. The Javadoc is constructed from the structure of your code and the Javadoc comments embedded in your code. To do this, NetBeans IDE uses the `javadoc` tool from the Java SE Development Kit (JDK).

The `javadoc` command has many options that allows you to configure how Javadoc files are produced, what information they include, and so on. These options can be configured in the NetBeans IDE project properties. To configure Javadoc settings for an existing Java ME Embedded Application project in NetBeans IDE, right-click the project, select **Properties**, and then open the **Documenting** category.

By default, Javadoc generates documentation only for protected and public classes and members. If you want to document all classes and members, including those with the private and package-private access levels, select **Include Private and Package Private Members**. This behavior is defined by the `-private` option of the `javadoc` command.

Javadoc can generate some additional pages to aid with navigation. By default, NetBeans IDE is configured to generate the class hierarchy tree pages, the class and package usage pages, the navigation bar, and index with a separate page for each letter.

To omit the class hierarchy tree pages, deselect **Class Hierarchy Tree**. This starts the `javadoc` tool with the `-notree` option.

To omit the usage pages, deselect **Class and Package Usage Pages**. This removes the `-use` option from the `javadoc` command.

To omit the navigation bar, header and footer, deselect **Navigation Bar**. This starts the `javadoc` tool with the `-nonavbar` option.

To omit the index, deselect **Index**. This starts the `javadoc` tool with the `-noindex` option.

To generate the index on a single page, deselect **Separate Index per Letter**. This removes the `-splitindex` option from the `javadoc` command.

By default, Javadoc does not process the `@author` and `@version` tags. To include information about the author or version, select the corresponding check box under **Document Additional Tags**. This is defined by the `-tag` option of the `javadoc` command.

To set the browser window title (that is, the `<title>` tag in the HTML code of the generated pages), enter it in the **Browser Window Title** field. This is defined by the `-windowtitle` option of the `javadoc` command. If the window title is not set explicitly,

the value of the `-doctitle` option is used, which is by default set to the name of the NetBeans IDE project.

To configure additional Javadoc tool settings, enter the corresponding `javadoc` command options in the **Additional Javadoc Options** field. Use the exact syntax that you would use when entering them after the `javadoc` command.

To open the generated Javadoc in your default browser after completion, select **Preview Generated Javadoc**.

Related Topics

For more information about the `javadoc` tool, see the *Java Platform, Standard Edition Tools Reference* at

`http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html`

5.1.9 Configuring Java ME Embedded Emulator Settings in NetBeans IDE

After the project is built, you can use NetBeans IDE to run your Java ME Embedded application. The Java ME Embedded Emulator is used to install and start the IMlet suite on the device.

To configure settings for running your Java ME Embedded Application project in NetBeans IDE, right-click the project, select **Properties**, and then open the **Run** category.

To configure the Java ME Embedded Emulator settings, enter the corresponding options in the **Emulator Command Line Options** field. Use the exact syntax that you would use when entering them after the `emulator` command.

By default, **Regular Execution** is selected under **Run Method**. This option means that the IMlet suite is executed on the device. To simulate the process of deploying the application from a server to a remote device, select **Execute through OTA**. This option means that only the JAD file is executed and the JAR is provisioned *over the air* (OTA).

Note: When debugging, the application is always executed regularly, no OTA is used.

To set the delay for which NetBeans IDE should wait before attaching a debugger, enter a value in milliseconds in the **Debugger timeout** field. By default, this value is `30000`.

5.2 Managing Java ME Projects in Eclipse IDE

Eclipse IDE with Mobile Tools for Java (MTJ) extensions includes a project type that is preconfigured for developing Java ME Embedded applications. The project defines settings that are specific to Java ME Embedded.

You can initially define some of the more important settings when the project is created. Other settings are preconfigured with default values, however, you can change them at any time.

To create a Java ME project in Eclipse IDE:

1. Ensure that the Java ME perspective is active as follows:
   a. Open the **Window** menu, select **Open Perspective**, then **Other**.
   b. In the **Open Perspective** window, select **Java ME** and click **OK**.

2. On the **File** menu, select **New** and then **Java ME Project**.
3. In the **New Java ME Project** dialog, specify initial settings as necessary. You can click **Next** and **Back** to navigate the wizard. When you are done, click **Finish**.

When you create a project in Eclipse IDE, you can view it in one of two ways:

- The **Package Explorer** tab provides a logical view of the project
- The **Navigator** tab provides a physical view of the project

To add an IMlet to a Java ME Embedded Application project:

1. Right-click the project on the **Package Explorer** tab, select **New**, then **Java ME MIDlet**.
2. In the **New Java ME MIDlet** dialog, specify initial settings as necessary. You can click **Next** and **Back** to navigate the wizard. When you are done, click **Finish**.

It is also possible to import an existing project into Eclipse IDE as follows:

1. Open the **File** menu and select **Import**.
2. In the list, expand the **General** node and select **Existing Project into Workspace**. Click **Next**.
3. Select the root directory of the project or an archive if you previously saved your project into an archive.
4. Select the project name in the list and click **Finish**.

To configure the settings of an existing project in Eclipse IDE, right-click the project on the **Package Explorer** tab, and select **Properties**.

5.2.1 Managing Java ME Project Device Configurations in Eclipse IDE

The Java ME Embedded platform included with Oracle Java ME SDK provides an emulation environment for Windows to run Java ME Embedded applications on an emulated device.

To manage the device configurations on which you want to run the project:

1. Right-click the project on the **Package Explorer** tab, and select **Properties**.
2. In the left pane of the Properties window, select the **Java ME** category.

The **Configurations** list contains a list of devices that are available to the project.

To add a device to the **Configurations** list:

1. Click **Add**.
2. Select the Java ME SDK platform and device that you want to add.
3. Optional: Specify a name for the configuration. By default, the name of the device is used.
4. Click **Finish** to add the configuration to the project.

To edit the configuration, select it in the list and click **Edit**. To remove a configuration from the project, select it in the list and click **Remove**.

Below the configuration list, you can specify names of generated JAR and JAD files.

5.2.2 Performing Code Validation for a Java ME Project in Eclipse IDE

Code validation is the process of checking that the source code of your application complies with standards and recommendations for Java. Clean sources can help you maintain your code base efficiently.

Code validation settings in Eclipse IDE are configured globally (for all projects), but you can also override some of the settings for a specific project. To configure global code validation settings for all projects in Eclipse IDE:

1. Open the **Window** menu and select **Preferences**.
2. In the left pane of the Preferences window, expand the **Java ME** category and select **Code Validation**.
3. Select the language constructs that you would like to be warned about or that you want to ignore when compiling.

To configure project-specific code validation settings for an existing Java ME project in Eclipse IDE:

1. Right-click the project on the **Package Explorer** tab, and select **Properties**.
2. In the left pane of the Properties window, select the **Code Validation** category under **Java ME**.
3. Select **Enable project specific settings**.
4. Select the language constructs that you would like to be warned about or that you want to ignore when compiling.

5.2.3 Managing Java ME Project Libraries in Eclipse IDE

A project may depend on external libraries of classes and source files. These dependencies are added to the class path so that they can be accessed during compilation and included into the JAR file. The list of libraries defines the `-classpath` option of the `javac` Java compiler.

To manage these libraries for an existing Java ME project in Eclipse IDE:

1. Right-click the project on the **Package Explorer** tab, and select **Properties**.
2. In the left pane of the Properties window, select the **Library** category under **Java ME**.
3. Select the libraries to be included from the list of available libraries.

Java ME Embedded applications may use LIBlets, which are a special type of shareable software components used at runtime. To add LIBlets to a Java ME project in Eclipse IDE:

1. Right-click the project on the **Package Explorer** tab, and select **Properties**.
2. In the left pane of the Properties window, expand the **Java Build Path** category and select **LIBlets**.
3. Click **Add** and configure the LIBlets that you want to add to the project build path.

5.2.4 Obfuscating Java ME Embedded Applications in Eclipse IDE

Obfuscation refers to deliberately making program code harder to understand, decompile, and reverse-engineer. Obfuscators are programs that transform readable code into obfuscated code.

Oracle Java ME SDK includes ProGuard, which is a Java bytecode obfuscator. It first shrinks, optimizes, and preverifies Java class files to make them more compact. Then it transforms the bytecode to make it almost impossible to reverse-engineer. This is an important security measure, because raw bytecode produced by the Java compiler contains much of the source code information, which is your intellectual property. Size optimization is also crucial for Java ME Embedded applications, which are designed for resource-constrained devices.

Obfuscation settings in Eclipse IDE are configured globally (for all projects), but you can also override some of the settings for a specific project. To configure global obfuscation settings for all projects in Eclipse IDE:

1. Open the **Window** menu and select **Preferences**.
2. In the left pane of the Preferences window, expand the **Java ME** category, then expand **Packaging**, and select **Obfuscation**.
3. If you want to set command-line options for running ProGuard, select **Use specified arguments** and enter the options in the field that follows the check box.
4. You can use the **Proguard Keep Expressions** list to define the list of classes and class members to be preserved as entry points to the application. By default, all public classes that extend the `javax.microedition.midlet.MIDlet` class are preserved from obfuscation. Click **Add** to add more regular expressions to the list. Double-click existing expressions to modify them. Select an expression and click **Remove** to remove it from the list.

To configure project-specific obfuscation settings for an existing Java ME project in Eclipse IDE:

1. Right-click the project on the **Package Explorer** tab, and select **Properties**.
2. In the left pane of the Properties window, select the **Obfuscation** category under **Java ME**.
3. Select **Enable project specific settings**.
4. If you want to set command-line options for running ProGuard, select **Use specified arguments** and enter the options in the field that follows the check box.
5. You can use the **Proguard Keep Expressions** list to define the list of classes and class members to be preserved as entry points to the application. By default, all public classes that extend the `javax.microedition.midlet.MIDlet` class are preserved from obfuscation. Click **Add** to add more regular expressions to the list. Double-click existing expressions to modify them. Select an expression and click **Remove** to remove it from the list.

For more information about command-line options of ProGuard, see the *ProGuard Reference Card* at

`http://proguard.sourceforge.net/#manual/refcard.html`

5.2.5 Setting Java ME Project Packaging Attributes in Eclipse IDE

Java ME Embedded applications are packaged into JAR files with corresponding descriptor JAD files. The manifest in the JAR and the JAD file contain attributes that describe the application.

Packaging settings in Eclipse IDE are configured globally (for all projects), but you can also override some of the settings for a specific project. To configure global packaging settings for all projects in Eclipse IDE:

1. Open the **Window** menu and select **Preferences**.

2. In the left pane of the Preferences window, expand the **Java ME** category and select **Packaging**.
3. If you want the version to automatically increment every time you build a project, select **Increment Version Automatically**. This option sets the `MIDlet-Version` attribute (or `LIBlet-Version` if you are building a LIBlet).
4. You can use the **Excluded Manifest Entries** list to define the list of attributes that should be excluded from the manifest. By default, the following attributes are excluded:
   - `MIDlet-Jar-URL`: Points to the location of the JAR file.
   - `MIDlet-Jar-Size`: Specifies the size of the JAR file.
   - `LIBlet-Jar-URL`: Points to the location of the JAR file.
   - `LIBlet-Jar-Size`: Specifies the size of the JAR file.

   Click **Add** to add more attributes to the list. Double-click existing attributes to modify them. Select an attribute and click **Remove** to remove it from the list.

To configure project-specific packaging attributes for an existing Java ME project in Eclipse IDE:

1. Right-click the project on the **Package Explorer** tab, and select **Properties**.
2. In the left pane of the Properties window, select the **Packaging** category under **Java ME**.
3. Select **Enable project specific settings**.
4. If you want the version to automatically increment every time you build the project, select **Increment Version Automatically**. This option sets the `MIDlet-Version` attribute (or `LIBlet-Version` if you are building a LIBlet).
5. You can use the **Excluded Manifest Entries** list to define the list of attributes that should be excluded from the manifest. By default, the following attributes are excluded:
   - `MIDlet-Jar-URL`: Points to the location of the JAR file.
   - `MIDlet-Jar-Size`: Specifies the size of the JAR file.
   - `LIBlet-Jar-URL`: Points to the location of the JAR file.
   - `LIBlet-Jar-Size`: Specifies the size of the JAR file.

   Click **Add** to add more attributes to the list. Double-click existing attributes to modify them. Select an attribute and click **Remove** to remove it from the list.

5.2.6 Configuring Source Code Preprocessor for Java ME Projects in Eclipse IDE

Preprocessing enables you to perform build-time source code transformations to deal with device fragmentation. Devices have different constraints and inconsistencies that require a developer to use preprocessing rather than create an application that can deal with this at runtime.

Preprocessor settings in Eclipse IDE are configured globally (for all projects), but you can also override some of the settings for a specific project. To configure global preprocessor settings for all projects in Eclipse IDE:

1. Open the **Window** menu and select **Preferences**.
2. In the left pane of the Preferences window, expand the **Java ME** category and select **Preprocessor**.

3. Select the debug level for the preprocessor to include during build time.

To configure project-specific preprocessor settings for an existing Java ME project in Eclipse IDE:

1. Right-click the project on the **Package Explorer** tab, and select **Properties**.
2. In the left pane of the Properties window, select the **Preprocessor** category under **Java ME**.
3. Select **Enable project specific settings**.
4. Select the debug level for the preprocessor to include during build time.

5.2.7 Signing Java ME Embedded Applications in Eclipse IDE

Signing a Java ME Embedded application allows MEEP devices to verify the integrity and origin of the IMlet suite. Signing information is used to check an application's source and validity before allowing it to access certain protected APIs. The certificate that is used to sign your application designates a security domain that defines the permitted protected APIs.

You should sign your applications for security reasons, to protect them from being tampered by malicious third parties, and to increase their acceptance by distribution channels. Security aware vendors always have more credibility in the industry.

Signing in Eclipse IDE is configured globally (for all projects), but you can also override some of the settings for a specific project. To configure global signing settings for all projects in Eclipse IDE:

1. Open the **Window** menu and select **Preferences**.
2. In the left pane of the Preferences window, expand the **Java ME** category and select **Signing**.
3. Specify the location of the key store.
4. Select whether you want to enter the password when it is required or save it in the current Eclipse IDE workspace keyring.
5. To create a new key pair in the specified key store, click **Create New Key Pair** to the right of the **Key Aliases** list.

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

   To import a certificate from a certificate authority (CA), click **Import Certificate**.

   To import a response to your certificate signing request (CSR), click **Import CSR Response**.
6. If you do not want to use Java defaults, configure additional settings under **Advanced Settings**.

To configure project-specific signing settings for an existing Java ME project in Eclipse IDE:

1. Right-click the project on the **Package Explorer** tab, and select **Properties**.
2. In the left pane of the Properties window, select the **Signing** category under **Java ME**.
3. Select **Enable project specific settings**.
4. Specify the location of the key store.

5. Select whether you want to enter the password when it is required, save it in the current Eclipse IDE workspace keyring, or save it as part of the project.
6. To create a new key pair in the specified key store, click **Create New Key Pair** to the right of the **Key Aliases** list.

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

   To import a certificate from a certificate authority (CA), click **Import Certificate**.

   To import a response to your certificate signing request (CSR), click **Import CSR Response**.
7. If you do not want to use Java defaults, configure additional settings under **Advanced Settings**.

6

Debugging Java ME Embedded Applications

Stable operation of an application depends on the ability to avoid as many errors during development as possible. You have to be able to detect errors and deal with them before your application is released.

Debugging in general refers to a complex set of methods for extensive diagnostics and troubleshooting of applications both at the source code level and during runtime. Besides interactive debugging, this may include analyzing log files, monitoring the application at runtime, and collecting profiling information.

The Java Platform, Standard Edition Development Kit (JDK) provides various diagnostic and monitoring tools, such as the `jdb` Java Debugger. There are also tools that are specific to various operating systems (OS), such as the `userdump` utility on Windows and `dbx` on Solaris.

For more information about debugging Java applications, see the *Java Platform, Standard Edition Troubleshooting Guide* at

`http://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot`

Oracle Java ME SDK maintains several logs and includes dedicated monitoring and profiling tools specifically for Java ME Embedded applications. You can also use general Java debugging and profiling features, as well as those provided by the IDE and the OS.

6.1 Interactive Debugging

Interactive debugging is accomplished by setting breakpoints and watchpoints in the source code and running the application with a debugger. This enables you to execute your code one line at a time and examine the state of your application to discover problems.

By default, both NetBeans IDE and Eclipse IDE use the Java Debugger based on the Java Platform Debugger Architecture (JPDA). Before debugging a project, connect the device on which you want to perform debugging and select it as the default for this project.

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

To debug a project in Eclipse IDE, select the project name in the **Package Explorer** view, open the **Run** menu, and select **Debug**.

6.2 Profiling Java ME Embedded Applications

Profiling refers to collecting information about the runtime behavior of an application. Oracle Java ME SDK provides a dedicated CPU profiler that keeps track of every method in your Java ME Embedded application.

As data is collected during the emulation session, the profiler figures out how much time was spent in each method. After you close the emulator, you can export the data to an NPS file (from NetBeans IDE) or to a PRF file (from Eclipse IDE) that you can load and view later. As you view the snapshot, you can investigate particular methods, classes, and packages, and save a customized snapshot (as a PNG file) for future reference.

Note: The profiling values obtained from the emulator do not reflect the actual values on an external device, because the emulator does not represent the actual device, it is one possible implementation of its supported APIs.

Before profiling a project, connect the device on which you want to perform profiling and select it as the default for this project.

Note: The profiler collects large amounts of data, so profiled IMlets require a larger heap to run on the device. To increase the heap size, configure the `Heapsize` property for the device. For more information about device properties, see Viewing and Editing Device Properties.

To profile a Java ME Embedded Application project in NetBeans IDE:

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

   When the application stops, profiling data is displayed in a tab labeled **cpu** with the time when the data was displayed.

To profile a Java ME project in Eclipse IDE:

1. In the **Package Explorer** view, select the project.
2. Open the **Run** menu and select **Profile**.
3. Select **CPU Profiler** and click **OK**.

   When the application stops, profiling data is displayed in a tab labeled **CPU** with the time and date when the data was collected.

In NetBeans IDE, to export the profile data to an NPS file, click the **Export to** button in the CPU Profiler tab and specify the file name and location. This data can be loaded at a later time.

In Eclipse IDE, to export the profile data to a PRF file, open the **File** menu and select **Save As**, then specify the file name and location. This data can be loaded at a later time.

6.3 Monitoring Memory Usage of Java ME Embedded Applications

The Java ME SDK plug-ins include a Memory Monitor that shows memory usage of a running Java ME Embedded application from within the IDE. It displays a dynamic detailed listing of the current memory usage per object in table form, and a graphical representation of the memory usage over time.

You can take a snapshot of the Memory Monitor data (as an MMS file, short for *memory monitor snapshot*). Snapshots can be loaded and analyzed later.

> **Note:** The memory usage values obtained from the emulator do not reflect the actual values on an external device, because the emulator does not represent the actual device, it is one possible implementation of its supported APIs.

To monitor memory usage of a Java ME Embedded Application project in NetBeans IDE:

1. In the **Projects** view, right-click the project 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 **Memory Monitor** and click **Run**.

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

To monitor memory usage of a Java ME project in Eclipse IDE:

1. In the **Package Explorer** view, select the project name.
2. Open the **Run** menu and select **Memory Monitor**.

   The **Memory Monitor** view opens in the main working area of Eclipse 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 this values over time. 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 heap memory on the device.

To sort the data in the object table, click a column header. Sorting is case sensitive.

To display the call stack tree, click a row in the object table. The call stack is displayed in a window to the right of the table. You can browse the call stack tree to see the methods that created the object by selecting the corresponding folder in the call stack.

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

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.

To load a memory monitor snapshot.

1. On the **File** menu, select **Open File**.

2. Browse to find the necessary MMS file.

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

6.4 Monitoring Network Activity of Java ME Embedded Applications

The Java ME SDK plug-ins include a Network Monitor that provides a convenient way to see the data that your Java ME Embedded 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 through HTTP and other protocols.

You can take a snapshot of the Network Monitor data (as an NMD file, short for *network monitor data*). Snapshots can be loaded and analyzed later.

To monitor network activity of a Java ME Embedded Application project in NetBeans IDE:

1. In the Projects view, right-click the project 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 **Network Monitor** and click **Run**.

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

To monitor network activity of a Java ME project in Eclipse IDE:

1. In the **Package Explorer** view, select the project name.
2. Open the **Run** menu and select **Network Monitor**.

   The **Network Monitor** view opens in the main working area of Eclipse IDE.

When the application makes a network connection, information about the connection is captured and displayed in the table in the top frame. 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.

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

To save your network monitor session, open the **File** menu and select **Save As**, then specify an NMD file name and location.

To load a network monitor session:

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

Filtering and Sorting Connections

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 list contains protocols that are available 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 sequentially (according to the time when they were started).
- **Protocol**: Connections are sorted by protocol name.
- **Device**: Connections are sorted by device name.
- **URL**: Connections are sorted by URL.
- **Time**: Connections are sorted by duration.
- **Size**: Connections are sorted by the size of data.
- **Active**: Connections are sorted based on whether it is currently active.

6.5 Logging Capabilities Provided by Oracle Java ME SDK

Oracle Java ME SDK relies on a set of tools that help you with development, such as the Device Manager, Device Selector, Custom Device Editor, and so on. Each tool maintains a separate log file to record errors, warnings, and informational events. When you experience problems with any of the tools, you can review its log file to understand which events caused a particular issue.

These log files are located in the Oracle Java ME SDK user configuration directory under `log`. For example, the Device Manager saves logging data to `JAVAME_SDK_USER\log\device-manager.log`.

The Java ME Embedded Emulator maintains a separate log file for each device. The device log is named `device.log` and is located in the device's configuration directory. For example, the logging data for the default EmbeddedDevice1 is saved to `JAVAME_SDK_USER\work\EmbeddedDevice1\device.log`.

For more information about the Oracle Java ME SDK user configuration directory, see Chapter 3.

By default, device logs record high-level program usage events. Debugging may require lower-level information about the operation of the underlying runtime. Recording this tracing data in a device's log file can be enabled using the device properties.

To enable tracing for a device:

1. Open the Device Selector and select the device. For more information about the Device Selector and device properties, see Chapter 7.
2. Expand the **Monitor** node of the device properties and select the desired trace properties:
   - **Trace GC**: Tracing garbage collection information can help you determine object health. The garbage collector cannot delete objects that do not have a null reference. Null objects are garbage collected and not reported as active.
   - **Trace Class Loading**: Tracing class initialization and loading is useful for determining dependencies among classes.
   - **Trace Exceptions**: Tracing caught exceptions can be used for debugging when analyzing the log file.
   - **Trace Method Calls**: Tracing 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 may cause your application to run out of memory on the device. To increase the device's heap memory size, expand the **General** node of the device properties and specify the value for the `Heapsize` property.

7

About Java ME Embedded Devices

Java ME Embedded applications run on small devices, with either a simple or no display at all, with low power consumption, and with limited network connectivity. Target devices include wireless modules, smart meters, industrial controllers, home automation systems, and so on.

Oracle Java ME SDK includes the Java ME Embedded runtime as a device emulation environment for Windows desktop computers. The emulation environment uses the same code base that Oracle licenses to device manufacturers for use on real devices. This enables you to perform extensive testing of your embedded applications on an emulated device before deploying them on a real device.

By default, Oracle Java ME SDK includes two generic emulated devices (`EmbeddedDevice1` and `EmbeddedDevice2`), and an emulation of the Qualcomm Internet-of-Everything (IoE) embedded device (`Qualcomm_IoE_Device`). For more information about the Qualcomm IoE device, see *Oracle Java ME Embedded Getting Started Guide for the Reference Platform (Qualcomm IoE)*.

You can also use Oracle Java ME SDK to connect a real external device and run your Java ME Embedded application on it. In this case, the Java ME Embedded Emulator provides an implementation of the device's APIs for testing, logging, and debugging using the tools available with Oracle Java ME SDK.

7.1 Managing Devices

During development, you may be required to run your Java ME Embedded application on several emulated and real devices. Oracle Java ME SDK includes the Device Manager to facilitate you with managing all the various devices.

The Device Manager is a Windows process named `device-manager.exe` that starts automatically and always runs in the background after you install Oracle Java ME SDK. The Device Manager automatically detects and registers all available emulated devices. It also allows you to connect and register external devices (see Section 7.2, "Connecting an External Device").

When the Device Manager is running, you can see its icon in the notification area of the Windows taskbar. To stop the Device Manager, right-click its icon and select **Exit**. To start the Device Manager manually, run the `device-manager.exe` file under `bin` in the Oracle Java ME SDK installation directory.

To see the list of registered devices, right-click the Device Manager icon and select **Registered Devices**. This list contains details about emulated and real external devices that were registered by the Device Manager and are available to Oracle Java ME SDK.

If you have multiple instances of Oracle Java ME SDK installed, you should see multiple Device Manager processes running.

When working with Java ME SDK plug-ins for NetBeans IDE, you can switch between the Device Manager processes to define the instance of Oracle Java ME SDK that should be used. To select the active Device Manager process in NetBeans IDE, open the **Tools** menu, select **Java ME**, and then make your choice under **Active Device Manager**.

To define the active instance of Oracle Java ME SDK in Eclipse IDE:

1. Open the **Window** menu and select **Preferences**.
2. In the left pane, select **Device Management** under **Java ME**.

7.2 Connecting an External Device

The Device Manager can detect and register external embedded devices with an active Java ME Embedded runtime on specific IP addresses and serial ports. Device connections are managed in the Device Connections Manager window.

To open the Device Connections Manager window, right click the Device Manager icon in the notification area of the Windows taskbar and select **Manage Device Connections**. The Device Connections Manager window contains a list of IP addresses and a COM ports that the Device Manager scans to detect and register available devices. By default, there are no IP addresses or COM ports in the list.

To add a connection that you would like the Device Manager to scan, click **Add** at the bottom of the Device Connections Manager window. You can add a connection by IP address, host name, or COM port. If there are no serial ports on you computer, then the **COM Port** option is not available. You can enter an IP address, host name or COM port explicitly, or select it from the corresponding drop-down list. The drop-down lists contain IP addresses on your network and COM ports where the Device Manager detected a running instance of the Java ME Embedded runtime. To rescan the network and COM ports, click **Refresh**. After you enter or select an address or COM port, click **OK** to add it to the list.

All connections in the Device Connections Manager list have one of the following statuses:

- `Connecting`: Means that the Device Manager is attempting to establish a connection and register the device with Oracle Java ME SDK. This is the initial status for any newly added connection.
- `Connected`: Means that the Device Manager registered the device with Oracle Java ME SDK. If there are several instances of the Java ME Embedded runtime started on the device, each one is added as a separate device.
- `Busy`: Means that the device is already registered with another Oracle Java ME SDK.
- `Wrong Runtime`: Means that the device is running a version of the Java ME Embedded runtime that is not supported by the current version of Oracle Java ME SDK.

To remove a connection from the list in the Device Connections Manager window, select it and click **Remove**.

7.2.1 Troubleshooting Device Connection Issues

If the IP address of a device with a running Java ME Embedded runtime instance is not available in the corresponding drop-down list when adding a device connection,

see the Device Manager log file. It is located under `logs` in the Oracle Java ME SDK configuration directory.

The Device Connection log file (`device-manager.log`) contains errors, warnings, and informational events that you can review in order to find the cause of the problem. The following are some of the common messages that you may encounter:

WARN - .vmagent.proxy.DeviceDetection - UDP device detection failed
java.net.BindException: Address already in use: Cannot bind

Cause: The device detection ports are used by another application on the host computer. By default, these ports are 55208 and 55209.

Action: The best solution is to stop the application that uses these ports or configure it to use different ports.

Alternatively, you can configure the device and Device Manager to use different ports as follows:

1. Change the ports specified by the `proxy.udp_device_detection_request_port` and `proxy.udp_device_detection_response_port` properties in `jwc_properties.ini` on the device.
2. Create a file named `proxyOptions.txt` under `toolkit-lib/lib` in the Oracle Java ME SDK installation directory and add the following line to it:

   ```
   -bcastports <request> <response>
   ```

   The `<request>` and `<response>` port numbers must match those specified in the device properties (see Step 1).

7.3 Creating and Managing Custom Emulated Devices

By default, Oracle Java ME SDK includes two generic emulated devices (`EmbeddedDevice1` and `EmbeddedDevice2`), and an emulation of the Qualcomm Internet-of-Everything (IoE) embedded device (`Qualcomm_IoE_Device`). You can use the Custom Device Editor to create and manage custom emulated devices.

The appearance of a custom emulated device is generic, but the functionality can be configured according to your specifications. Each emulated device should have a name and optionally a description. By default, Oracle Java ME SDK 8.1 supports devices based on the MEEP 8.0 profile and the CLDC 1.8 configuration. You can define the set of optional packages that the emulated device should support, as well as various protocols and interfaces that the device should emulate (for example, the pins and ports, channels, pulse counters, and other features).

To start the Custom Device Editor, run the `device-editor.exe` file under `bin` in the Oracle Java ME SDK installation directory.

Alternatively, if you have the Java ME SDK plug-ins installed, you can start the Custom Device Editor in the IDE.

To start the Custom Device Editor in NetBeans IDE:

- Open the **Tools** menu, select **Java ME**, and then **Custom Device Editor**.

To start the Custom Device Editor in Eclipse IDE:

- Open the **Run** menu and select **Custom Device Editor**.

The Custom Device Editor contains a list of custom emulated devices that are available, separated by platform. Oracle Java ME SDK 8.1 supports only the Java ME

Embedded Profile (MEEP) platform. The three default emulated devices cannot be customized by the user, so initially the list is empty.

The Custom Device Editor maintains configurations of custom devices, referred to as *device skins*. Each skin is a directory with files, which you can export into a ZIP file for backup or if you need to transfer it to another instance of Oracle Java ME SDK. The Custom Device Editor can then be used to import the ZIP file with the skin to create the custom device defined by the skin.

> **Caution:** You should manage custom device skins only using the Custom Device Editor. Do not change the configuration directly in the files.

To create a new custom device, select the platform and click **New**.

To edit an existing custom device, select it and click **Edit**.

To create a new custom device by cloning an existing one, select it and click **Clone**.

To remove a custom device, select it and click **Remove**.

To import a custom device defined by its skin, click **Import** and select the corresponding ZIP file.

To export a custom device skin as a ZIP file, select the device and click **Export**.

7.4 Viewing and Editing Device Properties

When the Device Manager registers a device, it appears in the Device Selector, which enables you to view and edit the device properties. You can also use the Device Selector to configure the security settings for an emulated device, and select the device on which to run your Java ME Embedded application.

To start the Device Selector, run the `device-selector.exe` file under `bin` in the Oracle Java ME SDK installation directory. Alternatively, if you have the Java ME SDK plug-ins installed, you can start the Device Selector as follows:

- In NetBeans IDE, open the **Tools** menu, select **Java ME**, and then **Device Selector**.
- In Eclipse IDE, open the **Window** menu, select **Show View**, and then **Device Selector**.

> **Caution:** You should edit device properties only using the Device Selector. Do not change them directly in the files, where properties are contained.

The Device Selector contains a list of devices that were registered by the Device Manager, grouped by platform. When you select a device, its information and properties are displayed in the corresponding panes.

> **Tip:** You can also view and edit some of the platform 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.

8

About the Java ME Embedded Emulator

The Java ME Embedded Emulator provides an implementation of the APIs supported by an emulated or real device that is registered with Oracle Java ME SDK. This enables you to perform extensive testing, logging, and debugging of Java ME Embedded applications.

The Java ME Embedded Emulator uses the Java ME Embedded runtime as a device emulation environment for Windows desktop computers. The emulation environment has the same code base that Oracle licenses to device manufacturers for use on real devices.

By default, Oracle Java ME SDK includes two generic emulated devices (`EmbeddedDevice1` and `EmbeddedDevice2`), and an emulation of the Qualcomm Internet-of-Everything (IoE) embedded device (`Qualcomm_IoE_Device`). For more information about the Qualcomm IoE device, see *Oracle Java ME Embedded Getting Started Guide for the Reference Platform (Qualcomm IoE)*.

Running the Java ME Embedded Emulator

When you run a Java ME Embedded application using Oracle Java ME SDK, the Java ME Embedded Emulator is started automatically. To start the Java ME Embedded Emulator manually, run `emulator.exe` with the `-Xjam` option under `bin` in the Java ME SDK installation directory.

The title of the Java ME Embedded Emulator window shows the name of the device that is being emulated. By default, the Java ME Embedded Emulator runs `EmbeddedDevice1`. You can specify a different device to run using the Device Selector or by specifying it in the `-Xdevice` option for `emulator.exe`.

For more information about running the Java ME Embedded Emulator from the command line, see Appendix A.

When developing a Java ME Embedded application in an IDE, you can select the default device on which you want to run the project.

For more information about selecting the default device for a project in NetBeans IDE, see Section 5.1.2, "Selecting Java ME Embedded Application Project Platform in NetBeans IDE".

For more information about selecting the default device for a project in Eclipse IDE, see Section 5.2.1, "Managing Java ME Project Device Configurations in Eclipse IDE".

Understanding the Main Window

The appearance of the main window of the Java ME Embedded Emulator can vary depending on the protocols, interfaces, and various features that the device supports. At a minimum, every Java ME Embedded device includes the *application management*

software (AMS) as part of the runtime. It is represented in the Java ME Embedded Emulator window by the **AMS** tab, and enables you to install, run, and manage IMlet suites. You can also view the output console of the device and the log that the Java ME Embedded Emulator maintains. Other devices may include various pins and ports, channels, pulse counters, and other features represented by separate tabs:

- **AMS**: Used to install, run, and manage IMlet suites on the emulated device.
- **GPIO Pins**: Used to view the state of the General Purpose Input/Output (GPIO) pins.
- **GPIO Ports**: Used to view the state of the GPIO ports, and which pins are bound to each port.
- **I2C**: Used to view the data sent to and received from a peripheral serial slave device through the Inter-Integrated Circuit (I2C) bus.
- **SPI**: Used to view the data sent to and received from a peripheral serial slave device through the Serial Peripheral Interface (SPI) bus.
- **MMIO**: Used to view the memory configuration and memory content for a peripheral device connected through the Memory-Mapped Input/Output (MMIO) interface bus.
- **ADC**: Used to view the signals sent to the Analog-to-Digital Converter (ADC) through the specified channel.
- **DAC**: Used to view the signals sent to the Digital-to-Analog Converter (DAC) through the specified channel.
- **PWM**: Used to view the signals sent to the Pulse Width Modulation (PWM) output signal through the specified channel.
- **Pulse Counters**: Used to view the state of the pulse counters.
- **Displays and Input Devices**: Used to view information about displays and headless input devices attached to your emulated device, including both primary and auxiliary displays.

For some devices you can view the message inbox, manage landmarks, the file system, and network interfaces, and generate external events (for example, simulate various sensor readings).

To force the Java ME Embedded Emulator window to always show on top of the others, click the **Emulator window always on top** button on the Java ME Embedded Emulator window toolbar, or select **Always on top** on the **View** menu.

8.1 Installing and Running IMlet Suites Using the Java ME Embedded Emulator

Java ME Embedded applications are distributed as IMlet suites that consist of a JAR file described by a JAD file. When you run a Java ME Embedded application using Oracle Java ME SDK, the corresponding IMlet suite is automatically installed and started on the emulated device.

When the Java ME Embedded Emulator is running, you can also install and run IMlet suites manually. One device can have several IMlet suites installed and running at the same time. To manage IMlet suites for a device, open the **AMS** tab.

To install an IMlet suite:

1. On the **AMS** tab, click **Install**.

2. Specify the path or URL to the corresponding JAR or JAD file, or click **Browse** to select it in the file explorer window.
3. Click **OK** and wait for the IMlet suite to install.

   The installed IMlet appears in the list with the status `Not Running`.

To reinstall an IMlet suite, select it in the list and click **Reload**.

To remove an IMlet, select it in the list and click **Remove**.

To view the manifest file information about the IMlet, select it in the list and click **Info**.

To run an installed IMlet:

1. Select it in the list and click **Run**.
2. Select the options for debugging, profiling, and monitoring if necessary.
3. Click **OK**.

   The IMlet status changes to `Running`.

To automatically install and run an IMlet suite:

1. Click the **Run IMlet Suite** button on the Java ME Embedded Emulator window toolbar, or select **Run IMlet Suite** on the **Application** menu.
2. Specify the path or URL to the corresponding JAR or JAD file, or click **Browse** to select it in the file explorer window.
3. Select the options for debugging, profiling, and monitoring if necessary.
4. Click **OK** and wait for the IMlet suite to install and start.

   The installed IMlet appears in the list with the status `Running`.

To stop a running IMlet, select it in the list and click **Stop**.

8.2 Viewing Device Output and Logs

The Java ME Embedded Emulator maintains a separate log file for each device to record errors, warnings, and informational events. A running Java ME Embedded application may write data to the standard output stream (`stdout`) or standard error stream (`stderr`). This information can be used for troubleshooting and debugging.

To view the device log file, open the **View** menu and select **Device Log**. You can select the level of detail that you would like to view among the following:

- **Trace**: Displays all logged events, including low-level tracing data.
- **Debug**: Displays all high-level events, including debugging data.
- **Info**: Displays high-level informational events and more severe ones.
- **Warn**: Displays high-level warning events that may possibly indicate a problem and more severe ones.
- **Error**: Displays high-level events that indicate an error, including fatal ones.
- **Fatal**: Displays only high-level fatal error events.

To save the output to a LOG file, click **Save**. To clear the log file, click **Clear**.

To view the device output console, open the **View** menu and select **Output Console**. You can select the streams that you would like to view among the following:

- **All**: Displays all console messages produced by the application.

- **Standard Output**: Displays messages printed to the `stdout` stream.
- **Standard Error**: Displays messages printed to the `stderr` stream.

To save the output to a LOG file, click **Save**. To clear the console output produced by the device, click **Clear**.

8.3 Viewing Messages

Some devices support wireless communication through JSR 120: Wireless Messaging API (WMA), which enables Java ME Embedded applications to send and receive messages. Any messages that are not addressed to a specific application on the device are stored in the inbox.

To see the messages in the inbox using the Java ME Embedded Emulator, open the **Device** menu and select **Messages**. The Messages window contains a list of messages in the inbox with the type, sender, timestamp, subject, encoding, and contents of each message.

8.4 Managing Landmarks

Some devices support location tracking through JSR 179: Location API for J2ME, which enables Java ME Embedded applications to use information about the physical location of the device. Such devices also maintain a database of landmarks that applications may rely on for guidance.

A landmark is identified by a name, and may be placed in a category that groups landmarks by type (for example, shops, restaurants, gas stations, and so on). A landmark can have a description, and it must define the location using the latitude and longitude. You can also define the altitude and accuracy of positioning. A landmark may contain data about the street address, phone number, URL, and so on.

To manage the landmarks that are known to the device in Java ME Embedded Emulator, open the **Tools** menu and select **Manage Landmarks**. Landmarks are stored in a landmark store. You can use the default landmark store or add several named stores.

Note: Although you may create separate landmark stores for separate applications, all landmark stores are always available to all applications on the device.

To add a new landmark store, in the **Landmark Stores** drop-down list, select **Add new landmark store**. To manage available landmark stores (add new ones or remove existing ones), click **Manage Landmark Stores** next to the drop-down list.

To add a category to the selected landmark store, click **Add Category**. To remove a category, select it and click **Remove Category**.

Note: Category names must be unique.

By default, the list of landmarks contains all available landmarks in the selected landmark store. To show only landmarks from certain categories, select the corresponding check boxes in the categories list.

To add a landmark, click **Add**. To edit a landmark, select it and click **Edit**. To remove a landmark, select it and click **Remove**. To assign a landmark to categories, select it and click **Assign Categories**.

Note: Landmarks can have similar names, coordinates, and other information.

When you select a landmark, you can preview the defined location information in the pane below the landmarks list.

8.5 Managing the File System

Some devices can interact with the file system resources using the `FileConnection` package which is part of JSR 75: PDA Optional Packages for the J2ME Platform. The `FileConnection` package gives Java ME Embedded applications access to the file system of the device, including external memory cards.

The Java ME Embedded Emulator enables IMlets to access files stored on your computer's hard disk as if they are in the device's storage or attached memory card. To manage the device's file system, open the **Tools** menu and select **Manage File System**. Mounted root directories are displayed in the top list, and unmounted root directories are displayed in the bottom list. Mounted root directories and their subdirectories are available to applications that implement the `FileConnection` interface.

Mounted root directories are located in the device's configuration folder of the Oracle Java ME SDK user configuration directory. For example, the root directories for `EmbeddedDevice1` are located under

```
JAVAME_SDK_USER\work\EmbeddedDevice1\appdb\filesystem
```

The default root directory folder is named `root1`. You cannot unmount the default root directory.

To create a new empty root directory, click **Mount Empty**, specify a name and click **OK**. A new folder is created in the device's `filesystem` folder and it is mounted as another root directory in addition to the default.

To create a root directory by copying an existing folder on your computer, click **Mount Copy** and select the necessary folder. This folder and its subfolders is copied to the device's `filesystem` folder and it is mounted as another root directory in addition to the default.

To make a directory inaccessible to the `FileConnection` API, select it in the list of mounted root directories and click **Unmount**. The selected root directory is unmounted and moved to the bottom list. To completely remove a mounted directory, select it and click **Unmount & Delete**.

To remount an unmounted root directory, select it in the list of unmounted root directories and click **Remount**. The root directory is moved to the top list.

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

8.6 Managing the Connectivity Configuration

Some devices can communicate over a wired, wireless, or cellular network. They can establish and accept connections to other devices, servers, and so on. There are several APIs that are used to provide this functionality to Java ME Embedded applications.

The Java ME Embedded Emulator enables you to set up the connectivity configuration of the running device, which you would like to emulate. To manage the connectivity configuration, open the **Tools** menu and select **Connectivity**. The Connectivity window is separated into several tabs.

Managing Access Points

An access point is a network connectivity configuration of a TCP/IP network interface that is defined on a device using the `javax.microedition.io.AccessPoint` class.

Use the **Access Points** tab of the Connectivity window to view and manage available access points. This tab contains a table that lists access points with the following columns:

- **Id**: Numeric identifier of the access point
- **Access Point Name**: Name of the access point
- **Connected**: Whether the access point is connected to a network
- **Default**: Whether this is the default access point

When you select an access point in the table, you can view and configure additional settings for the access point. Additional settings are displayed in the bottom panel of the **Access Points** tab. For any access point you can specify a name and select a network interface from the drop-down list. Other settings depend on the selected network interface (for example, the SSID for a WiFi access point).

Click **Add Access Point** below the table of access points to add a new access point. To remove an access point, select it in the table and click **Remove Access Point**.

Managing Network Interfaces

A network interface represents the network connection spot for an access point. It is defined on a device using the `javax.microedition.io.NetworkInterface` class.

Use the **Network Interfaces** tab of the Connectivity window to view and manage available network interfaces. This tab contains a table that lists network interfaces with the following columns:

- **Id**: Numeric identifier of the network interface
- **Network Interface Name**: Name of the network interface
- **IP Address**: Address of the device in the IP network
- **Connected**: Whether the interface is connected to a network

When you select a network interface in the table, you can view and configure additional settings for the network interface. Additional settings are displayed in the bottom panel of the **Network Interfaces** tab. For any network interface you can specify a name, select a type from the drop-down list, and map it to any physical network interface on the host machine.

Click **Add Network Interface** below the table of network interfaces to add a new network interface. To remove a network interface, select it in the table and click **Remove Network Interface**.

Managing Cellular Networks

A cellular network is a 3GPP or CDMA network that the device is registered on. It is defined on a device using the `javax.microedition.cellular.CellularNetwork` class.

Use the **Cellular Networks** tab of the Connectivity window to view and manage available cellular network. This tab contains a table that lists cellular networks with the following columns:

- **Id**: Numeric identifier of the cellular network
- **Cellular Network Name**: Name of the cellular network

When you select a cellular network in the table, you can view and configure additional settings for the cellular network. Additional settings are displayed in the bottom panel of the **Cellular Networks** tab. For any cellular network you can specify a name, select a network interface from the drop-down list, toggle roaming on and off, and so on.

Click **Add Cellular Network** below the table of cellular networks to add a new cellular network. To remove a cellular network, select it in the table and click **Remove Cellular Network**.

Managing Subscribers

A subscriber of a cellular network is represented by an identity, such as SIM, R-UIM, CSIM, and so on. Subscribers resides on the device in a physical or virtual slot exposed as a subscriber identifier. The primary subscriber always resides in slot number 1. A subscriber is defined on a device using the `javax.microedition.cellular.Subscriber` class.

Use the **Subscribers** tab of the Connectivity window to view and manage available subscribers. This tab contains a table that lists subscribers with the following columns:

- **Slot Number**: Numeric identifier of the subscriber slot number
- **Phone Number**: Phone number of the subscriber
- **Registered Cellular Network**: The cellular network of the subscriber

When you select a subscriber in the table, you can view and configure additional settings for the subscriber. Additional settings are displayed in the bottom panel of the **Subscribers** tab. For any subscriber you can specify the operator, phone number, and select the type of cellular network from the drop-down list. Other settings depend on the selected cellular network.

Click **Add Subscriber** below the table of subscribers to add a new subscriber. To remove a subscriber, select it in the table and click **Remove Subscriber**.

8.7 Generating External Events

Java ME Embedded devices are designed to interact with various external events. These events can include power management signals, location information, pulses, and so on. The Java ME Embedded Emulator enables you to simulate such events using the External Events Generator to see how your Java ME Embedded application processes them.

To access the External Events Generator, open the **Tools** menu and select **External Events Generator**. Alternatively, you can click the **External Events Generator** button on the Java ME Embedded Emulator main window toolbar. The External Events Generator window is separated into several tabs, depending on the configuration and capabilities of the device.

Generating Analog Input

Some devices can read analog input signals and convert them to numerical values using the Analog-to-Digital Converter (ADC). An ADC can have several channels, each one sampling a separate continuous input voltage.

Use the **ADC** tab to specify the input voltage for an ADC channel that is configured on the emulated device. Select the ADC channel from the drop-down list. Each ADC pin is denoted by a pin number and a channel number (for example, `ADC1Ch1`).

Specify the input voltage for the selected ADC channel by entering the value in the text field or using the slider.

Alternatively, you can set the output of an existing Digital-to-Analog Converter (DAC) channel to be used as the input voltage for the selected ADC channel by selecting **DAC Input** and using the drop-down list to choose the DAC channel.

Generating Button Events

Some devices have General-Purpose Input/Output (GPIO) pins that may be used as additional digital control lines. They can be controlled and programmed at runtime, and they are generally used for connecting buttons and light-emitting diodes (LEDs).

Use the **GPIO** tab to generate input events for the GPIO pins that are configured on the emulated device. For each input pin, you can toggle the input value between **Low** and **High** by clicking the corresponding button. This corresponds to pressing the hardware analogy of the button connected to the pin, toggling it on and off.

Under **Wave Generator**, you can set the frequency (in hertz) and duration (in milliseconds) of a more complex signal. To run and stop the wave generator for an individual pin, supply frequency and duration values, and use the corresponding **Run** and **Stop** buttons. You can also use the **Run All** or **Stop All** buttons to run or stop all pins simultaneously.

Generating Input From Emulated Peripheral Devices

Some devices support Inter-Integrated Circuit (I2C) and Serial Peripheral Interface (SPI) communication with peripheral devices, such as various sensors. To emulate a peripheral slave device on either of these bus links, there are two options: you can add a simple emulated device that echoes back any data that is sent to it, or add an implementation of the device using the embedded support API.

If you implement an I2C or SPI peripheral device, you can use the **I2C** or **SPI** tabs to generate events from it to the emulated device. For example, the default `Qualcomm_IoE_Device` emulator implements three I2C peripherals (an accelerometer, a light sensor, and a temperature sensor) and one SPI peripheral (an accelerometer).

On the **I2C** tab and **SPI** tab, at the top of the **G-Sensor** group, you can see the range and bandwidth of the emulated accelerometer implementation. Below you can use the sliders to specify the acceleration of the device along the three axes (X, Y, and Z).

On the **I2C** tab, in the **Light Sensor** group, you can see the type, range, resolution, illuminance limits, and whether the interrupt flag is set for the emulated light sensor implementation. Below you can use the slider to set the current illuminance level in lux.

On the **I2C** tab, in the **Temperature Sensor** group, you can see the temperature limits of the emulated temperature sensor implementation. Below you can use the slider to set the current temperature in degrees Celsius.

Generating Location Provider Information

Some devices support location tracking through JSR 179: Location API for J2ME, which enables Java ME Embedded applications to use information about the physical location of the device. A real device receives location information from a location provider.

Use the **Location** tab to generate data received by an emulated device as if from a location provider.

Set the following orientation measurements:

- **Azimuth**: Horizontal direction
- **Pitch**: Vertical elevation angle
- **Roll**: Rotation of the terminal around its longitudinal axis

By default, emulated devices have two location providers assigned to them. Select a state from the drop-down lists under **Location Provider #1** and **Location Provider #2** to test how your Java ME Embedded application handles unexpected conditions (that is, when location information is not available). The default state is **Available**. You can change it to **Temporarily Unavailable** or **Out of Service**.

Specify the latitude, longitude, altitude, speed (ground speed), and course (degrees relative to true north) for each location provider. Applications that use the Location API retrieve these values as the location of the emulator. Click **Send** to transmit these values to the emulator as if sent from a location provider.

You can also create a script to simulate the movement of the device by specifying different locations for different times. Specify the path to the location script file in the **Script** field, or click **Browse** and select the script file in the file explorer window.

Use the **Time** slider to change the starting point within the script. You can run the script, pause it, or move to the beginning or end of the script by using the buttons below the slider.

The script for the location provider is an XML file with the root `<waypoints>` element and a series of child `<waypoint>` elements. Each `<waypoint>` element must contain the `time` attribute that specifies the time in milliseconds from the previous waypoint or from the beginning. Position is specified using the `latitude`, `longitude`, and `altitude` attributes. You can specify the position for each provider separately. If only one position is present, the other one is given the same position. You can also specify the state of each provider using the `state` attribute with one of three values:

- `off`: means the provider is out of service
- `available`: means the provider is working and has data available
- `unavailable`: means the provider is working, but no data is available

The following example shows a sample script. The first waypoint is set to 1.5 seconds after beginning, the position is set only for default provider, the other provider is automatically set to the same position, both providers are available. The second waypoint is set to 2 seconds after the first one (or 3.5 seconds after beginning), the position is set only for the first provider, the second provider is automatically set to the same position, but this time first provider is out of service, and only second is available. The third waypoint is set to 3 seconds after the second one (or 6.5 seconds after beginning), the position is set only for second provider, the first provider is automatically set to the same position, but data from the first provider is not available, and only the second provider is available.

Example 8–1

```
<waypoints>
  <waypoint time=1500 latitude="14.4" longitude="50.1" altitude="310"
state1="available" state2="available" />
  <waypoint time=2000 latitude1="14.7" longitude1="49.7" altitude1="305"
state1="off" state2="available" />
  <waypoint time=3000 latitude2="14.9" longitude2="49.3" altitude2="303"
```

```
state1="unavailable" state2="available" />
</waypoints>
```

Generating Input From Memory-Mapped Peripherals

Some devices support communication with peripheral devices over the Memory-Mapped Input/Output (MMIO) interface bus. The MMIO APIs enable low-level control over the peripheral by reading from and writing to registers or memory blocks of the peripheral mapped to the memory of the emulated device.

You can either create a simple emulated peripheral device that echoes back any data that is sent to it, or add an implementation of the device using the Embedded Support API.

If you implement an MMIO peripheral device, you can use the **MMIO** tab to generate events from the peripheral to the embedded memory of the emulated device. On real devices, these events are produced by the hardware of the peripheral device.

Select an MMIO device from the **Device** drop-down list. For example, in the default implementation of the `EmbeddedDevice1` emulator, **BIG_ENDIAN_DEVICE** is the only possible choice.

Use the **Block or Register** drop-down list to select a memory block or register of the selected device, from which the event is sent.

Specify an event identifier in the **Event ID** field.

Click **Send event** to send the event from the specified device.

You can add a listener in your application to capture events with a particular identifier from a specific MMIO memory register or block. See the Javadoc for the Device I/O API, specifically the `MMIODevice.setMMIOEventListener()` methods. By default, Javadocs are located in the Oracle Java ME SDK installation directory under `docs/api`.

Generating Power Management Events

Use the **Power Management** tab to configure power settings of the emulated device. You can select between emulating an external power source or battery.

Under **Power Source**, select **Battery** to emulate a device powered by a battery, or select **External** to emulate a device powered from an external power source. If you select the battery as the power source, then specify the remaining time in seconds by using either the slider or the numeric field. If you do not want to set a specific value as the remaining time, then select **Unknown**.

Also, when the battery is selected as the power source, you can send power alerts. Select an alert from the **Power Alert** drop-down list and click **Send**. The following alerts are available:

- **Battery level is critically low**
- **Battery level is getting low**
- **Battery level has returned to normal value**
- **Phone connection is about to be terminated due to insufficient power**
- **Infrared connection is about to be terminated due to insufficient power**
- **Network connection is about to be terminated due to insufficient power**

Generating Pulses Counters Tab

Some devices can receive and count pulses (or events) sent on a digital input line, including a GPIO pin. Use the **Pulse Counters** tab to generate pulses for an emulated device with a pulse counter. The default implementation displays a counter name followed by a **Send Pulse** button.

Counter names correspond to the counters on the emulator's **Pulse Counters** tab in the main window of the Java ME Embedded Emulator. Click **Send Pulse** to send a pulse.

9

About the Java ME Embedded Security Model

Java ME Embedded applications are installed, run, closed, and restarted according to the IMlet suite life cycle described in the *Java ME Embedded Profile* specification. It defines a comprehensive security model based on *protection domains*. IMlet suites are installed into a protection domain that determines access to protected functions.

You can find the *Java ME Embedded Profile* specification in the `meep-8.0.zip` file located under `docs\api` in the Oracle Java ME SDK installation directory. In particular, the following chapters in the specification are relevant for understanding the security model:

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

The following is a general overview of how the security model works:

1. The author of the Java ME Embedded application, probably a software company, buys a signing key pair from a certificate authority (CA) and signs the IMlet suite.
2. When the IMlet suite is installed on a device, the authentication provider verifies the author's certificate using its own copy of the CA's root certificate. Then the authentication provider uses the author's certificate to verify the signature on the IMlet suite.
3. After verification, the IMlet suite is assigned 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.

Oracle Java ME SDK supports the following security provider clients by default:

- `Manufacturer`: This client is configured by the device vendor. You should not modify the list of permissions.
- `Operator`: This client has all permissions granted by default, and you can modify the list if necessary.
- `untrusted`: Defines the security policy for unsigned applications. According to the X.509 authentication scheme, unsigned applications are bound to the `untrusted` client.

9.1 Configuring the Security Policy for a Device

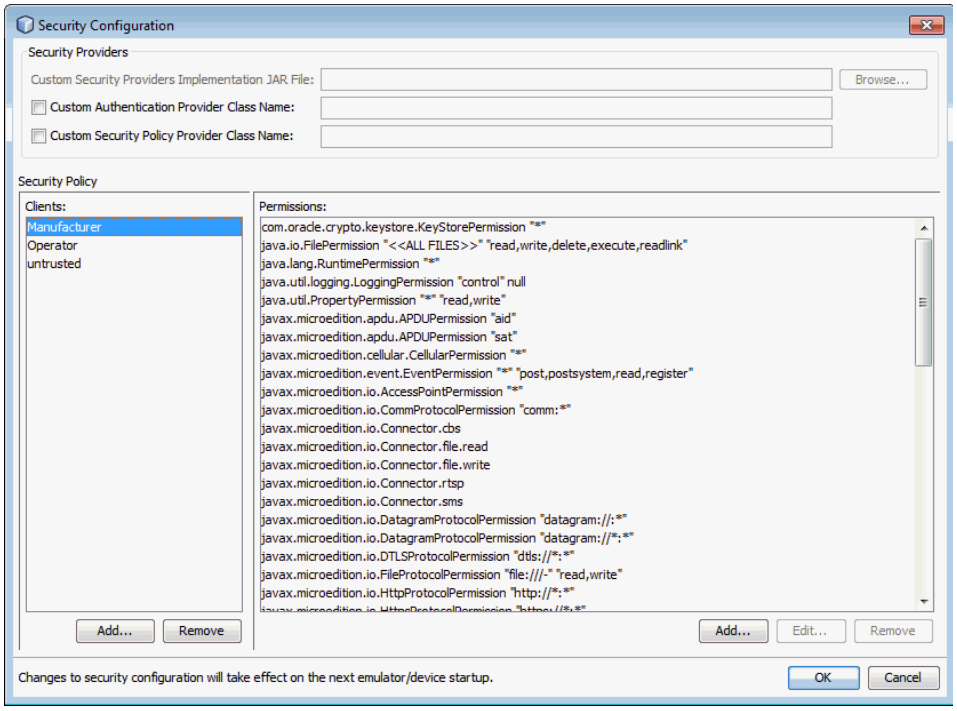
The security policy of a device defines clients to which an IMlet suite can be assigned after authentication. Each client has an associated security protection domain that defines the permissions that may be granted an application assigned to this client.

To configure the security policy for a device:

- Right-click the device in the **Device Selector** tab and select **Security Configuration**.

  Figure 9–1 shows the Security Configuration window.

Figure 9–1 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 authentication and security policy providers, see Chapter 10, "Custom Security Policy and Authentication Providers".

To add a client to the security policy, 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 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**.

9.2 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 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 Java ME Embedded Application project with a key pair in NetBeans IDE:

1. Right-click a project and select **Properties**.
2. In the **Signing** category, select **Sign JAR**.
3. Select a keystore and a key pair alias.
4. Click **OK**.

To sign a Java ME project with a key pair in Eclipse IDE:

1. Right-click a project and select **Properties**.
2. In the **Signing** category, select **Enable Project Specific Settings**.
3. Click **External** and select a keystore on the file system.
4. Enter the keystore password.
5. Select **Save password in workspace keyring**.
6. Enter the keystore password and key password in the corresponding text boxes.
7. Click **Apply**.

For an emulated device, it is also necessary to export the root certificate to the device. For more information, see Managing Root Certificates.

9.3 Managing Keystores and Key Pairs

For test purposes, you can create a signing key pair to sign an IMlet. In NetBeans IDE, the Keystores Manager administers this task. In Eclipse IDE, this is done in the global or project properties.

The instructions below relate to NetBeans IDE. If you are using Eclipse IDE, you should manage keystores and key pairs using `mekeytool`. For more information, see Manage Certificates (mekeytool).

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 in NetBeans IDE:

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 in NetBeans IDE:

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 in NetBeans IDE:

1. Open the **Tools** menu and select **Keystore Management**.
2. Select a keystore. If necessary, provide a password to unlock the 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**.

9.4 Managing Root Certificates

The Oracle Java ME SDK 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 a shared keystore and one for each of the security clients. The shared keystore file is named `_main.ks` and located under `appdb\certs` in the device's configuration directory.

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 in NetBeans IDE:

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

If you are using Eclipse IDE, you should export keys using `mekeytool`. For more information, see Manage Certificates (mekeytool).

9.5 Command-Line Security Features

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

9.5.1 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>

9.5.2 Manage Certificates (mekeytool)

`mekeytool` manages the public keys and key pairs provided by 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 DER, PKCS12, PEM, or JKS keystore. You can create one using the Java SE `keytool` utility (found in the `bin` directory under the JDK installation location) or OpenSSL.

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 shared public keys (used only for SSL/HTTPS connections) 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 (with or without private key).

> **Note:** Oracle Java ME SDK supports RSA encoded public keys stored in DER, PKCS12, PEM, and JKS formats, and private key pairs stored in PKCS12, PEM, and JKS formats.

This command has the following syntax:

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

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

-list
Lists the keys in the Java ME keystore, including the owner and validity period for each. This command has the following syntax:

-list [-client <name>]

You can specify the name of the target security client using the `-client` option.

-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>} [-client <name>]

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
`-client`	The name of the target security client	N/A

-export

Exports the key from the keystore. If only public certificate is present, it is exported in PEM format. If there is a public certificate with a private key, it is exported in PKCS12 format. This command has the following syntax:

-export -number <number> -out <filename> [-client <name>]

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
`-client`	The name of the target security client	N/A

-clients

Prints the list of security clients that can accept public keys.

10

Custom Security Policy and Authentication Providers

Device emulators in Oracle Java ME SDK are bundled with default security policy and authentication providers that can be used without any modification or configured to your needs, as described in Configuring the Security Policy for a Device. You can also create custom security policy and authentication providers, as defined in the MEEP specification.

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.

- Sample Custom Security Policy Provider
- Sample Custom Authentication Provider
- Installing Custom Providers

10.1 Sample Custom 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 Java ME Embedded 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 10–1 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 10–1 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", null);
```

```
        clientA.addPermissions(new
javax.microedition.io.HttpProtocolPermission("http://locahost:80/"),
            new javax.microedition.io.SSLProtocolPermission("ssl://:*"));
        addClient(clientA);

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

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

10.2 Sample Custom Authentication Provider

The purpose of an authentication provider is to verify a Java ME Embedded 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 10–2 shows how to create a custom authentication provider that selects clients depending on the application vendor property.

Example 10–2 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() {
    }
}
```

10.3 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_8.1\lib\ext\security_api.jar`
2. Right-click an emulated device in the **Device Selector** tab 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 for a Device".

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

11

About Java ME Demo Applications

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

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

Demo applications are installed with the Java ME SDK plug-ins for NetBeans IDE and Eclipse IDE. Additional demo applications may become available through the update center. For more information about installing the plug-ins and the update center, see Chapter 2.

The following demos are available by default:

- **Calculator Sample**: Calculates several arithmetic expressions concurrently. Demonstrates the following functionality: Multithreading, Service Loader.
- **Data Collection Demo**: Reads data from peripheral devices using Device Access API (DAAPI) and processes it. Demonstrates the following functionality: MVM, Inter-IMlet communication using local datagrams, Device I/O pulse counter, Device I/O SPI, Logging API.
- **Data Forwarder Sample**: Forwards text data received from a connected TCP client to a set of applications. Demonstrates the following functionality: MEEP IMC, MEEP Events, GCF Server Socket.
- **Directory Files Tree Sample**: Prints the tree of files and directories. Demonstrates a part of CLDC NIO files functionality.
- **GPIO Demo**: Changes the values of GPIO buttons. Demonstrates the General Purpose Input Output (GPIO) part of DAAPI.
- **I2C Demo**: Acquires a slave device named `I2C_Joystick`, writes data to it, and retrieves it. Demonstrates the usage of interfaces and classes for Inter-Integrated Circuit Bus (I2C) part of DAAPI.
- **Light Track Demo**: Flashes corresponding LEDs depending on the current value of the Analog-to-Digital (ADC) channel.
- **NetworkDemo**: Demonstrates data exchange using socket connections.

- **Network Info Sample**: Displays information about available network interfaces, access points, IP address and host reachability (ping). Demonstrates the following functionality: GCF Network Interfaces, GCF Access Points, GCF NetworkUtilities.
- **PDAPDemo**: Demonstrates the usage of JSR 75: File Connection API.
- **System Controller Sample**: Displays the state of other applications, enables you to run and stop them using a simple command-line interface, and does other application management tasks. Demonstrates the following functionality: MEEP Software Management (SWM), CLDC Logging, CLDC Timers, GCF Server Sockets, DIO GPIO Pins, DIO Watchdog, MEEP Power, Application autostart.
- **System Info Sample**: Reads system properties and information about the amount of memory used by the JVM, displays details about installed and running applications. Demonstrates part of MEEP SWM functionality.
- **Temperature Collector Sample**: Measures temperature periodically and stores results. When a remote client connects, it sends the average temperature to the client. Demonstrates the following functionality: MEEP PushRegistry Alarm, MEEP PushRegistry Push Notifications, MEEP Record Store Management (RMS), GCF Server Socket, CLDC system time.

11.1 Running Demo Applications

You can run the default demo applications from NetBeans IDE or Eclipse IDE by creating the corresponding project. Then you run the project to try out the features that the demo was created for.

To create a Java ME demo application project in NetBeans IDE:

1. Open the **File** menu and select **New Project**.
2. In the **Categories** list, expand **Samples** and select **Java ME SDK 8.1**.
3. Select the demo application from the **Projects** list and click **Next**.
4. Enter a name for the project or leave the default, change other settings as necessary, and click **Finish**.

To create a Java ME demo application project in Eclipse IDE:

1. Open the **File** menu, select **New**, and then **Example**.
2. In the **New Example** wizard, select **Java ME Sample Applications** and click **Next**.
3. Select the demo application from the list and click **Finish**.

11.2 Configuring the Web Browser and Proxy Settings

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

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

1. Open the Device Selector.
2. Select the platform name to view its properties.
3. Specify the **HTTP Proxy Settings**, **HTTP Proxy Host**, and **HTTP Proxy Port** fields to match your network and browser settings.

11.3 Troubleshooting

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

- Some demonstrations require specific setup and instructions. For example, 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.

A

Java ME Embedded Emulator Command-Line Reference

The Java ME Embedded Emulator can be started from the Windows command line.

Synopsis

emulator.exe *command* [*option* ...]

Description

When you start the emulator, you can pass a command to it that defines what it should do, and options that adjust the behavior.

Commands

The following commands can be used to define what the emulator should do:

-help

Prints the usage information.

-version

Prints the version information.

-Xautotest:<jad_file>

Runs the specified JAD file on the emulator in autotest mode. The following options cannot be used with this command:

```
-Xdebug
-Xrunjdwp
-Xprofile
-Xnetmon
```

-Xdescriptor:<jad_file>

Installs and runs the specified JAD file on the emulator in normal mode, then removes it when you close the emulator.

-Xi3test[:<option>[=<value>],...[,<testclass>]]

Runs tests on the emulator. You can specify the `testclass` argument to run just that one test. Otherwise, it runs all known tests.

The following options are available:

`-filter=<pattern>`: Runs tests which names contain the specified pattern.
`-timeout=<min>`: Sets the maximum execution time in minutes.
`-keyword=<key>`: Runs tests which keywords contain the specified key.
`-list`: Lists all known tests.

`-selftest`: Runs the framework's self test.
`-verbose`: Enables verbose output.

-Xjam[:option]
Runs the interactive Java application manager. You can pass several `-Xjam` commands with different options. The following options are available:

`force`: Can be used in conjunction with `install` to force the removal of an IMlet that is already present in a storage name.
`install=<jad_url>`: Installs the IMlet from the specified URL.
`list`: Lists all installed IMlets and exits.
`remove={<name>|<number>}`: Removes the IMlet in the specified storage name or number. The system-defined application `all` can be used to remove all IMlets.
`run={<name>|<number>}`: Runs the IMlet in the specified storage name or number.
`storagreNames`: Lists all storage names in the order of assigned storage numbers and exits.
`transient=<jad_url>`: Installs, runs, and removes the IMlet from the specified URL.

-Xquery
Prints information about available devices and exits. If used with the `-Xdevice` option, prints information only about the specified device.

-XshutdownAll
Closes the emulator frame and shuts down the device manager.

Options

The following options can be used to adjust the behavior of the emulator:

-D<property>=<value>
Sets the system property to a value. This option can be used during development to pass parameters to an application without rebuilding and repackaging it.

-Xdebug
Enables runtime debugging. This option should be used with the `-Xrunjdwp` option that runs and controls a Java Debug Wire Protocol (JDWP) session.

-Xdevice
Runs the specified device in the emulator. If this option is not specified, the default device is used, except for the `-Xquery` command that prints information about all available devices if `-Xdevice` is not present.

-Xmemmon
Enables the Memory Monitor.

-Xnetmon
Enables the Network Monitor.

-Xprofile[:file=<file>]
Enables the CPU Profiler. You can use the `file` argument to specify the file to which the snapshot should be stored. If you do not specify the `file` argument, profiler data will be passed on to a connected profiler and not stored as a snapshot.

-Xrunjdwp:<name>=<value>,...
Runs and controls a Java Debug Wire Protocol (JDWP) session when the `-Xdebug` option is enabled. You should specify at least one name and value pair. Multiple pairs are separated by commas. The following names are available:

address: Specifies the address for the debugger connection. The value is specified as a host name and port number separated by a colon. If you specify only the port number, than localhost is assumed as the host name.
server: Specifies whether to start the debug agent as a server. The value is specified as y or n. By default, it is set to n.
suspend: Specifies whether to suspend the JVM immediately after establishing a connection with the debugger. The value is specified as y or n. By default, it is set to n.
transport: Specifies the transport mechanism used to communicate with the debugger. The default value is dt_socket.

Examples

The following example runs the sample_imlet.jad file on EmbeddedDevice1, assuming Oracle Java ME SDK was installed to the default location:

```
C:\Java_ME_platform_SDK_8.1\bin> emulator.exe -Xdevice:EmbeddedDevice1
-Xdescriptor:C:\Java_ME_platform_SDK_8.1\apps\sample\sample_imlet.jad
```

B

Installation and Runtime Security Guidelines

Oracle Java ME SDK 8.1 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 8.1 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 8.1 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 8.1.

B.1 Maintaining Optimum Network Security

To maintain optimum network security, Oracle Java ME SDK 8.1 can be installed and run in an isolated network environment, where the Oracle Java ME SDK 8.1 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 8.1 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 8.1 application execution process. Whether or not an Internet connection is required depends on the particular application running on Oracle Java ME SDK 8.1. For example, some applications can use an HTTP connection.

If Oracle Java ME SDK 8.1 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 8.1 behind a secure firewall that strictly limits unauthorized network access to the Oracle Java ME SDK 8.1 file system and services. Limit access privileges to those that are required for Oracle Java ME SDK 8.1 usage while allowing all the bidirectional local network communications that are necessary for Oracle Java ME SDK 8.1 functionality. The firewall configuration must support these requirements to run the Oracle Java ME SDK 8.1 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 8.1.
- Do not store any sensitive information on the same file system that is hosting Oracle Java ME SDK 8.1.
- To maintain the maximum level of security, ensure that all the latest updates for the operating system are installed.